LNAI 6173

Jürgen Giesl
Reiner Hähnle (Eds.)

Automated Reasoning

5th International Joint Conference, IJCAR 2010
Edinburgh, UK, July 2010
Proceedings

Springer

Jürgen Giesl   Reiner Hähnle (Eds.)

Automated Reasoning

5th International Joint Conference, IJCAR 2010
Edinburgh, UK, July 16-19, 2010
Proceedings

Springer

Series Editors

Randy Goebel, University of Alberta, Edmonton, Canada
Jörg Siekmann, University of Saarland, Saarbrücken, Germany
Wolfgang Wahlster, DFKI and University of Saarland, Saarbrücken, Germany

Volume Editors

Jürgen Giesl
RWTH Aachen, LuFG Informatik 2
Ahornstr. 55, 52074 Aachen, Germany
E-mail: giesl@informatik.rwth-aachen.de

Reiner Hähnle
Chalmers University of Technology
Department of Computer Science
41296 Gothenburg, Sweden
E-mail: reiner@chalmers.se

Library of Congress Control Number: 2010929611

CR Subject Classification (1998): F.4.1, I.2.3, F.3, I.2, D.2.4, D.1.6

LNCS Sublibrary: SL 7 – Artificial Intelligence

ISSN	0302-9743
ISBN-10	3-642-14202-8 Springer Berlin Heidelberg New York
ISBN-13	978-3-642-14202-4 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper   06/3180

Preface

This volume contains the proceedings of the 5th International Joint Conference on Automated Reasoning (IJCAR 2010). IJCAR 2010 was held during July 16-19 as part of the 2010 Federated Logic Conference, hosted by the School of Informatics at the University of Edinburgh, Scotland. Support by the conference sponsors – EPSRC, NSF, Microsoft Research, Association for Symbolic Logic, CADE Inc., Google, Hewlett-Packard, Intel – is gratefully acknowledged.

IJCAR is the premier international joint conference on all topics in automated reasoning, including foundations, implementations, and applications. Previous IJCAR conferences were held at Siena (Italy) in 2001, Cork (Ireland) in 2004, Seattle (USA) in 2006, and Sydney (Australia) in 2008. IJCAR comprises several leading conferences and workshops. In 2010, IJCAR was the fusion of the following events:

- **CADE:** International Conference on Automated Deduction
- **FroCoS:** International Symposium on Frontiers of Combining Systems
- **FTP:** International Workshop on First-Order Theorem Proving
- **TABLEAUX:** International Conference on Automated Reasoning with Analytic Tableaux and Related Methods

There were 89 submissions (63 regular papers and 26 system descriptions) of which 40 were accepted (28 regular papers and 12 system descriptions). Each submission was assigned to at least three Program Committee members, who carefully reviewed the papers, with the help of 92 external referees. Afterwards, the submissions were discussed by the Program Committee during two weeks by means of Andrei Voronkov's *EasyChair* system. We want to thank Andrei very much for providing his system, which was very helpful for the management of the submissions and reviews and for the discussion of the Program Committee. We are extremely grateful to the members of the Program Committee and to the external reviewers for all of their hard work in putting together an excellent technical program spanning the entire spectrum of research in automated reasoning.

IJCAR 2010 had invited talks by Leonardo de Moura and Johan van Benthem. Moreover, together with CAV, CSF, and ICLP, IJCAR had an invited plenary talk by David Basin and an invited keynote talk by Deepak Kapur. These proceedings contain a full paper and two abstracts corresponding to the invited talks. We want to thank the invited speakers for their interesting and inspiring presentations.

The Herbrand Award for distinguished contributions to automated reasoning was given to David Plaisted in recognition of his many outstanding seminal results in the field. The selection committee for the Herbrand Award included the previous award winners of the last ten years, the CADE trustees, and the IJCAR

2010 Program Committee. The Herbrand Award ceremony and the acceptance speech were part of the conference program.

We are grateful to the IJCAR 2010 Workshop Chair Aaron Stump for attracting the following workshops that took place in association with IJCAR:

- **Automatheo:** Workshop on Automated Mathematical Theory Exploration
- **CLoDeM:** International Workshop on Comparing Logical Decision Methods
- **EMSQMS:** Workshop on Evaluation Methods for Solvers and Quality Metrics for Solutions
- **LfSA:** Logics for System Analysis
- **MLPA:** Workshop on Module Systems and Libraries for Proof Assistants
- **PAAR:** Workshop on Practical Aspects of Automated Reasoning
- **SVARM:** Synthesis, Verification and Analysis of Rich Models
- **UITP:** International Workshop on User Interfaces for Theorem Provers
- **UniDL:** Workshop on Uncertainty in Description Logics
- **UNIF:** International Workshop on Unification
- **VERIFY:** International Verification Workshop
- **WING:** Workshop on Invariant Generation
- **WST:** International Workshop on Termination

Moreover, the following competitions were held as part of IJCAR:

- **CASC-J5:** 5th IJCAR ATP System Competition
- **Termination Competition 2010:** 7th International Termination Competition

Many people helped to make IJCAR 2010 a success. We want to thank the IJCAR Publicity Chair Viorica Sofronie-Stokkermans and we are grateful to Geoff Sutcliffe for hosting the IJCAR website. The IJCAR 2010 Steering Committee consisted of Peter Baumgartner, Maria Paola Bonacina (Chair), Alan Bundy, Jürgen Giesl, Rajeev Goré, Bernhard Gramlich, Reiner Hähnle, and Ullrich Hustadt. We are particularly indebted to the IJCAR 2010 Conference Chair and IJCAR representative in the FLoC Steering Committee Alan Bundy, the FLoC Steering Committee Chair Moshe Y. Vardi, the FLoC Conference Chairs Leonid Libkin and Gordon Plotkin, and all the other members of the FLoC Organizing Committee and the Local Organizing Committee, who organized IJCAR and the other conferences and workshops of FLoC in a very careful way.

May 2010

Jürgen Giesl
Reiner Hähnle

Conference Organization

Program Chairs

Jürgen Giesl
Reiner Hähnle

Program Committee

Carlos Areces
Alessandro Armando
Franz Baader
Peter Baumgartner
Bernhard Beckert
Christoph Benzmüller
Nikolaj Bjørner
Maria Paola Bonacina
Alan Bundy
Gilles Dowek
Christian Fermüller
Ulrich Furbach
Didier Galmiche
Silvio Ghilardi
Martin Giese
Rajeev Goré
Bernhard Gramlich
Ullrich Hustadt
Deepak Kapur
Viktor Kuncak
Rustan Leino
Carsten Lutz
George Metcalfe
Aart Middeldorp
Neil Murray
Tobias Nipkow
Nicola Olivetti
Nicolas Peltier
Frank Pfenning
Brigitte Pientka
Andre Platzer
Christophe Ringeissen
Albert Rubio
Renate A. Schmidt
Carsten Schürmann
Roberto Sebastiani
Viorica Sofronie-Stokkermans
Aaron Stump
Geoff Sutcliffe
Cesare Tinelli
Ashish Tiwari
Andrei Voronkov
Christoph Weidenbach

Conference Chair

Alan Bundy

Workshop Chair

Aaron Stump

Publicity Chair

Viorica Sofronie-Stokkermans

Local Organization

Leonid Libkin, Gordon Plotkin, Philip Scott, Nicole Schweikardt, Stephan Kreutzer, Seth Fogarty, Floris Geerts, Kousha Etessami, Anuj Dawar, Bartek Klin, Perdita Stevens, Claire David, Ian Stark, Paul Jackson, Jacques Fleuriot

External Reviewers

Vincent Aravantinos
Serge Autexier
Taus Brock-Nannestad
Thomas Bolander
Thorsten Bormer
Thierry Boy de la Tour
Daniel Bruns
Ricardo Caferra
Serenella Cerrito
Kaustuv Chaudhuri
Yannick Chevalier
Ranald Clouston
Jeremy Dawson
Stéphane Demri
Mnacho Echenim
Bernd Finkbeiner
Camillo Fiorentini
Pascal Fontaine
Oliver Friedmann
Sicun Gao
Laura Giordano
Birte Glimm
Valentina Gliozzi
Amit Goel
Daniel Gorín
Alberto Griggio
Gudmund Grov
Joe Hendrix
Thomas Hillenbrand
Carsten Ihlemann
Swen Jacobs
Manfred Jaeger
Emil Jerabek
Moa Johansson
Yevgeny Kazakov
Vladimir Klebanov
Martin Korp
Markus Krötzsch
Alexander Kurz
Ralf Küsters
Roman Kuznets
Dominique Larchey-Wendling
Stéphane Lengrand
Espen H. Lian
Sven Linker
Salvador Lucas
Michael Ludwig
Thomas Lukasiewicz
João Marcos
Andrew Matusiewicz
Guillaume Melquiond
Daniel Méry
Ralf Möller
César Muñoz
Enrica Nicolini
Ligia Nistor
Albert Oliveras
Luigi Palopoli
Björn Pelzer
Rafael Peñaloza
Adam Poswolsky
Gian Luca Pozzato
Florian Rabe
Silvio Ranise
Hilverd Reker
David Renshaw
Enric Rodríguez-Carbonell
Philipp Rümmer
Gernot Salzer
Andreas Schaefer
Luis Menasché Schechter
Andreas Schnabl
Lutz Schröder
Gert Smolka
Christian Sternagel
Mark Stickel
Audun Stolpe
Lutz Straßburger
Thomas Studer
Philippe Suter
René Thiemann
Stefano Tonetta
Xavier Urbain
Vincent van Oostrom
Helmut Veith
Michele Vescovi
Laurent Vigneron
Uwe Waldmann
Edwin Westbrook
Florian Widmann
Sarah Winkler
Patrick Wischnewski

Table of Contents

Curry-Style Explicit Substitutions for the Linear and Affine Lambda Calculus[⋆]

Anders Schack-Nielsen and Carsten Schürmann

IT University of Copenhagen
Copenhagen, Denmark
{anderssn,carsten}@itu.dk

Abstract. We introduce a calculus of explicit substitutions for the λ-calculus with linear, affine, and intuitionistic variables and meta-variables. Using a Curry-style formulation, we redesign and extend previously suggested type systems for linear explicit substitutions. This way, we obtain a fine-grained small-step reduction semantics suitable for efficient implementation. We prove that subject reduction, confluence, and termination holds. All theorems have been formally verified in the Twelf proof assistant.

1 Introduction

Explicit substitutions [1,7,13] are central for modern implementations of systems that provide mechanisms for variable binding, such as logical frameworks [14], theorem provers [8,2], proof assistants [3], and programming language implementations [19,12,18,15] and analysis [6]. Many of these systems, especially proof assistants, employ meta-variables as a means to implement proof search and type reconstruction. As substructural logics become more prevalent — examples include separation logic, the logic of bunched implications, and even concurrent process calculi — we are faced with the challenge of engineering *sound*, *efficient*, and *implementable* explicit substitution calculi for substructural λ-calculi that give us confidence about our design decisions and attest to the reliability of the final product.

When denoting a λ-term using de Bruijn indices, we usually use one of two commonly accepted notation styles. Following Church, we encode the simply-typed term $\lambda x : \mathsf{a}. x$ (of type $\mathsf{a} \to \mathsf{a}$) as $\lambda \cdot : \mathsf{a}. 1$ where 1 refers to the innermost binder occurrence. Following Curry, we simplify this encoding, omit the type label, and write $\lambda 1$. Church style encodings are a bit more verbose, but the real disadvantage lies in the additional code that has to be written to keep type labels up to date, for example in substitution application for dependently typed λ-terms. Therefore, for a real implementation, the brevity of Curry-style notation renders it preferable. However, in reality, it is not as easily adopted, in part because one needs to be sure a priori that type labels are indeed irrelevant

⋆ This work was in part supported by NABITT grant 2106-07-0019 of the Danish Strategic Research Council.

or at the very least efficiently inferable whenever necessary using techniques such as, for example, bi-directional type checking.

Further differences between the two notation systems become apparent if we consider substructural λ-calculi. Linear λ-calculi [4] are of interest predominantly to the logical framework community because they permit elegant and direct encodings of formal languages that consume resources. A linear λ-term binds a resource that must be consumed (referred to) in the body of the term *exactly once*. For two reasons, affine λ-calculi are of interest as well. First, affine types are useful when modelling programming languages with state, because heap cells behave similarly to affine resources. Second, affine types have proven useful in the development of linear unification algorithms [17]. The difference to linear resources is that affine resources must be consumed *at most once*, which means *exactly once* or *not at all*.

Consider, for example, the term $M \hat{~} N$ (pronounced M linearly applied to N) of type B in some context Γ. Given the standard formulation of the $\multimap$ elimination rule

$$\frac{\Gamma_1 \vdash M : A \multimap B \quad \Gamma_2 \vdash N : A}{\Gamma_1, \Gamma_2 \vdash M \hat{~} N : B} \multimap \mathsf{E}$$

it seems impossible to derive how to split Γ into Γ_1 and Γ_2 without examining the structure of M or N. Even worse, deriving the context split might be further complicated if M or N contain meta-variables.

When working with explicit substitutions this poses a real problem, since substitutions must correspond to the context they are substituting for. Furthermore, this means that the reduction $(M \hat{~} N)[s] \rightarrow M[s_1] \hat{~} N[s_2]$ depends on the context split.

The lack of information on how to split the context in the Curry-style encoding suggests that the alternative Church-style version $M \; \widehat{\Gamma_1 \bowtie \Gamma_2} \; N$ is to be preferred. This version, however, renders terms, types, and contexts mutually dependent, which puts a significant additional burden on the implementation effort. In fact, we suspect that the prospect of the unwieldy complexity is at least in part responsible for that there exist only so few implementations of substructural logical frameworks, theorem provers, and other linear logic based systems.

The problem is not new but has been observed in the past. Cervesato et al. [5] give a good overview and discuss various suggestions and their shortcomings. But none of the suggestions are satisfactory as they are either too cumbersome to be usable in practice or lack basic properties such as subject reduction. Additionally, none of the suggestions scale to meta-variables.

In this paper we demonstrate that Curry-style explicit substitutions can be made to work. We define a type system for a linear and affine λ-calculus with explicit substitutions and meta-variables together with a Curry-style reduction semantics. We prove subject reduction, confluence, and termination for our calculus. This means that we can guarantee type preservation of the reduction $(M \hat{~} N)[s] \rightarrow M[s] \hat{~} N[s]$ without splitting the substitution, and that our calculus therefore permits much more concise data structures in implementations than previously suggested type systems for linear λ-calculi.

The main contributions of this work are the subject reduction and confluence theorems. Subject reduction is achieved by a novel type system allowing controlled occurrence of garbage terms (see the rules **typ-cons-ua** and **typ-cons-ul** in Figure 3). Confluence on terms with meta-variables is achieved by a limited η-expansion of substitutions.

All theorems proven in this paper have been mechanically checked by Twelf. The formalized theorems are annotated with the corresponding source file in which the proof can be found. All Twelf source files can be downloaded at `http://www.twelf.org/~celf/download/ex-sub-aff.tgz`. As for empirical evidence, we have implemented our calculus as the core data structure of the Celf system [16]. The system implements several algorithms, including hereditary substitutions, higher-order affine and linear unification, type reconstruction, and a logic programming inspired proof search engine. Our concise Curry-style representation proved to be both sufficient and elegant in all these extensive applications.

This paper is organized as follows. In Section 2 we define the explicit substitution calculus. The type system is defined in Section 3, and the reduction semantics is defined in Section 4. Section 5 gives a brief sketch of the extension to dependent types. Finally, Section 6 concludes.

2 Explicit Substitutions

We define a calculus of explicit substitutions for the linear and affine λ-calculus. In order to simplify the presentation we restrict attention to the simply typed fragment. A generalization to dependent types is direct, but omitted for presentation purposes. We will sketch the extension to dependent types in Section 5. Our calculus denotes variables in de Bruijn notation, since it is closer to a real implementation, avoids naming problems due to α-conversion, and highlights the nature of the explicit substitutions.

$$\begin{array}{ll}
\textbf{Types:} & A, B ::= a \mid A \mathbin{\&} B \mid A \multimap B \mid A \mathbin{-@} B \mid A \to B \\
\textbf{Terms:} & M, N ::= 1 \mid M[s] \mid \langle M, N\rangle \mid \mathsf{fst}\ M \mid \mathsf{snd}\ M \mid X[s] \\
 & \qquad \mid \widehat{\lambda} M \mid \mathring{\lambda} M \mid \lambda M \mid M\,\hat{}\,N \mid M@N \mid M\ N \\
\textbf{Substitutions:} & s, t ::= \mathsf{id} \mid \uparrow \mid M^f.s \mid s \circ t \\
\textbf{Linearity flags:} & f ::= \mathbf{I} \mid \mathbf{A} \mid \mathbf{L}
\end{array}$$

Types consist of base types (a), additive pairs ($A \mathbin{\&} B$), linear functions ($A \multimap B$), affine functions ($A \mathbin{-@} B$), and intuitionistic functions ($A \to B$).

Terms consist of the various introduction and elimination forms for each of the type constructs along with variable indices (1), meta-variables (X), and closures ($M[s]$). Variables n with $n > 1$ do not need to be included explicitly in the syntax, since they can be represented by a closure as described below. We require each meta-variable to be under a closure as it gives rise to more uniform normal forms; an alternative would have been to add the "reduction" rule $X \to X[\mathsf{id}]$. Meta-variables are also called logic variables.

Substitutions are composed from identity, shifts, intuitionistic ($M^{\mathbf{I}}$), affine ($M^{\mathbf{A}}$), and linear ($M^{\mathbf{L}}$) extensions, and an explicit substitution composition. In the interest of readability we introduce a syntactic category f for linearity flags.

Example 1. Consider the term $(\widehat{\lambda} 1 \,\hat{}\, 2) \,\hat{}\, 2$ where we write 2 as a shorthand for $1[\uparrow]$. In a named representation, this term is written as $(\widehat{\lambda} x.\, x \,\hat{}\, y) \,\hat{}\, z$. The 1 refers to the variable bound by the $\widehat{\lambda}$, the first 2 to the first free variable (because it is in the scope of the $\widehat{\lambda}$), and the second 2 to the other free variable. Explicit substitutions are used to represent an intermediate stage during ordinary β-reduction. Our term reduces to $(1 \,\hat{}\, 2)[2^{\mathbf{L}}.\mathsf{id}]$ where the $\mathbf{L}$ flag signifies that we are substituting for a linear variable.

The intuition behind each of the substitution constructs is the following: A term under an identity is supposed to reduce to itself. A shift applied to a term increments all freely occurring variables by one. An extension $M^f.s$ will substitute M for the variable 1, decrement all other freely occurring variables, and then apply s to them. Finally a composition of two substitutions $s \circ t$ represents the substitution that first applies s and then t, i.e. $M[s \circ t]$ is supposed to reduce to the same term as reducing each closure individually in $M[s][t]$.

We will use $\uparrow^n$ where $n \geq 0$ as a short-hand for n compositions of shift, i.e. $\uparrow \circ (\uparrow \circ (\ldots \circ (\uparrow \circ \uparrow) \ldots))$, where $\uparrow^0$ means id. Additionally, de Bruijn indices n with $n > 1$ are short-hand for $1[\uparrow^{n-1}]$.

3 The Type System

Before we get to the actual typing judgments we introduce contexts.

3.1 Contexts

Since we are using de Bruijn indices, contexts are simply ordered lists of types without any names. Looking up a variable n in a context Γ amounts to selecting the nth element of Γ. This means that the usual context splitting from named versions of linear λ-calculus has to be redefined, as this would otherwise ruin the meaning of de Bruijn variables. We have to essentially introduce dummy elements in the context whenever we split, in order to maintain the correct position of every type in the context. These dummy elements are in some presentations [5] written as $\Gamma \widehat{,} \_$. Alternatively, one may view this as never actually splitting the context, but instead maintaining a bit-vector with the same length as the context, which indicates whether each particular variable is available.

For a concise representation we will superimpose the bit-vector signifying availability on the context along with the information about whether the declaration is linear, affine, or intuitionistic. This information is called a context linearity flag.

Contexts:	$\Gamma ::= \cdot \mid \Gamma, A^l$
Context linearity flags:	$l ::= f \mid \mathbf{U_L} \mid \mathbf{U_A}$

$$\frac{}{\cdot = \cdot \bowtie \cdot}\textbf{join-nil} \qquad \frac{\Gamma = \Gamma_1 \bowtie \Gamma_2}{\Gamma, A^{\mathbf{I}} = \Gamma_1, A^{\mathbf{I}} \bowtie \Gamma_2, A^{\mathbf{I}}}\textbf{join-i}$$

$$\frac{\Gamma = \Gamma_1 \bowtie \Gamma_2}{\Gamma, A^{\mathbf{U_L}} = \Gamma_1, A^{\mathbf{U_L}} \bowtie \Gamma_2, A^{\mathbf{U_L}}}\textbf{join-l-used}$$

$$\frac{\Gamma = \Gamma_1 \bowtie \Gamma_2}{\Gamma, A^{\mathbf{L}} = \Gamma_1, A^{\mathbf{L}} \bowtie \Gamma_2, A^{\mathbf{U_L}}}\textbf{join-l-L} \qquad \frac{\Gamma = \Gamma_1 \bowtie \Gamma_2}{\Gamma, A^{\mathbf{L}} = \Gamma_1, A^{\mathbf{U_L}} \bowtie \Gamma_2, A^{\mathbf{L}}}\textbf{join-l-R}$$

$$\frac{\Gamma = \Gamma_1 \bowtie \Gamma_2}{\Gamma, A^{\mathbf{U_A}} = \Gamma_1, A^{\mathbf{U_A}} \bowtie \Gamma_2, A^{\mathbf{U_A}}}\textbf{join-a-used}$$

$$\frac{\Gamma = \Gamma_1 \bowtie \Gamma_2}{\Gamma, A^{\mathbf{A}} = \Gamma_1, A^{\mathbf{A}} \bowtie \Gamma_2, A^{\mathbf{U_A}}}\textbf{join-a-L} \qquad \frac{\Gamma = \Gamma_1 \bowtie \Gamma_2}{\Gamma, A^{\mathbf{A}} = \Gamma_1, A^{\mathbf{U_A}} \bowtie \Gamma_2, A^{\mathbf{A}}}\textbf{join-a-R}$$

Fig. 1. Context splitting

The usual linear, affine, and intuitionistic assumptions are written as $\Gamma, A^{\mathbf{L}}$, $\Gamma, A^{\mathbf{A}}$, and $\Gamma, A^{\mathbf{I}}$, respectively. We flag a declaration with $\mathbf{U_A}$ to denote that an affine assumption is not available, and similarly flag unavailable linear assumptions with $\mathbf{U_L}$.

The standard definition of the context splitting judgment $\Gamma = \Gamma_1 \bowtie \Gamma_2$ (or context joining judgment depending on the direction it is being read) is shown in Figure 1.

We will introduce a couple of auxiliary definitions to do with context splitting and context linearity flag management. Any context may be trivially split into $\Gamma = \Gamma \bowtie \Gamma'$ by putting all affine and linear assumptions to the left. This means that Γ' will consist of only the intuitionistic parts of Γ. We will denote this by $\overline{\Gamma}$ and make it into a separate definition.

$$\overline{\cdot} = \cdot \qquad \overline{\Gamma, A^{\mathbf{I}}} = \overline{\Gamma}, A^{\mathbf{I}} \qquad \overline{\Gamma, A^{\mathbf{L}}} = \overline{\Gamma}, A^{\mathbf{U_L}} \qquad \overline{\Gamma, A^{\mathbf{A}}} = \overline{\Gamma}, A^{\mathbf{U_A}}$$

$$\overline{\Gamma, A^{\mathbf{U_L}}} = \overline{\Gamma}, A^{\mathbf{U_L}} \qquad \overline{\Gamma, A^{\mathbf{U_A}}} = \overline{\Gamma}, A^{\mathbf{U_A}}$$

We will need to make reference to the largest context that can split to a given context. This is denoted $\underline{\Gamma}$ and defined easily by changing every $\mathbf{U_L}$ to $\mathbf{L}$ and $\mathbf{U_A}$ to $\mathbf{A}$.

$$\underline{\cdot} = \cdot \qquad \underline{\Gamma, A^{\mathbf{I}}} = \underline{\Gamma}, A^{\mathbf{I}} \qquad \underline{\Gamma, A^{\mathbf{L}}} = \underline{\Gamma}, A^{\mathbf{L}} \qquad \underline{\Gamma, A^{\mathbf{A}}} = \underline{\Gamma}, A^{\mathbf{A}}$$

$$\underline{\Gamma, A^{\mathbf{U_L}}} = \underline{\Gamma}, A^{\mathbf{L}} \qquad \underline{\Gamma, A^{\mathbf{U_A}}} = \underline{\Gamma}, A^{\mathbf{A}}$$

Additionally, we will need a predicate on contexts specifying that there are no linear assumptions. We write this predicate as $\mathsf{nolin}(\Gamma)$ and it is defined to be true iff no context linearity flag in Γ is $\mathbf{L}$.

Notice that $\Gamma = \overline{\Gamma}$ implies $\mathsf{nolin}(\Gamma)$ whereas the opposite does not hold, since $\mathsf{nolin}(\Gamma)$ does not preclude occurrences of $\mathbf{A}$ in Γ.

$$\frac{\mathsf{nolin}(\Gamma)}{\Gamma, A^f \vdash 1 : A}\,\textbf{typ-var} \qquad \frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash \langle M, N\rangle : A \mathbin{\&} B}\,\textbf{typ-pair}$$

$$\frac{\Gamma \vdash M : A \mathbin{\&} B}{\Gamma \vdash \mathsf{fst}\ M : A}\,\textbf{typ-fst} \qquad \frac{\Gamma \vdash M : A \mathbin{\&} B}{\Gamma \vdash \mathsf{snd}\ M : B}\,\textbf{typ-snd}$$

$$\frac{\Gamma, A^{\mathbf{L}} \vdash M : B}{\Gamma \vdash \widehat{\lambda} M : A \multimap B}\,\textbf{typ-lam-l} \qquad \frac{\Gamma, A^{\mathbf{A}} \vdash M : B}{\Gamma \vdash \mathring{\lambda} M : A \mathrel{-\!@} B}\,\textbf{typ-lam-a}$$

$$\frac{\Gamma = \Gamma_1 \bowtie \Gamma_2 \quad \Gamma_1 \vdash M : A \multimap B \quad \Gamma_2 \vdash N : A}{\Gamma \vdash M\hat{}N : B}\,\textbf{typ-app-l}$$

$$\frac{\Gamma = \Gamma_1 \bowtie \Gamma_2 \quad \mathsf{nolin}(\Gamma_2) \quad \Gamma_1 \vdash M : A \mathrel{-\!@} B \quad \Gamma_2 \vdash N : A}{\Gamma \vdash M@N : B}\,\textbf{typ-app-a}$$

$$\frac{\Gamma, A^{\mathbf{I}} \vdash M : B}{\Gamma \vdash \lambda M : A \to B}\,\textbf{typ-lam-i} \qquad \frac{\Gamma \vdash M : A \to B \quad \overline{\Gamma} \vdash N : A}{\Gamma \vdash M\ N : B}\,\textbf{typ-app-i}$$

$$\frac{\Gamma \vdash s : \Gamma' \quad \Gamma' \vdash M : A}{\Gamma \vdash M[s] : A}\,\textbf{typ-clos} \qquad \frac{\Gamma \vdash s : \Gamma_X}{\Gamma \vdash X[s] : A_X}\,\textbf{typ-metavar}$$

Fig. 2. Typing of terms

Finally, we will need an affine weakening relation $\Gamma \succ_{\text{aff}} \Gamma'$, which is defined as

$$\Gamma \succ_{\text{aff}} \Gamma' \;\equiv\; \exists \Gamma''.\ \Gamma = \Gamma'' \bowtie \Gamma' \;\wedge\; \mathsf{nolin}(\Gamma'')$$

Notice that affine weakening is reflexive and transitive, as it merely amounts to changing some number of **A**s into $\mathbf{U_A}$s.

3.2 Types

The typing judgments for terms and substitutions are denoted $\Gamma \vdash M : A$ and $\Gamma \vdash s : \Gamma'$, respectively. The typing rules are shown in Figures 2 and 3. In both cases the Γ describes the types and availability of the free variables, and in the case of substitution typing, Γ' describes the context that s substitutes for.

Each meta-variable X carries its own context Γ_X and type A_X as referenced in the typing rule **typ-metavar**. This is equivalent to introducing a new contextual modal context of the form $\Delta ::= \cdot \mid \Delta, (X :: A \ \mathsf{in}\ \Gamma)$. In order to avoid clutter, we omit this additional context from the judgments, since it would simply remain constant and be copied everywhere. Instead we keep the lookup implicit by writing Γ_X and A_X to mean that $X :: A_X \ \mathsf{in}\ \Gamma_X$ is in Δ.

The **typ-cons-i**, **typ-cons-a**, and **typ-cons-l** are the natural typing rules for substitution extensions. A Church-style explicit substitution calculus would then add two new syntactic substitution constructs, $\bot^{\mathbf{U_L}}.s$ and $\bot^{\mathbf{U_A}}.s$, to correspond to the contexts $\Gamma', A^{\mathbf{U_L}}$ and $\Gamma', A^{\mathbf{U_A}}$ with the following typing rules.

$$\frac{\Gamma \vdash s : \Gamma'}{\Gamma \vdash \bot^{\mathbf{U_L}}.s : \Gamma', A^{\mathbf{U_L}}}\,\textbf{typ-cons-ul'} \qquad \frac{\Gamma \vdash s : \Gamma'}{\Gamma \vdash \bot^{\mathbf{U_A}}.s : \Gamma', A^{\mathbf{U_A}}}\,\textbf{typ-cons-ua'}$$

$$\frac{\Gamma \succ_{\text{aff}} \Gamma'}{\Gamma \vdash \mathsf{id} : \Gamma'}\ \textbf{typ-id} \qquad \frac{\Gamma \succ_{\text{aff}} \Gamma' \quad l \in \{\mathbf{I}, \mathbf{A}, \mathbf{U_L}, \mathbf{U_A}\}}{\Gamma, A^l \vdash \uparrow : \Gamma'}\ \textbf{typ-shift}$$

$$\frac{\Gamma = \Gamma_1 \bowtie \Gamma_2 \quad \Gamma_1 \vdash M : A \quad \Gamma_2 \vdash s : \Gamma'}{\Gamma \vdash M^{\mathbf{L}}.s : \Gamma', A^{\mathbf{L}}}\ \textbf{typ-cons-l}$$

$$\frac{\Gamma = \Gamma_1 \bowtie \Gamma_2 \quad \mathsf{nolin}(\Gamma_1) \quad \Gamma_1 \vdash M : A \quad \Gamma_2 \vdash s : \Gamma'}{\Gamma \vdash M^{\mathbf{A}}.s : \Gamma', A^{\mathbf{A}}}\ \textbf{typ-cons-a}$$

$$\frac{\underline{\Gamma} \vdash_i M : A \quad \Gamma \vdash s : \Gamma'}{\Gamma \vdash M^{\mathbf{L}}.s : \Gamma', A^{\mathbf{U_L}}}\ \textbf{typ-cons-ul} \qquad \frac{\underline{\Gamma} \vdash_i M : A \quad \Gamma \vdash s : \Gamma'}{\Gamma \vdash M^{\mathbf{A}}.s : \Gamma', A^{\mathbf{U_A}}}\ \textbf{typ-cons-ua}$$

$$\frac{\overline{\Gamma} \vdash M : A \quad \Gamma \vdash s : \Gamma'}{\Gamma \vdash M^{\mathbf{I}}.s : \Gamma', A^{\mathbf{I}}}\ \textbf{typ-cons-i} \qquad \frac{\Gamma \vdash s_2 : \Gamma'' \quad \Gamma'' \vdash s_1 : \Gamma'}{\Gamma \vdash s_1 \circ s_2 : \Gamma'}\ \textbf{typ-comp}$$

Fig. 3. Typing of substitutions

These substitution constructs read as: do not substitute anything for the variable 1 (since it does not occur), but decrement all free variables by one and apply s. This approach has been tried [9] and yields the problem described in the introduction, i.e. the reduction $(M \hat{} N)[s] \to M[s_1] \hat{} N[s_2]$ needs to split s according to the context split and thus cannot be performed without this additional information.

Our solution to this problem is to reuse the syntax $M^{\mathbf{L}}.s$ and $M^{\mathbf{A}}.s$ where we would expect $\perp^{\mathbf{U_L}}.s$ and $\perp^{\mathbf{U_A}}.s$. The idea is then to perform the substitution splitting on the typing judgments instead of the syntax — this is the crucial switch from the Church-style approach to our Curry-style formulation. However, we cannot reuse the typing rules **typ-cons-ul'** and **typ-cons-ua'**, since this would leave M untyped and prevent us from proving termination. And we cannot just require M to be typed in some context, since M can potentially violate linearity constraints. We therefore introduce a relaxed typing judgment $\Gamma \vdash_i M : A$ and get the typing rules **typ-cons-ul** and **typ-cons-ua**. This relaxed judgment is similar to $\Gamma \vdash M : A$ except that it makes all variables available everywhere disregarding linearity and affineness, i.e. it can be obtained by removing all nolin constraints and by replacing all the relations "$\Gamma = \Gamma_1 \bowtie \Gamma_2$", "$\Gamma \succ_{\text{aff}} \Gamma'$", and "$\Gamma' = \overline{\Gamma}$" by identity relations.

Example 2. Consider again the term $(1 \hat{} 2)[2^{\mathbf{L}}.\mathsf{id}]$ from Example 1. If we follow the Church-style system sketched above, this term reduces to $1[2^{\mathbf{L}}.\mathsf{id}] \hat{} 2[\perp^{\mathbf{U_L}}.\mathsf{id}]$, which in a few more steps reduces to the normal form $2 \hat{} 1$. Note that the two explicit substitutions are syntactically different. In our system, we can however leave the substitution unchanged $1[2^{\mathbf{L}}.\mathsf{id}] \hat{} 2[2^{\mathbf{L}}.\mathsf{id}]$ and perform splitting only on the type level using the rule **typ-cons-ul** defined in Figure 3. The resulting normal form is of course the same.

The fact that this solves the problem and allows us to split type derivations of substitutions without changing the actual syntactic substitution is shown in Lemma 3.

Lemma 1 (`int-typing-lemmas.elf`)

1. *If $\Gamma \vdash M : A$ then $\underline{\Gamma} \vdash_i M : A$.*
2. *If $\Gamma \vdash s : \Gamma'$ then $\underline{\Gamma} \vdash_i s : \underline{\Gamma'}$.*

Since affine assumptions can be weakened away at all leaves of a typing derivation, we get the following weakening lemma:

Lemma 2 (`weakening.elf`)

1. *If $\Gamma_1 \vdash M : A$ and $\Gamma_2 \succ_{\mathrm{aff}} \Gamma_1$ then $\Gamma_2 \vdash M : A$.*
2. *If $\Gamma_1 \vdash s : \Gamma'$ and $\Gamma_2 \succ_{\mathrm{aff}} \Gamma_1$ then $\Gamma_2 \vdash s : \Gamma'$.*
3. *If $\Gamma \vdash s : \Gamma_1'$ and $\Gamma_1' \succ_{\mathrm{aff}} \Gamma_2'$ then $\Gamma \vdash s : \Gamma_2'$.*

4 Reduction Semantics

The reduction rules defining the semantics are given in Figure 4. In order to give a concise presentation of the congruence rules, we use $Y\{Z\}$ to denote an expression Y with a hole in it, where the hole has been replaced by Z. Note that this is completely syntactical and has no α-equivalence problems as we are using de Bruijn indices.

Most reduction systems with explicit substitutions also include reductions such as $1 \,.\, \uparrow \to \mathsf{id}$ and $1[s] \,.\, (\uparrow \circ s) \to s$, and indeed without them (or something similar) the system is not confluent. However, since these reductions essentially are η-reductions on substitutions, it seems that they should really turn the other way, enabling us to η-expand substitutions. We will regain confluence by allowing substitution expansion at meta-variables. This allows us to bound the expansion by the context carried by the meta-variable, since this must match the type of the substitution. The substitution expansion rules are given in Figure 5 with the **eta-sub** rule extending the rules in Figure 4. Notice that the **eta-x-shifts-*** rules are exactly $s \to 1[s] \,.\, (\uparrow \circ s)$ in the case where $s = \uparrow^n$.

Six desirable features of explicit substitution calculi are often referred to as confluence (**C**), meta-confluence (**MC**), preservation of strong normalization (**PSN**), strong normalization (**SN**), simulation of ordinary β-reduction (**SIM**), and full composition (**FC**) [10]. As our calculus has its roots in the calculus $\lambda\sigma$ we do not have strong normalization [11], but for implementations relying on controlled reduction strategies this is not necessarily an issue. However, our confluence result (Theorem 5) actually proves meta-confluence since we have included meta-variables in our calculus.

4.1 Type Preservation

Lemma 3 (`subst-lemmas.elf`). *Substitutions preserve context splits.*

beta-l	$(\widehat{\lambda} M)\hat{\ } N$	$\rightarrow M[N^{\mathbf{L}}.\mathsf{id}]$
beta-a	$(\mathring{\lambda} M)@N$	$\rightarrow M[N^{\mathbf{A}}.\mathsf{id}]$
beta-i	$(\lambda M)\ N$	$\rightarrow M[N^{\mathbf{I}}.\mathsf{id}]$
beta-fst	$\mathsf{fst}\langle M, N\rangle$	$\rightarrow M$
beta-snd	$\mathsf{snd}\langle M, N\rangle$	$\rightarrow N$
clos-var	$1[M^f.s]$	$\rightarrow M$
clos-clos	$M[s][t]$	$\rightarrow M[s \circ t]$
clos-metavar	$X[s][t]$	$\rightarrow X[s \circ t]$
clos-pair	$\langle M, N\rangle[s]$	$\rightarrow \langle M[s], N[s]\rangle$
clos-fst	$(\mathsf{fst}\ M)[s]$	$\rightarrow \mathsf{fst}\ (M[s])$
clos-snd	$(\mathsf{snd}\ M)[s]$	$\rightarrow \mathsf{snd}\ (M[s])$
clos-lam-l	$(\widehat{\lambda} M)[s]$	$\rightarrow \widehat{\lambda}(M[1^{\mathbf{L}}.(s \circ \uparrow)])$
clos-lam-a	$(\mathring{\lambda} M)[s]$	$\rightarrow \mathring{\lambda}(M[1^{\mathbf{A}}.(s \circ \uparrow)])$
clos-lam-i	$(\lambda M)[s]$	$\rightarrow \lambda(M[1^{\mathbf{I}}.(s \circ \uparrow)])$
clos-app-l	$(M\hat{\ }N)[s]$	$\rightarrow M[s]\hat{\ }N[s]$
clos-app-a	$(M@N)[s]$	$\rightarrow M[s]@N[s]$
clos-app-i	$(M\ N)[s]$	$\rightarrow M[s]\ N[s]$
clos-id	$M[\mathsf{id}]$	$\rightarrow M$
comp-id-L	$\mathsf{id} \circ s$	$\rightarrow s$
comp-id-R	$s \circ \mathsf{id}$	$\rightarrow s$
comp-shift	$\uparrow \circ (M^f.s)$	$\rightarrow s$
comp-cons	$(M^f.s) \circ t$	$\rightarrow M[t]^f.(s \circ t)$
comp-comp	$(s_1 \circ s_2) \circ s_3$	$\rightarrow s_1 \circ (s_2 \circ s_3)$

$$\frac{N \rightarrow N'}{M\{N\} \rightarrow M\{N'\}}\ \textbf{cong-tm-tm} \qquad \frac{s \rightarrow s'}{M\{s\} \rightarrow M\{s'\}}\ \textbf{cong-tm-sub}$$

$$\frac{M \rightarrow M'}{s\{M\} \rightarrow s\{M'\}}\ \textbf{cong-sub-tm} \qquad \frac{t \rightarrow t'}{s\{t\} \rightarrow s\{t'\}}\ \textbf{cong-sub-sub}$$

Fig. 4. Reduction rules

1. *If* $\Gamma \vdash s : \Gamma'$ *and* $\Gamma' = \Gamma'_1 \bowtie \Gamma'_2$ *then there exists* Γ_1 *and* Γ_2 *such that* $\Gamma_1 \vdash s : \Gamma'_1$, $\Gamma_2 \vdash s : \Gamma'_2$, *and* $\Gamma = \Gamma_1 \bowtie \Gamma_2$.
2. *If* $\Gamma \vdash s : \Gamma'$ *and* $\Gamma' = \Gamma' \bowtie \Gamma'_2$ *then there exists* Γ_2 *such that* $\Gamma_2 \vdash s : \Gamma'_2$ *and* $\Gamma = \Gamma \bowtie \Gamma_2$.
3. *If* $\Gamma \vdash s : \Gamma'$ *and* $\mathsf{nolin}(\Gamma')$ *then* $\mathsf{nolin}(\Gamma)$.

Notice that the second part of Lemma 3 is equivalent to the statement that $\Gamma \vdash s : \Gamma'$ implies $\overline{\Gamma} \vdash s : \overline{\Gamma'}$. Also it might be tempting to try and prove the second part from the first, but this does not work since $\Gamma_2 \vdash s : \overline{\Gamma'}$ does not imply $\Gamma_2 = \overline{\Gamma_2}$ due to the possible existence of affine assumptions in Γ_2.

With this lemma we can now prove type-preservation:

Theorem 1 (`preservation-thm.elf`). *The reduction relation* $\rightarrow$ *is type-preserving.*

$$\begin{array}{lll}
\textbf{eta-x-shifts-i} & \uparrow^n : \Gamma, A^{\mathbf{I}} & \to_\eta (n+1)^{\mathbf{I}}.\uparrow^{n+1} \\
\textbf{eta-x-shifts-a} & \uparrow^n : \Gamma, A^{\mathbf{A}} & \to_\eta (n+1)^{\mathbf{A}}.\uparrow^{n+1} \\
\textbf{eta-x-shifts-ua} & \uparrow^n : \Gamma, A^{\mathbf{U_A}} & \to_\eta (n+1)^{\mathbf{A}}.\uparrow^{n+1} \\
\textbf{eta-x-shifts-l} & \uparrow^n : \Gamma, A^{\mathbf{L}} & \to_\eta (n+1)^{\mathbf{L}}.\uparrow^{n+1} \\
\textbf{eta-x-shifts-ul} & \uparrow^n : \Gamma, A^{\mathbf{U_L}} & \to_\eta (n+1)^{\mathbf{L}}.\uparrow^{n+1}
\end{array}$$

$$\frac{s : \Gamma_X \to_\eta s'}{X[s] \to X[s']}\,\textbf{eta-sub} \qquad \frac{s : \Gamma \to_\eta s'}{M^{\mathbf{I}}.s : \Gamma, A^{\mathbf{I}} \to_\eta M^{\mathbf{I}}.s'}\,\textbf{eta-x-i}$$

$$\frac{s : \Gamma \to_\eta s'}{M^{\mathbf{A}}.s : \Gamma, A^{\mathbf{A}} \to_\eta M^{\mathbf{A}}.s'}\,\textbf{eta-x-a} \qquad \frac{s : \Gamma \to_\eta s'}{M^{\mathbf{A}}.s : \Gamma, A^{\mathbf{U_A}} \to_\eta M^{\mathbf{A}}.s'}\,\textbf{eta-x-ua}$$

$$\frac{s : \Gamma \to_\eta s'}{M^{\mathbf{L}}.s : \Gamma, A^{\mathbf{L}} \to_\eta M^{\mathbf{L}}.s'}\,\textbf{eta-x-l} \qquad \frac{s : \Gamma \to_\eta s'}{M^{\mathbf{L}}.s : \Gamma, A^{\mathbf{U_L}} \to_\eta M^{\mathbf{L}}.s'}\,\textbf{eta-x-ul}$$

Fig. 5. Substitution expansion

1. *If $\Gamma \vdash M : A$ and $M \to M'$ then $\Gamma \vdash M' : A$.*
2. *If $\Gamma \vdash s : \Gamma'$ and $s \to s'$ then $\Gamma \vdash s' : \Gamma'$.*
3. *If $\Gamma \vdash s : \Gamma'$ and $s : \Gamma' \to_\eta s'$ then $\Gamma \vdash s' : \Gamma'$.*

4.2 σ-Reduction

Before we prove confluence and termination of the entire calculus, we deal with substitutions. This will allow us to reduce the confluence and termination proofs to the case of the ordinary λ-calculus.

Consider the reduction relation without the **beta-*** rules. We will call this sub-relation σ-reduction and denote it $\to_\sigma$. A term or substitution that cannot σ-reduce is said to be in σ-normal form. We write this as the postfix predicate $\not\to_\sigma$.

Theorem 2 (`signf-exists.elf`)**.** *σ-reduction is terminating.*

1. *For all terms M there exists a term M' such that $M \to_\sigma^* M'$ and $M' \not\to_\sigma$.*
2. *For all substitutions s there exists a substitution s' such that $s \to_\sigma^* s'$ and $s' \not\to_\sigma$.*

Furthermore typed σ-reduction is confluent, which is equivalent to having unique normal forms, since it is terminating.[1]

For typed terms M and substitutions s we will denote their unique σ-normal forms $\sigma(M)$ and $\sigma(s)$, respectively.

Theorem 3 (`signf-uniq.elf`)**.** *Typed σ-reduction is confluent.*

1. *If $\Gamma \vdash M : A$, $M \to_\sigma^* M_1$, and $M \to_\sigma^* M_2$ then there exists a term M' such that $M_1 \to_\sigma^* M'$ and $M_2 \to_\sigma^* M'$.*

[1] The reason why we need types is because confluence relies on η-expansion of substitutions.

$$\frac{}{\uparrow^n \sim_\eta \uparrow^n} \qquad \frac{s \sim_\eta s' \quad M \not\to_\sigma}{M^f.s \sim_\eta M^f.s'}$$

$$\frac{s \sim_\eta s'}{s' \sim_\eta s} \qquad \frac{\uparrow^{n+1} \sim_\eta s}{\uparrow^n \sim_\eta (n+1)^{ff}.s}$$

Fig. 6. η-equivalence of substitutions

2. *If $\Gamma \vdash s : \Gamma'$, $s \to_\sigma^* s_1$, and $s \to_\sigma^* s_2$ then there exists a substitution s' such that $s_1 \to_\sigma^* s'$ and $s_2 \to_\sigma^* s'$.*

Having confluence of σ-reduction gives us a lot of nice algebraic properties. One of the most important properties for the formalizations of the proofs is $\sigma(\sigma(M[s])[t]) = \sigma(M[s \circ t])$, which shows that substitution composition indeed behaves as expected. But since we have restricted η-conversions on substitutions we get no immediate corollaries from σ-confluence concerning η-equivalences. To remedy this we will first define η-equivalence on substitutions in σ-normal form and then show that η-equivalent substitutions indeed behave identically. The definition of η-equivalence is given in Figure 6. The following lemma shows that $\sim_\eta$ is an equivalence relation on typed substitutions since symmetry is given by definition.

Lemma 4 (`signf-equiv.elf`). *Reflexivity and transitivity for η-equivalence of substitutions.*

1. *If $s \not\to_\sigma$ then $s \sim_\eta s$.*
2. *If $\Gamma \vdash s_1 : \Gamma'$, $\Gamma \vdash s_2 : \Gamma'$, $\Gamma \vdash s_3 : \Gamma'$, $s_1 \sim_\eta s_2$, and $s_2 \sim_\eta s_3$ then $s_1 \sim_\eta s_3$.*

Lemma 5 (`signf-equiv.elf`). *η-equivalence of substitutions contains η-expansion.*
If $s : \Gamma \to_\eta s'$ and $s \not\to_\sigma$ then $s \sim_\eta s'$.

Theorem 4 (`signf-equiv.elf`). *Substitutions that are η-equivalent behave the same way with respect to σ-reduction.*
If $\Gamma' \vdash M : A$, $\Gamma' \vdash t : \Gamma''$, $\Gamma \vdash s : \Gamma'$, $\Gamma \vdash s' : \Gamma'$, and $s \sim_\eta s'$ then:

1. $\sigma(M[s]) = \sigma(M[s'])$
2. $\sigma(t \circ s) \sim_\eta \sigma(t \circ s')$

4.3 Confluence and Termination

Now we have the tools necessary to prove confluence of the entire reduction relation.

Lemma 6 (Generalized Interpretation Method). *Given two sets $B \subseteq A$, reduction relations R_1 and R_2 on A and B, respectively, and a function $f : A \to B$ such that:*

$$\begin{tikzcd}
 & M \arrow[dl, "R_1^*"'] \arrow[d, "R_1^*"] \arrow[dr, "R_1^*"] & \\
M_1 \arrow[d, "R_1^*"'] & f(M) \arrow[dl, "R_2^*"'] \arrow[dr, "R_2^*"] & M_2 \arrow[d, "R_1^*"] \\
f(M_1) \arrow[dr, "R_2^*"] & & f(M_2) \arrow[dl, "R_2^*"'] \\
 & M' &
\end{tikzcd}
\qquad
\begin{tikzcd}
 & M \arrow[dl] \arrow[d] \arrow[dr] & \\
M_1 \arrow[d] & \sigma(M) \arrow[dl, "\beta^*"'] \arrow[dr, "\beta^*"] & M_2 \arrow[d] \\
\sigma(M_1) \arrow[dr, "\beta^*"] & & \sigma(M_2) \arrow[dl, "\beta^*"'] \\
 & M' &
\end{tikzcd}$$

Fig. 7. Confluence by the interpretation method

1. *$R_2 \subseteq R_1^*$.*
2. *$\forall M \in A:\ M \rightarrow_{R_1}^* f(M)$.*
3. *$\forall M, M' \in A:\ M \rightarrow_{R_1}^* M' \Rightarrow f(M) \rightarrow_{R_2}^* f(M')$.*

Then confluence of R_2 implies confluence of R_1.

Proof. The proof is by simple diagram chasing as shown in Figure 7 on the left.

To prove confluence of the entire reduction relation we use Lemma 6 with σ-normalization as the interpreting function f. As R_2 we take ordinary β-reduction on σ-normal forms, which in our case can be defined as one of the **beta-*** rules followed by σ-normalization. The situation is illustrated in Figure 7 on the right. First everything is σ-normalized, then σ-normalization is shown to preserve β-reduction, and finally confluence of the usual λ-calculus is used to conclude confluence of our calculus.

Theorem 5 (`confluence.elf`). *The reduction relation $\rightarrow$ is confluent on typed terms and substitutions.*

1. *If $\Gamma \vdash M : A$, $M \rightarrow^* M_1$, and $M \rightarrow^* M_2$ then there exists a term M' such that $M_1 \rightarrow^* M'$ and $M_2 \rightarrow^* M'$.*
2. *If $\Gamma \vdash s : \Gamma'$, $s \rightarrow^* s_1$, and $s \rightarrow^* s_2$ then there exists a substitution s' such that $s_1 \rightarrow^* s'$ and $s_2 \rightarrow^* s'$.*

Similarly we can also prove termination of $\rightarrow$ from termination of the usual λ-calculus by σ-normalization.

Theorem 6 (`nf-exists.elf`). *The reduction relation $\rightarrow$ is terminating on typed terms and substitutions.*

1. *If $\Gamma \vdash M : A$ then there exists a term M' such that $M \rightarrow M'$ and $M' \not\rightarrow$.*
2. *If $\Gamma \vdash s : \Gamma'$ then there exists a substitution s' such that $s \rightarrow s'$ and $s' \not\rightarrow$.*

The normal forms for the reduction relation $\rightarrow$ are called β-normal forms and because of Theorems 5 and 6 we know that they exist and are unique.

5 Dependent Types

The presented calculus can easily be extended to dependent types, since this extension is orthogonal to linear and affine types. We will sketch the extension here.

The base types a and intuitionistic function types $A \to B$ are replaced by type families $a\ M_1 \ldots M_n$ and dependent functions $\Pi A. B$. The type system is extended with a judgment $\Gamma \vdash A : \mathsf{type}$ specifying that the type A is well-formed in the context Γ and a well-formedness requirement on contexts stating that whenever we form the context Γ, A^l we must have $\overline{\Gamma} \vdash A : \mathsf{type}$. The system is then tied together by the central invariant stating that whenever $\Gamma \vdash M : A$ then $\overline{\Gamma} \vdash A : \mathsf{type}$.

The typing rules of the system become slightly more complicated to account for the objects occurring in the types. As an example of one of the updated rules, consider **typ-cons-l** which becomes:

$$\frac{\Gamma = \Gamma_1 \bowtie \Gamma_2 \quad \Gamma_1 \vdash M : A[s] \quad \Gamma_2 \vdash s : \Gamma'}{\Gamma \vdash M^{\mathbf{L}}.s : \Gamma', A^{\mathbf{L}}}\ \textbf{deptyp-cons-l}$$

Notice that $\overline{\Gamma_1} = \overline{\Gamma_2}$ and that Lemma 3 gives us $\overline{\Gamma_2} \vdash s : \overline{\Gamma'}$ to ensure that $A[s]$ is well-formed in the right context.

Theorems 5 and 6 extend to the dependently typed setting as well, since the proofs can be reused after type erasure.

6 Conclusion

Implementing substructural logics is a complex undertaking, because previous formulations of explicit substitution calculi have not been very amenable to implementation. We believe that the same implementation difficulties persist even if one chooses nominal techniques.

In this paper we present a calculus of explicit substitutions for the linear and affine λ-calculus including meta-variables. We have defined a Curry-centric type system and a small-step reduction semantics that are directly implementable without the need to split substitutions syntactically. We have proved subject reduction, confluence, and termination for this calculus and mechanically checked all lemmas and theorems. The calculus serves as the foundation of the implementation of the logical framework Celf — the Celf source code can be downloaded from `http://www.twelf.org/~celf`

We believe this to be an important contribution to the implementation of any system based on substructural λ-calculi.

References

1. Abadi, M., Cardelli, L., Curien, P.-L., Lévy, J.-J.: Explicit substitutions. Journal of Functional Programming 1(4), 375–416 (1991)
2. Baelde, D., Gacek, A., Miller, D., Nadathur, G., Tiu, A.: The Bedwyr system for model checking over syntactic expressions. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 391–397. Springer, Heidelberg (2007)

3. Barras, B.: Programming and computing in hol. In: Aagaard, M.D., Harrison, J. (eds.) TPHOLs 2000. LNCS, vol. 1869, pp. 17–37. Springer, Heidelberg (2000)
4. Bierman, G.: On Intuitionistic Linear Logic. PhD thesis, University of Cambridge (1994)
5. Cervesato, I., de Paiva, V., Ritter, E.: Explicit Substitutions for Linear Logical Frameworks: Preliminary Results. In: Felty, A. (ed.) Workshop on Logical Frameworks and Meta-languages — LFM'99, Paris, France, September 28 (1999)
6. Chaudhuri, A., Naldurg, P., Rajamani, S.: A type system for data-flow integrity on Windows Vista. SIGPLAN Notices 43(12), 9–20 (2008)
7. Dowek, G., Hardin, T., Kirchner, C., Pfenning, F.: Unification via explicit substitutions: The case of higher-order patterns. Rapport de Recherche 3591, INRIA, Preliminary version appeared at JICSLP'96 (December 1998)
8. Gacek, A.: The Abella interactive theorem prover (system description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 154–161. Springer, Heidelberg (2008)
9. Ghani, N., de Paiva, V., Ritter, E.: Linear explicit substitutions. Logic Journal of IGPL 8(1), 7 (2000)
10. Kesner, D.: The theory of calculi with explicit substitutions revisited. In: Duparc, J., Henzinger, T.A. (eds.) CSL 2007. LNCS, vol. 4646, pp. 238–252. Springer, Heidelberg (2007)
11. Mellies, P.-A.: Typed λ-calculi with explicit substitutions may not terminate. Typed Lambda Calculi and Applications, 328–334 (1995)
12. Nadathur, G., Mitchell, D.J.: System description: Teyjus - a compiler and abstract machine based implementation of lambda-Prolog. In: Ganzinger, H. (ed.) CADE 1999. LNCS (LNAI), vol. 1632, pp. 287–291. Springer, Heidelberg (1999)
13. Nadathur, G., Wilson, D.S.: A notation for lambda terms. a generalization of environment. Theoretical Computer Science 198(1-2), 49–98 (1998)
14. Pfenning, F., Schürmann, C.: System description: Twelf — a meta-logical framework for deductive systems. In: Ganzinger, H. (ed.) CADE 1999. LNCS (LNAI), vol. 1632, pp. 202–206. Springer, Heidelberg (1999)
15. Poswolksy, A., Schürmann, C.: Practical programming with higher-order encodings and dependent types. In: Drossopoulou, S. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 93–107. Springer, Heidelberg (2008)
16. Schack-Nielsen, A., Schürmann, C.: System description: Celf - a logical framework for deductive and concurrent systems. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 320–331. Springer, Heidelberg (2008)
17. Schack-Nielsen, A., Schürmann, C.: Pattern unification for the lambda calculus with linear and affine types. Under consideration for publication (2010)
18. Shao, Z., League, C., Monnier, S.: Implementing typed intermediate languages. In: ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming, pp. 313–323. ACM, New York (1998)
19. Shinwell, M.R., Pitts, A.M., Gabbay, M.J.: FreshML: Programmming with binders made simple. In: Eighth ACM SIGPLAN International Conference on Functional Programming (ICFP 2003), Uppsala, Sweden, pp. 263–274. ACM Press, New York (2003)

Beluga: A Framework for Programming and Reasoning with Deductive Systems (System Description)

Brigitte Pientka and Jana Dunfield

McGill University, Montréal, Canada

{bpientka@cs.mcgill.ca}

Abstract. Beluga is an environment for programming and reasoning about formal systems given by axioms and inference rules. It implements the logical framework LF for specifying and prototyping formal systems via higher-order abstract syntax. It also supports reasoning: the user implements inductive proofs about formal systems as dependently typed recursive functions. A distinctive feature of Beluga is that it not only represents binders using higher-order abstract syntax, but directly supports reasoning with contexts. Contextual objects represent hypothetical and parametric derivations, leading to compact and elegant proofs. Our test suite includes standard examples such as the Church-Rosser theorem, type uniqueness, proofs about compiler transformations, and preservation and progress for various ML-like languages. We also implemented proofs of structural properties of expressions and paths in expressions. Stating these properties requires nesting of quantifiers and implications, demonstrating the expressive power of Beluga.

1 Introduction

Beluga is an environment for programming with and reasoning about deductive systems. It uses a two-level approach. The data level implements the logical framework LF [3], which has been successfully used to define logics and represent derivations and proofs. Its strength and elegance comes from supporting encodings based on higher-order abstract syntax (HOAS), in which binders in the object language are represented as binders in LF's meta-language.

On top of LF, we provide a computation level that supports analyzing and manipulating LF data via pattern matching. A distinctive feature of Beluga is explicit support for contexts and contextual objects, which concisely characterize hypothetical and parametric derivations (proof objects). These contextual objects are analyzed and manipulated naturally by pattern matching.

The Beluga system is unique in having context variables, allowing generic functions that abstract over contexts. As types classify terms, context schemas classify contexts. Contexts whose schemas are superficially incompatible can be reasoned with via context weakening and context subsumption.

The main application of Beluga is to prototype deductive systems together with their meta-theory. Deductive systems given via axioms and inference rules are common in the design and implementation of programming languages, type systems, authorization and security logics, and so on. In Beluga, inductive proofs about deductive systems are directly implemented as recursive functions that case-analyze some given (possibly hypothetical) derivations. At the same time, Beluga serves as an experimental framework for programming with proof objects, useful for certified programming and proof-carrying code [6].

Beluga is implemented in OCaml. It provides a completely new implementation of LF [3] together with type reconstruction, constraint-based higher-order unification and type checking. In addition, it provides a computation language that supports writing dependently-typed recursive functions over contextual objects and explicit contexts. Building on our earlier work [10,11], we designed a palatable source language. To achieve a practical system, we implemented bidirectional type reconstruction for dependently typed functions.

We tested our implementation of LF type reconstruction on many examples from the Twelf repository [7] and found its performance competitive. We also implemented a broad range of proofs as recursive Beluga functions, including proofs of the Church-Rosser theorem, proofs about compiler transformations, subject reduction, and translation from natural deduction to Hilbert style. To illustrate the expressive power of Beluga, our test suite includes simple theorems about structural relationships between expressions and proofs about the paths in expressions. These latter theorems have nested quantifiers and implications, placing them outside the fragment of propositions expressible in systems such as Twelf. Type reconstruction of these proofs takes less than a second. Finally, Beluga provides an interpreter, based on a lazy environment-based semantics, to execute computation-level programs.

The Beluga system, including source code, examples, and an Emacs mode, is available from `http://complogic.cs.mcgill.ca/beluga/`.

To provide an intuition for what Beluga accomplishes and how it is used, we first present a type uniqueness proof in Beluga. Section 3 compares Beluga to systems with similar applications. Section 4 discusses Beluga's implementation.

2 Example: Type Uniqueness

To illustrate the core ideas behind Beluga, we implement a proof of type uniqueness for the simply-typed lambda-calculus (STLC). First, we briefly review how to represent the STLC and its typing rules in LF.

```
tp: type .
nat: tp.
arr: tp → tp → tp.
```

```
exp: type .
lam : tp → (exp→exp) → exp.
app : exp → exp → exp.
```

```
oft: exp → tp → type .

t_app: oft E1 (arr T2 T) → oft E2 T2
       → oft (app E1 E2) T.

t_lam: ({x:exp} oft x T1 → oft (E x) T2)
        → oft (lam T1 E) (arr T1 T2).
```

```
equal: tp → tp → type .
e_ref: equal T T.
```

The first part states that `nat` is a data-level type `tp` and that `arr` takes two arguments of type `tp` and constructs a `tp`. To represent λ-term constructors, we use higher-order abstract syntax: The constructor `lam` takes a `tp` and the body of the abstraction, of type `(exp → exp)`. For example, lam $x{:}nat\,.\,x$ is represented by `lam nat λx.x`. In the second part, we represent the typing judgment $M : T$ in our object language by an LF type `oft`, and the typing rules are represented by LF type constants `t_app` and `t_lam`.

The rule `t_app` encodes the typing rule for applications: from derivations of `oft E1 (arr T2 T)` and `oft E2 T2` we get `oft (app E1 E2) T`. The rule `t_lam` uses a parametric hypothetical derivation "for all `x` assuming `oft x T1` we can derive `oft (E x) T2`", represented as a function type `{x:exp} oft x T → oft (E x) T2`. Finally, we define the equality judgment, which simply encodes reflexivity.

The above is standard in LF. We now state type uniqueness:

Theorem. *If* $\Gamma \vdash$ `oft E T` *and* $\Gamma \vdash$ `oft E T'` *then* `equal T T'`*.*

This statement makes explicit the context Γ containing variable typing assumptions. Note that while terms `E` can depend on variables declared in Γ, no variables can occur in the types `T` and `T'`, though this is not captured by the statement above.

The theorem corresponds to a type of a recursive function in Beluga. Before showing how to implement it, we describe more precisely the shape of contexts Γ, using a context schema declaration:

```
schema tctx = some [t:tp] block x:exp. oft x t;
```

The schema `tctx` describes a context containing assumptions `x:exp`, each associated with a typing assumption `oft x t` for some type `t`. Formally, we are using a dependent product Σ (used only in contexts) to tie `x` to `oft x t`. We thus do not need to establish separately that for every variable there is a unique typing assumption: this is inherent in the definition of `tctx`.

We can now state the Beluga type corresponding to the statement above:

```
{g:tctx} (oft (E ..) T)[g] → (oft (E ..) T')[g] → (equal T T')[ ]
```

Read it as follows: for all contexts `g` of schema `tctx`, given derivations of `(oft (E ..) T)[g]` and of `(oft (E ..) T')[g]` we can construct a derivation of `(equal T T')[ ]`. The `[ ]` means the result is closed. As we remarked, only the term `E` can contain variables; the type `T` is closed. Although we did not state this dependency in the on-paper statement, Beluga distinguishes closed objects from objects depending on assumptions. To describe the dependency of `E` on the context `g`, we write `(E ..)` associating `..` with the variable `E`. (Technically, `..` is an identity substitution mapping variables from `g` to themselves.) In contrast, `T` by itself denotes a closed `tp` that cannot depend on hypotheses in `g`.

The proof of type uniqueness is by case analysis on the first derivation. Accordingly, the recursive function in Figure 1 pattern-matches on the first derivation `d`, of type `(oft (E ..) T)[g]`. The first two cases correspond to `d` concluding with `t_app` or `t_lam`. The third case corresponds to when `d` derives `(oft (E ..) T)[g]` by using a declaration from the context `g`. If the context were concrete, we could

```
rec  unique : {g:tctx} (oft (E ..) T)[g]
                       → (oft (E ..) T')[g]
                       → (equal T T')[ ]
= fn  d ⇒ fn  f ⇒ case  d of
| [g] t_app (D1 ..) (D2 ..) ⇒                           % Application case
  let  [g] t_app (F1 ..) (F2 ..) = f in
  let  [ ] e_ref = unique ([g] D1 ..) ([g] F1 ..) in
    [ ] e_ref

| [g] t_lam (\x.\u. D .. x u) ⇒                         % Abstraction case
  let  [g] t_lam (\x.\u. F .. x u) = f in
  let  [ ] e_ref = unique ([g,b:block x:exp.oft x _ ] D .. b.1 b.2)
                          ([g,b] F .. b.1 b.2) in
   [ ] e_ref

| [g] #q.2 .. ⇒               % d : oft #q.1  T          % Assumption case
  let  [g] #r.2 .. = f   in   % f  : oft #q.1  T'
    [ ] e_ref ;
```

Fig. 1. Implementation of type uniqueness in Beluga

simply refer to the concrete variable names listed, but our function is generic for any context g. So we use a *parameter variable* #q that stands for *some* declaration in g.

The first (application) case is essentially a simpler version of the second (abstraction) case, so we omit the application case.

Abstraction case: If the first derivation d concludes with t_lam, it matches the pattern [g] t_lam (λx. λu. D .. x u), and is a contextual object in the context g of type oft (lam T1 (λx. E0 .. x)) (arr T1 T2). Thus, E .. = lam T1 (λx. E0 .. x) and T = arr T1 T2. Pattern matching—through a let-binding—serves to invert the second derivation f, which must have been by t_lam with a subderivation F1 .. x u deriving oft (E0 .. x) T2' that can use x, u:oft x T1, and assumptions from g. Hence, after pattern matching on d and f, we know that E = lam T1 (λx. E0 .. x) and T = arr T1 T2 and T' = arr T1 T2'.

The use of the induction hypothesis on D and F in a paper proof corresponds to the recursive call to unique. To appeal to the induction hypothesis, we need to extend the context by pairing up x and its typing assumption: g, b:block x:exp. oft x T1. In the code, we wrote an underscore _ instead of T1, which tells Beluga to reconstruct it. (We cannot write T1 there without binding it by explicitly giving the type of D, so it is much easier to write _.) To retrieve x we take the first projection b.1, and to retrieve x's typing assumption we take the second projection b.2. Note that while unique's type says it takes a context variable {g:tctx}, we do not pass it explicitly; Beluga infers it from the context g, b:block x:exp. oft x _ in the first argument passed.

Now we can appeal to the induction hypothesis using D1 .. b.1 b.2 and F1 .. b.1 b.2 in the context g,b:block x:exp. oft x T1. We pass three arguments: the context g and two contextual objects. From the i.h. we get a contextual object, a closed derivation of (equal (arr T1 T2) (arr T1 T2'))[ ]. The only rule that could derive this is e_ref, and pattern matching establishes that

`T2` must equal `T2'`, and hence `arr T1 T2` = `arr T1 T2'`, i.e. `T = T'`. Hence, there is a proof of `[] equal T T'`, and we can finish with the reflexivity rule `e_ref`.

Assumption case: Here, we must have used an assumption from the context `g` to construct the derivation `d`. Parameter variables `#q` allow a generic case that matches a declaration `block x:exp.oft x S` for any `S` in `g`. Since our pattern match proceeds on typing derivations, we want the second component, written `#q.2`. The pattern match on `d` also establishes that `E = #q.1` and `S = T`. Next, we pattern match on `f`, which has type `oft (#q.1 ..) T'` in the context `g`. Clearly, the only possible way to derive `f` is by using an assumption from `g`. We call this assumption `#r`, standing for a declaration `block y:exp. oft y S'`, so `#r.2` refers to the second component `oft (#r.1 ..) S'`. Pattern matching between `#r.2` and `f` also establishes that both types are equal and that `S' = T'` and `#r.1 = #q.1`. Finally, we observe that `#r.1 = #q.1` only if `#r` is equal to `#q`. Consequently, both parameters have equal types, and `S = S' = T = T'`. (In general, unification in the presence of Σ-types does not yield a unique unifier, but in Beluga only parameter variables and variables from the context can be of Σ type, yielding a unique solution.)

3 Related Work

There are many approaches for specifying and reasoning about formal systems. Our work builds on the experience with Twelf [7], which provides a meta-language for specifying, implementing and reasoning about formal systems using higher-order abstract syntax. However, proofs in Twelf are relations; one needs to prove separately that the relation constitutes a total function and Twelf supports both termination and coverage checking.

A second key difference is that Twelf does not explicitly support contexts and contextual data; contexts (worlds) in Twelf are implicit. Consequently, it is not possible to distinguish between different contexts within the same statement and base cases are scattered.

The third key difference is its expressiveness. In Twelf, we can only encode forall-exists statements, while Beluga directly handles a larger class of statements. An example is the statement that if, for all paths P through a lambda-term M, we know that P also characterizes all paths through another term N, then M and N must be equal.

Delphin [12] is closest to Beluga. Its implementation uses much of the Twelf infrastructure, but proofs are implemented as functions (like Beluga) rather than relations. As in Twelf, contexts are implicit with similar consequences. To have more fine-grained control over assumptions which we typically track in a context, Delphin users can use a continuation-based approach where the continuation plays the role of a context and must be explicitly managed by the programmer.

Abella [2] is an interactive theorem prover for reasoning about specifications of formal systems. Its theoretical basis is different, but it supports encodings based on higher-order abstract syntax. However, contexts are not first-class and must be managed explicitly. For example, type uniqueness requires a lemma that each variable has a unique typing assumption, which comes for free in Beluga.

Finally, the Hybrid system [5] tries to exploit the advantages of HOAS within the well-understood setting of higher-order logic as implemented by systems such as Isabelle and Coq. Hybrid provides a definitional layer where higher-order abstract syntax representations are compiled to de Bruijn representations, with tools for reasoning about them using tactical theorem proving and principles of (co)induction. This is a flexible approach, but contexts must be defined explicitly and properties about them must be established separately.

4 Implementation

Beluga is implemented in OCaml. It provides a complete reimplementation of the logical framework LF. Similarly to the Twelf core, Beluga supports type reconstruction for LF signatures based on higher-order pattern unification with constraints. In addition, we designed and implemented a type reconstruction algorithm for dependently-typed functions on contextual data.

Type reconstruction is, in general, undecidable for the data level (that is, LF) and for the computation level. For LF, our algorithm reports a principal type, a type error, or that the source term needs more type information. For our computation language, we check functions against a given type and either succeed, report a type error, or fail by asking for more type information. It is always possible to make typing unambiguous by adding more annotations.

An efficient implementation of higher-order unification is crucial to this. For higher-order patterns [4], we implemented a unification algorithm [8] and, similarly to Twelf, extended it with constraints. We also extended the algorithm to handle parameter variables and Σ-types for variables.

Beluga also supports context subsumption, so one can provide a contextual object in a context Ψ in place of a contextual object in some other context Φ, provided Ψ can be obtained by weakening Φ. This mechanism, similar to world subsumption in Twelf, is crucial when assembling larger proofs.

Finally, Beluga includes an interpreter with a lazy environment-based operational semantics. This allows us to execute Beluga programs, producing concrete derivations and other LF data.

In the future, we plan to address two significant issues.

Totality. Type-checking guarantees local consistency and partial correctness, but does not guarantee that functions are total. Thus, while we can implement, partially verify, and execute functions about derivations in deductive systems, Beluga does not currently guarantee the validity of a meta-proof. The two missing pieces are coverage and termination. We formulated an algorithm [1] to ensure that all cases are covered, and plan to implement it over the next few months. Verifying termination will follow ideas in Twelf [13,9] for checking that arguments in recursive calls are indeed smaller.

Automation. Currently, the recursive functions that implement induction proofs must be written by hand. We plan to explore how to enable the user to interactively develop functions in collaboration with theorem provers that can fill in parts of functions (that is, proofs) automatically.

References

1. Dunfield, J., Pientka, B.: Case analysis of higher-order data. In: International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'08), June 2009. Electronic Notes in Theoretical Computer Science (ENTCS), vol. 228, pp. 69–84. Elsevier, Amsterdam (2009)
2. Gacek, A.: The Abella interactive theorem prover (system description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 154–161. Springer, Heidelberg (2008)
3. Harper, R., Honsell, F., Plotkin, G.: A framework for defining logics. Journal of the ACM 40(1), 143–184 (1993)
4. Miller, D.: A logic programming language with lambda-abstraction, function variables, and simple unification. Journal of Logic and Computation 1(4), 497–536 (1991)
5. Momigliano, A., Martin, A.J., Felty, A.P.: Two-Level Hybrid: A system for reasoning using higher-order abstract syntax. In: International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'07). Electronic Notes in Theoretical Computer Science (ENTCS), vol. 196, pp. 85–93. Elsevier, Amsterdam (2008)
6. Necula, G.C.: Proof-carrying code. In: 24th Annual Symposium on Principles of Programming Languages (POPL'97), January 1997, pp. 106–119. ACM Press, New York (1997)
7. Pfenning, F., Schürmann, C.: System description: Twelf — a meta-logical framework for deductive systems. In: Ganzinger, H. (ed.) CADE 1999. LNCS (LNAI), vol. 1632, pp. 202–206. Springer, Heidelberg (1999)
8. Pientka, B.: Tabled higher-order logic programming. PhD thesis, Department of Computer Science, Carnegie Mellon University, CMU-CS-03-185 (2003)
9. Pientka, B.: Verifying termination and reduction properties about higher-order logic programs. Journal of Automated Reasoning 34(2), 179–207 (2005)
10. Pientka, B.: A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In: 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08), pp. 371–382. ACM Press, New York (2008)
11. Pientka, B., Dunfield, J.: Programming with proofs and explicit contexts. In: ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'08), July 2008, pp. 163–173. ACM Press, New York (2008)
12. Poswolsky, A., Schürmann, C.: System description: Delphin—a functional programming language for deductive systems. In: International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'08), June 2009. Electronic Notes in Theoretical Computer Science (ENTCS), vol. 228, pp. 135–141. Elsevier, Amsterdam (2009)
13. Rohwedder, E., Pfenning, F.: Mode and termination checking for higher-order logic programs. In: Nielson, H.R. (ed.) ESOP 1996. LNCS, vol. 1058, pp. 296–310. Springer, Heidelberg (1996)

MCMT: A Model Checker Modulo Theories

Silvio Ghilardi[1] and Silvio Ranise[2]

[1] Dipartimento di Informatica, Università degli Studi di Milano, Italia
[2] FBK-Irst, Trento, Italia

Abstract. We describe MCMT, a fully declarative and deductive symbolic model checker for safety properties of infinite state systems whose state variables are arrays. Theories specify the properties of the indexes and the elements of the arrays. Sets of states and transitions of a system are described by quantified first-order formulae. The core of the system is a backward reachability procedure which symbolically computes pre-images of the set of unsafe states and checks for safety and fix-points by solving Satisfiability Modulo Theories (SMT) problems. Besides standard SMT techniques, efficient heuristics for quantifier instantiation, specifically tailored to model checking, are at the very heart of the system. MCMT has been successfully applied to the verification of imperative programs, parametrised, timed, and distributed systems.

1 Introduction

In [6], we have presented a fully declarative approach to verify safety properties of infinite state systems—whose variables are arrays—by backward reachability. Such systems can be used as suitable abstractions of many classes of systems ranging from parametrised protocols to sequential programs manipulating arrays. The idea is to use classes of quantified first-order formulae to represent an infinite set of states of the system so that the computation of pre-images boils down to symbolic manipulations. Using suitable theories over the elements and the indexes of the arrays, we are able to declaratively specify both the data manipulated by the system and its topology (in the case of parametrised systems) or properties of the indexes of arrays (in the case of imperative programs).

In the framework of [6], the key to mechanize backward reachability is to reduce the checks for fixed-point and safety to Satisfiability Modulo Theories (SMT) problems of first-order formulae containing (universal) quantifiers. Under suitable hypotheses on the theories over the indexes and the elements of the arrays, these SMT problems are decidable [6] by integrating a quantifier instantiation procedure with SMT solving techniques for quantifier-free formulae. In [8,9], we described heuristics to reduce the number of quantified variables and—most importantly—of instances while preserving the completeness of SMT solving. Unfortunately, the decidability of safety and fixed-point checks is not yet enough to ensure the termination of the backward reachability analysis. Theoretically, termination for this procedure can be ensured for well-structured systems [1]; in [6], we explained how to recast this notion in our framework so as to import all the decidability results available in the literature. Pragmatically, it is well-known that termination of backward reachability can be obtained by using invariants and we discussed how natural it is to guess and use them in our framework and also gave a characterization of the completeness of the proposed method [7].

In this paper, we give the first comprehensive high-level description of MCMT v. 1.0, a significant extension of the prototype tool used in our previous work [7,8]. MCMT v. 1.0 uses Yices (`http://yices.csl.sri.com`) as the back-end SMT solver and is available at `http://www.dsi.unimi.it/~ghilardi/mcmt`. Besides various ameliorations and refinements to previously available functionalities as well as new utilities (like Bounded Model-Checking), MCMT v. 1.0 supports the following new features, which widen its scope of applicability and greatly improve its performances (in particular, w.r.t. termination) when used with care: (i) transitions with existentially quantified variables ranging over data values (and not only indexes), provided that the theory over data admits elimination of quantifiers; (ii) synthesis of invariants with two universally quantified variables (previously [7], they were limited to containing just one variable); (iii) a form of predicate abstraction, called *signature abstraction*, together with limited support for the acceleration of transitions [4]. For lack of space, only an excerpt of the experiments are described here (for full details, consult the MCMT web-page).

2 The MCMT Way of Life

We present our vision of model checking infinite state systems underlying MCMT. To this end, we believe it is convenient to recall two distinct and complementary approaches among the many possible alternatives available in the literature.

The first approach is pioneered in [1] and its main notion is that of well-structured system. Recently, it was implemented in two systems [2,3], which were able to automatically verify several protocols for mutual exclusion and cache coherence. One of the key ingredients to the success of these tools is their capability to perform accurate fixed-point checks so as to reduce the number of iterations of the backward search procedure. A fixed-point check is implemented by 'embedding' an old configuration (i.e. a finite representation of a potentially infinite set of states) into a newly computed pre-image; if this is the case, then the new pre-image is considered "redundant" (i.e., not contributing new information about the set of backward reachable states) and thus can be discarded without loss of precision. Indeed, the exhaustive enumeration of embeddings has a high computational cost. Furthermore, constraints are only used to represent the data manipulated by the system while its topology is encoded by *ad hoc* data structures. A change in the topology of the system requires the implementation from scratch of algorithms for both pre-image and embedding computation. On the contrary, MCMT uses particular classes of *first-order formulae* to represent configurations parametrised with respect to two theories, one for data and one for the topology so that pre-image computation reduces to a fixed set of logical manipulations and fixed-point checking to solve SMT problems containing universally quantified variables. To mechanize these tests, a quantifier-instantiation procedure is used, which is the logical counterpart of the enumeration of embeddings.

Interestingly, this notion of embedding can be recaptured (via classical model theory) [6] in the logical framework underlying MCMT, a fact that allows us to import the decidability results of [1] for backward reachability. Another important advantage of the approach underlying MCMT over that proposed in [1] is its broader scope of applications with respect to the implementations in [2,3]. The use of theories for specifying the data and the topology allows one to model disparate classes of systems in a natural way. Furthermore, even if the quantifier instantiation procedure becomes incomplete with rich theories, it can soundly be used and may still permit the proof of the safety of a system. In fact, MCMT has been successfully employed to verify sequential programs (such as sorting algorithms) that are far beyond the reach of the systems described in [2,3].

The second and complementary approach to model checking infinite state systems relies on *predicate abstraction* techniques, initially proposed in [10]. The idea is to abstract the system to one with finite states, to perform finite-state model checking, and to refine spurious traces (if any) by using decision procedures or SMT solvers. This technique has been implemented in several tools and is often combined with interpolation algorithms for the refinement phase. As pointed out in [5,11], predicate abstraction must be carefully adapted when (universal) quantification is used to specify the transitions of the system or its properties, as it is the case for the problems tackled by MCMT. There are two crucial problems to be solved. The first is to find an appropriate set of predicates to compute the abstraction of the system. In fact, besides system variables, universally quantified variables may also occur in the system. The second problem is that the computation of the abstraction as well as its refinement reduce to solving proof obligations containing universal quantifiers. Hence, we need to perform suitable quantifier instantiations in order to enable the use of decision procedures or SMT solving techniques for quantifier-free formulae. The first problem is solved by Skolemization [5] or fixing the number of variables in the system [11] so that standard predicate abstraction techniques can still be used. The second problem is solved by adopting very straightforward (sometimes naive) and incomplete quantifier instantiation procedures. While being computationally cheap and easy to implement, the heuristics used for quantifier instantiation are largely imprecise and do not permit the detection of redundancies due to variable permutations, internal symmetries, and so on. Experiments performed with MCMT, tuned to mimic these simple instantiation strategies, show much poorer performance. We believe that the reasons of success of the predicate abstraction techniques in [5,11] lie in the clever heuristics used to find and refine the set of predicates for the abstraction. The current implementation of MCMT is orthogonal to the predicate abstraction approach; it features an extensive quantifier instantiation (which is complete for some theories over the indexes and is enhanced with completeness preserving heuristics to avoid useless instances), but it performs only a primitive form of predicate abstraction, called signature abstraction (see Section 4). Another big difference is how abstraction is used in MCMT: the set of backward reachable states is always computed precisely while abstraction is only exploited for guessing candidate invariants which are then used to prune the set of backward reachable states. Since we represent sets of states by formulae, guessing and then using the synthesized invariants turns out to be extremely easy, thereby helping to solve the tension between model checking and deductive techniques that has been discussed a lot in the literature and is still problematic in the tools described in [2,3] where sets of states are represented by *ad hoc* data structures. We plan to enhance predicate abstraction techniques in future releases of MCMT, so as to find the best trade-off between the advantages of predicate abstraction and extensive quantifier instantiation.

3 The Input Language for Safety Problems

The input language of MCMT can be seen as a parametrised extension of the one used by UCLID (`http://www.cs.cmu.edu/~uclid`). Formally, it is a sub-set of multi-sorted first-order logic, extended with the ternary expression constructor "if-then-else" (which is standard in the SMT-LIB format). For lack of space, we omit the presentation of the concrete syntax which is fully described in the on-line User Manual.

Sorts. We use the following distinguished sorts: Ind for indexes, $Elem_1, ..., Elem_m$ for elements of arrays, and $Arr_1, ..., Arr_m$ for array variables (where Arr_k corresponds to arrays of elements of sort $Elem_k$, for $k = 1, ..., m$).

Theories. We assume that the mono-sorted theories T_I and T_{E_k} are given over the sorts Ind and $Elem_k$, respectively, for $k = 1, ..., m$. The three-sorted theories $A_I^{E_k}$ are obtained as the combination of the theories T_I and T_{E_k} for each $k = 1, ..., m$ by adding the sort Arr_k to Ind and $Elem_k$, by taking the union of the symbols of T_I and T_{E_k}, and by adding the binary symbol $\_[\_]_k : Arr_k \times Ind \to Elem_k$ for reading the content of an array at a given index (the subscript k is omitted if clear from the context). Finally, we let $A_I^E := \bigcup_{k=1}^m A_I^{E_k}$.

Formats of formulae. We use two classes of formulae to describe sets of states: $\forall \underline{i}.\phi(\underline{i}, \underline{a})$ and $\exists \underline{i}.\phi(\underline{i}, \underline{a})$, where $\underline{i}$ is a tuple of variables of sort Ind, $\underline{a}$ is a tuple of length m of array variables of sorts $Arr_1, ..., Arr_m$, and ϕ is quantifier-free formula containing at most the variables in $\underline{i} \cup \underline{a}$ as free variables. The former are called $\forall^I$-formulae and the latter $\exists^I$-formulae. An $\exists^I$-formula $\exists \underline{i}.\phi$ is *primitive* when ϕ is a conjunction of literals; it is *differentiated* when it is primitive and ϕ contains as a conjunct the disequation $i_k \neq i_l$ for each $1 \leq k < l \leq length(\underline{i})$. By applying simple logical manipulations, it is always possible to transform any $\exists^I$-formula into a disjunction of primitive differentiated ones. To specify transitions, we use a particular class of formulae (called *transition formulae*) corresponding to a generalization of the usual notion of guarded assignment system:

$$\exists i_1, i_2, e. \left(G(i_1, i_2, e, \underline{a}) \ \wedge \ \bigwedge_{k=1}^{m} \forall j.\, a'_k[j] = Upd_k(j, i_1, i_2, e, \underline{a}) \right),$$

where i_1, i_2 are variables of sort Ind (having at most two existentially quantified variables is not too restrictive since many disparate systems can be formalized in this format as shown by the experiments available on-line), e is a variable of sort

$Elem_k$ (for some $k = 1..., m$), $\underline{a}$ is a tuple of array state variables, a_k (in $\underline{a}$) is the actual value of a state variable and a'_k is its value after the execution of the transition, G is a conjunction of literals (called the *guard*), and Upd_k is a function defined by cases (for $k = 1, ..., m$), i.e. by suitably nested if-then-else expressions whose conditionals are again conjunctions of literals. The format for transition formulae above—because of the presence of the existentially quantified variable e over data values—is the first significant amelioration of the actual version of MCMT as it allows one to specify classes of systems which were not previously accepted by the tool such as real time systems or those with non-deterministic updates. Notice that the theory T_{E_k} over the sort $Elem_k$ of the variable e must be Linear Arithmetic (over the integers or the reals). This limitation allows us to maintain the closure of the class of $\exists^I$-formulae under pre-image computation by exploiting quantifier elimination (implemented only in the latest version of MCMT).

Safety problem. Let I be a $\forall^I$-formula describing the set of initial states, Tr a finite set of transition formulae, and U an $\exists^I$-formula for the set of unsafe states. The *safety problem* solved by MCMT consists in establishing whether there exists an $n \geq 0$ such that the formula

$$I(\underline{a}^0) \wedge \tau(\underline{a}^0, \underline{a}^1) \wedge \cdots \wedge \tau(\underline{a}^{n-1}, \underline{a}^n) \wedge U(\underline{a}^n) \tag{1}$$

is A_I^E-satisfiable, where $\underline{a}^h = a_1^h, ..., a_m^h$ for $h = 0, ..., n$, and $\tau := \bigvee_{\tau_i \in Tr} \tau_i$. If there is no such n, then the system is *safe* (w.r.t. U); otherwise, it is said to be *unsafe* since the A_I^E-satisfiability of (1) implies the existence of a run (of length n) leading the system from a state in I to a state in U.

4 The Main Loop: Deductive Backward Reachability

MCMT implements backward reachability to solve safety problems. For $n \geq 0$, the *n-pre-image* of an $\exists^I$-formula $K(\underline{a})$ is $Pre^0(\tau, K) := K$ and $Pre^{n+1}(\tau, K) := Pre(\tau, Pre^n(\tau, K))$, where $Pre(\tau, K) := \exists \underline{a}'.(\tau(\underline{a}, \underline{a}') \wedge K(\underline{a}'))$. It is easy to show [6] that the class of $\exists^I$-formulae is closed under pre-image computation under the assumption that T_{E_k} admits elimination of quantifiers (if an existentially quantified variable of sort $Elem_k$ occurs in a transition formula). The formula $BR^n(\tau, U) := \bigvee_{i=0}^n Pre^i(\tau, U)$ represents the set of states which are backward reachable from the states in U in at most $n \geq 0$ steps. So, backward reachability consists of computing $BR^n(\tau, U)$ for increasing values of n and checking whether $BR^n(\tau, U) \wedge I$ is A_I^E-satisfiable or $\neg(BR^n(\tau, U) \to BR^{n-1}(\tau, U))$ is A_I^E-unsatisfiable. In the first case (*safety* test), one concludes the unsafety of the system while in the second (*fixed-point* test), it is possible to stop computing pre-images as no new states can be reached and, if the safety test has been passed, one can infer the safety of the system.

Figure 1 introduces the Tableaux-like calculus used by MCMT to implement backward reachability [7]. We initialize the tableau with the $\exists^I$-formula $U(\underline{a})$ representing the set of unsafe states. The computation of the pre-image is realized by applying rule PreImg (we use square brackets to indicate the applicability

$$\frac{K \quad [K \text{ is primitive differentiated}]}{\mathsf{Pre}(\tau_1, K) \mid \cdots \mid \mathsf{Pre}(\tau_m, K)}\ \mathsf{PreImg} \qquad \frac{K}{K_1 \mid \cdots \mid K_n}\ \mathsf{Beta}$$

$$\frac{K \quad [K \text{ is } A_I^E\text{-unsatisfiable}]}{\times}\ \mathsf{NotAppl} \qquad \frac{K \quad [I \wedge K \text{ is } A_I^E\text{-satisfiable}]}{\mathsf{UnSafe}}\ \mathsf{Safety}$$

$$\frac{K \quad [K \wedge \bigwedge\{\neg K' | K' \preceq K\} \text{ is } A_I^E\text{-unsatisfiable}]}{\times}\ \mathsf{FixPoint}$$

Fig. 1. The calculus underlying MCMT

condition of a rule), where $\mathsf{Pre}(\tau_h, K)$ computes the $\exists^I$-formula which is logically equivalent to $Pre(\tau_h, K)$. Since the $\exists^I$-formulae labeling the consequents of the rule PreImg may not be primitive and differentiated (because of nested if-then-else expressions and incompleteness of variable distinction), we need to apply the Beta rule to an $\exists^I$-formula so as to eliminate the conditionals by case-splitting and derive $K_1, \ldots, K_n$ primitive differentiated $\exists^I$-formulae whose disjunction is A_I^E-equivalent to K. By repeatedly applying PreImg and Beta, it is possible to build a tree whose nodes are labelled by $\exists^I$-formulae whose disjunction is equivalent to $BR^n(\tau, U)$ for some $n \geq 0$. Indeed, there is no need to fully expand the tree; it is useless to apply the rule PreImg to a node ν labelled by an A_I^E-unsatisfiable $\exists^I$-formula (rule NotAppl). One can terminate the whole search because of the safety test (rule Safety), in which case one can extract from the branch a *bad trace*, i.e. a sequence of transitions leading the array-based system from a state satisfying I to one satisfying U. A branch can be terminated by the fixed-point test described by rule FixPoint, where $K' \preceq K$ means that K' is a primitive differentiated $\exists^I$-formula labeling a node preceding the node labeled by K (nodes can be ordered according to the strategy for expanding the tree).

For the effectiveness of rules Safety and FixPoint, it is necessary to check the A_I^E-satisfiability of $\exists^I\forall^I$-formulae, i.e. formulae containing an alternation of quantifiers over variables of sort *Ind*. In MCMT, we have integrated a quantifier instantiation procedure with SMT solving techniques for quantifier-free formulae, augmented with heuristics to avoid the generation of useless instances and incrementality of satisfiability checks [8]. The technique is complete under some hypotheses on T_I and the T_{E_k}'s [6]. Even if such hypotheses are not satisfied, the instantiation procedure can still be soundly used without loss of precision for a final safety result of the backward reachability procedure, although its termination is less guaranteed. This point is particularly delicate and merits some discussion. Consider the verification of a mutual exclusion protocol (due to Szymanski) which can be considered as a typical example of problems about parametrised systems. To show the safety of this problem, MCMT generates 1153 satisfiable and 4043 unsatisfiable SMT problems. While our quantifier instantiation procedure with Yices is capable of solving all SMT problems, Yices alone with its quantifier handling techniques returns 'unknown' on all the 1153 satisfiable instances while it can solve 2223 of the unsatisfiable ones and returns 'unknown' on the remaining 1820. We are currently adding the capability of generating satisfiability problems in the SMT-LIB format to MCMT so as to evaluate the quantifier handling procedures available in various SMT solvers.

The main novelty of the latest version of MCMT is the more extensive support for invariant synthesis, abstraction, and *acceleration* for computing the repeated application of a sub-set of the transitions an arbitrary number of times in a single step. Invariant synthesis has been introduced in [7] but the implementation was able to generate universally quantified invariants with just one variable. MCMT v. 1.0 supports the generation of invariants with up to two universal quantifiers. While performing backward reachability (which is always precise), candidate invariants are guessed according to some heuristics [7] and then a resource bounded (secondary) backward reachability is used to keep or discard the candidates. The invariants found in this way are used in the main backward reachability procedure when checking for fix-points. The present version of MCMT features also a new technique for the synthesis of invariants, named *signature abstraction*: it consists of projecting away (by quantifier elimination, whenever possible) those literals containing a sub-set of the array variables; thereby obtaining an over-approximation of the set of reachable states. This is without loss of precision, since it is done in the secondary backward reachability procedure for invariant synthesis while the main procedure continues to compute the set of backward reachable states precisely. The last novelty of MCMT v. 1.0 is some support for acceleration along the lines of [4] in the hope of a better convergence of the backward reachable procedure; this is often the case for systems formalized by arithmetic constraints such as Petri nets.

5 Experiments

To show the flexibility and the performance of MCMT, we have taken some pain to build a library of benchmarks in the format accepted by our tool by translating safety problems from a variety of sources, such as the distributions of the infinite state model checkers described in [2,3] or imperative programs manipulating arrays taken from standard books about algorithms. For lack of space, we include here only an excerpt of the experiments (see the tool web-page for a full report).

We divide the problems in four categories: mutual exclusion (M) and cache coherence (C) protocols, imperative programs manipulating arrays (I), and heterogeneous (H) problems. We tried the tool in two configurations: the "Default Setting" is when MCMT is invoked without any option and the "Best Setting" is when the tool is run with some options turned on. In Table 1, the column 'd' is the depth of the tableaux obtained by applying the rules in Figure 1, '#n' is the number of nodes in the tableaux, '#del' is the number of subsumed nodes, '#SMT' is the number of invocations to Yices, '#i' is the number of invariants found by MCMT (for the "Default Setting," column '#i' is not shown because MCMT's default is to turn off invariant synthesis), and 'time' is the total amount of time (in seconds) taken by the tool to solve the problem on a Pentium Intel 1.73 GHz with 1 Gb Sdram running Linux Gentoo. In the cases when the tool seemed to diverge, we aborted execution, and put 't(ime)o(out)' in the column of timings and leave the others empty ('-'). All systems—except 'GermanBug' (a bugged version of 'German07')—are certified to be safe by MCMT while for 'GermanBug,' the tool returns an error trace consisting of 16 transitions. Invariant

Table 1. Some experimental results

	Default setting					Best setting					
Problem	d	#n	#del	#SMT	time	d	#n	#del	#SMT	#i	time
Lamport (M)	23	913	242	47574	120.62	23	248	42	19254	7	32.84
RickAgr (M)	13	458	119	35355	187.04	13	458	119	35355	0	187.04
Szymanski_at (M)	23	1745	311	424630	540.19	9	22	10	2987	42	1.25
German07 (C)	26	2442	576	121388	145.68	26	2442	576	121388	0	145.68
GermanBug (C)	16	1631	203	41497	49.70	16	1631	203	41497	0	49.70
GermanPFS (C)	33	11605	2755	858184	1861.0	33	11141	2673	784168	149	1827.0
SelSort (I)	-	-	-	-	to	5	13	2	1141	11	0.62
Strcat (I)	-	-	-	-	to	2	2	2	80	2	0.07
Strcmp (I)	-	-	-	-	to	2	1	1	21	3	0.01
Fischer (H)	10	16	2	336	0.16	10	16	2	336	0	0.16
Ticket (H)	-	-	-	-	to	3	4	2	201	10	0.06

synthesis (especially the signature abstraction technique introduced in this version of the tool) is helpful to reduce the solving time for problems in (M), and to obtain termination for those in (I), but has no effect on problems in (C).

References

1. Abdulla, P.A., Cerans, K., Jonsson, B., Tsay, Y.-K.: General decidability theorems for infinite-state systems. In: LICS, pp. 313–321 (1996)
2. Abdulla, P.A., Delzanno, G., Henda, N.B., Rezine, A.: Regular model checking without transducers. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 721–736. Springer, Heidelberg (2007)
3. Abdulla, P.A., Delzanno, G., Rezine, A.: Parameterized verification of infinite-state processes with global conditions. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 145–157. Springer, Heidelberg (2007)
4. Bérard, B., Fribourg, L.: Reachability Analysis of (Timed) Petri Nets Using Real Arithmetic. In: Baeten, J.C.M., Mauw, S. (eds.) CONCUR 1999. LNCS, vol. 1664, pp. 178–193. Springer, Heidelberg (1999)
5. Flanagan, C., Qadeer, S.: Predicate abstraction for software verification. In: POPL, pp. 191–202. ACM, New York (2002)
6. Ghilardi, S., Nicolini, E., Ranise, S., Zucchelli, D.: Towards SMT Model-Checking of Array-based Systems. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 67–82. Springer, Heidelberg (2008)
7. Ghilardi, S., Ranise, S.: Goal Directed Invariant Synthesis for Model Checking Modulo Theories. In: Giese, M., Waaler, A. (eds.) TABLEAUX 2009. LNCS (LNAI), vol. 5607, pp. 173–188. Springer, Heidelberg (2009)
8. Ghilardi, S., Ranise, S.: Model Checking Modulo Theory at work: the intergration of Yices in MCMT. In: AFM (co-located with CAV'09) (2009)
9. Ghilardi, S., Ranise, S., Valsecchi, T.: Light-Weight SMT-based Model-Checking. In: AVOCS 07-08, ENTCS (2008)
10. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254. Springer, Heidelberg (1997)
11. Lahiri, S.K., Bryant, R.E.: Predicate abstraction with indexed predicates. ACM Transactions on Computational Logic (TOCL) 9(1) (2007)

On Hierarchical Reasoning in Combinations of Theories

Carsten Ihlemann and Viorica Sofronie-Stokkermans

Max-Planck-Institut für Informatik, Campus E1 4, Saarbrücken, Germany

Abstract. In this paper we study theory combinations over non-disjoint signatures in which hierarchical and modular reasoning is possible. We use a notion of locality of a theory extension parameterized by a closure operator on ground terms. We give criteria for recognizing these types of theory extensions. We then show that combinations of extensions of theories which are local in this extended sense also have a locality property and hence allow modular and hierarchical reasoning. We thus obtain parameterized decidability and complexity results for many (combinations of) theories important in verification.

1 Introduction

Many problems in mathematics and computer science can be reduced to proving the satisfiability of conjunctions of literals in a background theory (which can be the extension of a base theory with additional functions – e.g., free, monotone, or recursively defined – or a combination of theories). Considerable work has been dedicated to the task of identifying situations where reasoning in extensions and combinations of theories can be done efficiently and accurately. The most important issues which need to be addressed in this context are: (i) finding possibilities of reducing the search space without losing completeness, and (ii) making modular or hierarchical reasoning possible.

In [10,17] Givan and McAllester introduced the so-called "local inference systems" (for which validity of ground Horn clauses can be checked in polynomial time). A link between this proof theoretic notion of locality and algebraic arguments used for identifying classes of algebras with a word problem decidable in PTIME [4] was established in [7]. In [8,21] these results were further extended to so-called *local extensions* of theories. Locality phenomena were also studied in the verification literature, mainly motivated by the necessity of devising methods for efficient reasoning in theories of pointer structures [18] and arrays [3]. In [14] we showed that these results are instances of a general concept of locality of a theory extension – parameterized by a closure operator on ground terms.

Efficient reasoning in combinations of theories is also very important. Methods for checking satisfiability of conjunctions of ground literals in combinations of theories which have disjoint signatures, or only share constants, are well studied. The Nelson/Oppen combination procedure [19] can be applied for combining decision procedures of stably infinite theories over disjoint signatures; various extensions have been established either by relaxing the requirement that the theories to be combined are stably-infinite [26]; or by relaxing the requirement that the theories to be combined have disjoint signatures [1,25,9]. These extensions require additional conditions, e.g. generalizations of stable infinity on the component theories, noetherianity of the shared theory. Since the notion of local extensions we studied [21] imposes no major restrictions on the base theory, it offers interesting, orthogonal criteria for the transfer of decidability in combinations of theories. In this paper we show that efficient reasoning techniques can be provided for combinations of local theory extensions as well. To this end, we present new results on preservation of locality and Ψ-locality of theory extensions under theory combinations, extending earlier results in [23] (cf. also [22]). The main results of this paper can be summarized as follows:

- We present semantic characterizations for various notions of locality (possibly parameterized by a closure operator on ground terms).
- We present various results on transfer of locality (and hence also of decidability), some with a model theoretical flavor.
- We identify increasingly complex conditions under which locality is preserved under taking unions of theories. We strengthen some results in [23,22] by considering the embeddability conditions (EEmb_w) instead of (Comp_w) and by considering combinations of Ψ_i-local extensions of a theory. We briefly discuss the way these ideas are implemented in H-PILoT.

The paper is structured as follows. Section 2 contains generalities on local theory extensions, partial algebras, weak validity and embeddability. In Sect. 3 we present ways of recognizing locality. In Sect. 4 we give semantical characterizations of locality; these are used in Sect. 5 to transfer locality results. Section 6 presents our results on combinations of local theory extensions, and a description of the way we implemented hierarchical reasoning in such combinations.

2 Preliminaries

We assume standard definitions from first-order logic. In this paper, (logical) theories are simply sets of sentences.

Extensions of theories. Let Π_0=(Σ_0, Pred) be a signature, and $\mathcal{T}_0$ be a "base" theory with signature Π_0. We consider the following types of extensions of $\mathcal{T}_0$:

- *Extensions with sets of clauses* are extensions $\mathcal{T} := \mathcal{T}_0 \cup \mathcal{K}$ of $\mathcal{T}_0$ with new function symbols Σ (called *extension functions*) whose properties are axiomatized using a set $\mathcal{K}$ of (universally closed) clauses in the extended signature $\Pi = (\Sigma_0 \cup \Sigma, \mathsf{Pred})$, which contain function symbols in Σ.
- *Extensions with augmented clauses* are extensions $\mathcal{T} := \mathcal{T}_0 \cup \mathcal{K}$ with new function symbols Σ whose properties are axiomatized using a set $\mathcal{K}$ of formulas of the form $\forall \bar{x}\, (\Phi(\bar{x}) \vee D(\bar{x}))$ where $\Phi(\bar{x})$ is an *arbitrary formula* in the base signature Π_0 and $D(\bar{x})$ is a clause in the extended signature Π, which contains at least one function symbol of Σ.

If for every formula $\forall \bar{x}\, (\Phi(\bar{x}) \vee D(\bar{x})) \in \mathcal{K}$, $\Phi(\bar{x})$ is universal we speak of *extension by universal augmented clauses*; if $\Phi(\bar{x})$ belongs to a certain class $\mathcal{F}$ of Π_0-formulae we speak of *extension by $\mathcal{F}$-augmented clauses.*

Example 1. The following examples illustrate the notions above:

(i) Let $\mathcal{T}_0$ be the theory of Presburger arithmetic, with signature Π_0. Let $\Sigma = \{f\}$ where f is a new function symbol, and let $\mathcal{K}_f = \{\forall x, y, z\, (y \neq z \rightarrow f(x, y) \neq f(x, z))\}$ be an axiomatization for the injectivity of f in its second argument. Then $\mathcal{T}_1 := \mathcal{T}_0 \cup \mathcal{K}_f$ is an extension of $\mathcal{T}_0$ with the set $\mathcal{K}$ of clauses and f is an extension function.

(ii) Let $\mathcal{T}_1$ be the theory defined at (i) and let $\Sigma = \{g\}$, where $g \notin \Sigma_0 \cup \Sigma$ and let $\mathcal{K}_g = \{\forall x, y([\forall z(z \neq y \rightarrow f(x, z) < f(x, y))] \rightarrow g(x) = f(x, y))\}$. Then $\mathcal{T} := \mathcal{T}_1 \cup \mathcal{K}_g$ is an extension of $\mathcal{T}_1$ with a set $\mathcal{K}_g$ of augmented clauses (in fact $\mathcal{F}$-augmented clauses where $\mathcal{F}$ is the $\exists$-fragment of $\mathcal{T}_1$).

Our goal is to address proof tasks of the form $G \models_{\mathcal{T}_0 \cup \mathcal{K}} \perp$, where G is a set of ground clauses with additional (fresh) constants in a countable set C, i.e. in the signature $\Pi^C = (\Sigma_0 \cup \Sigma \cup C, \mathsf{Pred})$.

For the case of extensions $\mathcal{T}_0 \cup \mathcal{K}$ by augmented clauses we also consider the more general task of checking satisfiability problems of the form $\Gamma \models_{\mathcal{T}_0 \cup \mathcal{K}} \perp$, where Γ is a conjunction of sentences of the form $\Phi_0 \vee D$, where D is a ground clause in the signature Π^C, and Φ_0 is a Π_0^C-sentence.

We also consider combinations of extensions $(\mathcal{T}_0 \cup \mathcal{K}_1)$ and $(\mathcal{T}_0 \cup \mathcal{K}_2)$ of the base theory $\mathcal{T}_0$, where $\mathcal{K}_i$ are sets of (augmented) clauses over $(\Sigma_0 \cup \Sigma_i, \mathsf{Pred})$. Our proof tasks G, then, will be in the signature $(\Sigma_0 \cup \Sigma_1 \cup \Sigma_2 \cup C, \mathsf{Pred})$. Using new constants, we can always separate G into a $(\Sigma_0 \cup \Sigma_1 \cup C, \mathsf{Pred})$-part G_1, a $(\Sigma_0 \cup \Sigma_2 \cup C, \mathsf{Pred})$-part G_2, and a Π_0-part G_0.

Locality conditions. Let $\mathcal{T}_0$ be an arbitrary theory with signature $\Pi_0 = (\Sigma_0, \mathsf{Pred})$, where the set of function symbols is Σ_0. Let $\Pi = (\Sigma_0 \cup \Sigma, \mathsf{Pred}) \supseteq \Pi_0$ be an extension by a non-empty set Σ of new function symbols and $\mathcal{K}$ be a set of (implicitly universally closed) clauses in the extended signature. Let C be a fixed countable set of fresh constants. We say that an extension $\mathcal{T}_0 \cup \mathcal{K}$ of $\mathcal{T}_0$ is *local* if it satisfies the following condition[1]:

(Loc) For every set G of ground clauses in Π^C it holds that
$\mathcal{T}_0 \cup \mathcal{K} \cup G \models \perp$ if and only if $\mathcal{T}_0 \cup \mathcal{K}[G] \cup G \models \perp$

where $\mathcal{K}[G]$ consists of those instances of $\mathcal{K}$ in which the terms starting with *extension functions* are in the set $\mathsf{est}(\mathcal{K}, G)$ of extension ground terms (i.e. terms starting with a function in Σ) which already occur in G or $\mathcal{K}$.

The notion of local theory extension generalizes the notion of *local theories* [10,17,11,7]. In [14] we generalized condition (Loc) by considering operators on ground terms. This allows us to be more flexible w.r.t. the instances needed.

[1] It is easy to check that the formulation we give here and that in [21] are equivalent.

Definition 2. *With the notations above, let T be a set of ground terms in the signature Π^C. We denote by $\mathcal{K}[T]$ the set of all instances of $\mathcal{K}$ in which the terms starting with a function symbol in Σ are in T. Formally:*

$$\mathcal{K}[T] := \{\varphi\sigma \mid \forall \bar{x}.\, \varphi(\bar{x}) \in \mathcal{K}, \text{ where (i) if } f \in \Sigma \text{ and } t = f(t_1, ..., t_n) \text{ occurs in } \varphi\sigma \text{ then } t \in T; \text{ (ii) if } x \text{ is a variable that does not appear below some } \Sigma\text{-function in } \varphi \text{ then } \sigma(x) = x\}.$$

Definition 3. *Let Ψ be a map associating with every set T of ground terms a set $\Psi(T)$ of ground terms. For any set G of (augmented) ground Π^C-clauses we write $\mathcal{K}[\Psi_{\mathcal{K}}(G)]$ for $\mathcal{K}[\Psi(\mathsf{est}(\mathcal{K}, G))]$. We define versions of locality*[2] *in which the set of terms used in the instances of the axioms is described using the map Ψ. Let $\mathcal{T}_0 \cup \mathcal{K}$ be an extension of $\mathcal{T}_0$ with clauses in $\mathcal{K}$. We define:*

(Loc^Ψ) *For every set G of ground clauses in Π^C it holds that $\mathcal{T}_0 \cup \mathcal{K} \cup G \models \bot$ if and only if $\mathcal{T}_0 \cup \mathcal{K}[\Psi_{\mathcal{K}}(G)] \cup G \models \bot$.*

Let $\mathcal{T}_0 \cup \mathcal{K}$ be an extension of $\mathcal{T}_0$ with augmented clauses in $\mathcal{K}$. We define:

(ELoc^Ψ) *For every set of formulas $\Gamma = \Gamma_0 \cup G$, where Γ_0 is a Π_0-sentence and G is a set of ground Π^C-clauses, it holds that $\mathcal{T}_0 \cup \mathcal{K} \cup \Gamma \models \bot$ if and only if $\mathcal{T}_0 \cup \mathcal{K}[\Psi_{\mathcal{K}}(G)] \cup \Gamma \models \bot$.*

Extensions satisfying condition (Loc^Ψ) are called Ψ-local; we refer to (ELoc^Ψ) as the extended Ψ-locality *condition.* Finite *locality conditions* ((E)Loc^Ψ_f) *are defined restricting the locality conditions to hold for* finite *sets G of ground clauses.*

Example 4. Local theory extensions are Ψ-local, where Ψ is the identity operator. The order-local theories introduced in [2] satisfy a Ψ-locality condition, where for every set T of ground clauses $\Psi(T) = \{s \mid s$ ground term and $s \preceq t$ for some $t \in T\}$, where $\prec$ is the order on terms considered in [2].

Hierarchical reasoning. Let $\mathcal{T}_0 \subseteq \mathcal{T} = \mathcal{T}_0 \cup \mathcal{K}$ be a theory extension satisfying ((E)Loc^Ψ). To check the satisfiability w.r.t. $\mathcal{T}$ of a formula $\Gamma_0 \cup G$, where Γ_0 is a Π_0^C-sentence[3] and G is a set of ground Π^C-clauses, we proceed as follows:

Step 1: By locality, $\mathcal{T} \cup \Gamma_0 \cup G \models \bot$ iff $\mathcal{T}_0 \cup \mathcal{K}[\Psi_{\mathcal{K}}(G)] \cup \Gamma_0 \cup G \models \bot$.

Step 2: Purification. We purify $\mathcal{K}[\Psi_{\mathcal{K}}(G)] \cup G$ (by introducing, in a bottom-up manner, new constants c_t for subterms $t = f(g_1, \dots, g_n)$ with $f \in \Sigma$, g_i ground Π_0^C-terms, and corresponding definitions $c_t \approx t$) and obtain the set of formulae $\mathcal{K}_0 \cup G_0 \cup \Gamma_0 \cup D$, where D consists of definitions $f(g_1, \dots, g_n) \approx c$, where $f \in \Sigma$, c is a constant, $g_1, \dots, g_n$ are ground Π_0^C-terms, and $\mathcal{K}_0, G_0, \Gamma_0$ are Π_0^C-formulae.

Step 3: Reduction to testing satisfiability in $\mathcal{T}_0$. We reduce the problem to testing satisfiability in $\mathcal{T}_0$ by replacing D with the following set of clauses:

$$\mathsf{Con}_0 = \{\bigwedge_{i=1}^{n} c_i \approx d_i \rightarrow c = d \mid f(c_1, \dots, c_n) \approx c, f(d_1, \dots, d_n) \approx d \in D\}.$$

This yields a sound and complete hierarchical reduction to a satisfiability problem in the base theory $\mathcal{T}_0$:

[2] It is easy to check that the formulation we give here and that in [14] are equivalent.

[3] In the case of condition (Loc^Ψ), $\Gamma_0 = \top$.

Theorem 5 ([14]). *Let $\mathcal{K}$ and $\Gamma_0 \wedge G$ be as specified above. Assume that $\mathcal{T}_0 \subseteq \mathcal{T}_0 \cup \mathcal{K}$ satisfies condition $((\mathsf{E})\mathsf{Loc}^{\Psi})$. Let $\mathcal{K}_0 \cup G_0 \cup \Gamma_0 \cup \mathsf{Con}_0$ be obtained from $\mathcal{K}[\Psi_{\mathcal{K}}(G)] \cup \Gamma_0 \cup G$ by purification (cf. Step 2). The following are equivalent:*

(1) $\mathcal{T}_0 \cup \mathcal{K} \cup \Gamma_0 \cup G \models \perp$.
(2) $\mathcal{T}_0 \cup \mathcal{K}_0 \cup G_0 \cup \Gamma_0 \cup \mathsf{Con}_0 \models \perp$.

Thus, satisfiability of goals $\Gamma_0 \cup G$ as above w.r.t. $\mathcal{T}$ is decidable provided $\mathcal{K}[\Psi_{\mathcal{K}}(G)]$ is finite and $\mathcal{K}_0 \cup G_0 \cup \Gamma_0 \cup \mathsf{Con}_0$ belongs to a decidable fragment of $\mathcal{T}_0$.

Implementation. This method is implemented in the program H-PILoT (Hierarchical Proving by Instantiation in Local Theory Extensions) ([13]). H-PILoT carries out a hierarchical reduction to $\mathcal{T}_0$ step-by-step if the user specifies different levels for the extension functions in a chain of theory extensions. Standard SMT provers or specialized provers can be used for testing the satisfiability of the formulas obtained after the reduction. If the result of the reduction is a satisfiable problem, H-PILoT is able to *generate a model.* Ψ-locality is handled for the *array property fragment* and a fragment of the theory of pointers [3,18,14], which are fully integrated into H-PILoT. In this paper we establish ways of recognizing (Ψ-)locality for wider classes of theories.

Since local extensions can be recognized by showing that certain partial models embed into total ones, we introduce the main definitions here.

Partial Structures. Let $\Pi = (\Sigma, \mathsf{Pred})$ be a first-order signature with set of function symbols Σ and set of predicate symbols Pred. A *partial Π-structure* is a structure $\mathcal{A} = (A, \{f_{\mathcal{A}}\}_{f \in \Sigma}, \{P_{\mathcal{A}}\}_{P \in \mathsf{Pred}})$, where A is a non-empty set, for every $f \in \Sigma$ with arity n, $f_{\mathcal{A}}$ is a partial function from A^n to A, and for every $P \in \mathsf{Pred}$, $P_{\mathcal{A}} \subseteq A^n$. We consider constants (0-ary functions) to be always defined. $\mathcal{A}$ is called a *total structure* if the functions $f_{\mathcal{A}}$ are all total. Given a (total or partial) Π-structure $\mathcal{A}$ and $\Pi_0 \subseteq \Pi$ we denote the reduct of $\mathcal{A}$ to Π_0 by $\mathcal{A}|_{\Pi_0}$.

The notion of evaluating a term t with variables X w.r.t. an assignment $\beta : X \to A$ for its variables in a partial structure $\mathcal{A}$ is the same as for total algebras, except that the evaluation is undefined if $t = f(t_1, \dots, t_n)$ and at least one of $\beta(t_i)$ is undefined, or else $(\beta(t_1), \dots, \beta(t_n))$ is not in the domain of $f_{\mathcal{A}}$. Recall that for total Π-structures $\mathcal{A}$ and $\mathcal{B}$, $\varphi : \mathcal{A} \to \mathcal{B}$ is an embedding if and only if it is an injective homomorphism and has the property that for every $P \in \mathsf{Pred}$ with arity n and all $(a_1, \dots, a_n) \in A^n$, $(a_1, \dots, a_n) \in P_{\mathcal{A}}$ iff $(\varphi(a_1), \dots, \varphi(a_n)) \in P_{\mathcal{B}}$. In particular, an embedding preserves the truth of all literals. A similar notion can be defined for partial structures.

Definition 6 (Weak Π-Embedding). *A* weak Π-embedding *between partial Π-structures $\mathcal{A} = (A, \{f_{\mathcal{A}}\}_{f \in \Sigma}, \{P_{\mathcal{A}}\}_{P \in \mathsf{Pred}})$ and $\mathcal{B} = (B, \{f_{\mathcal{B}}\}_{f \in \Sigma}, \{P_{\mathcal{B}}\}_{P \in \mathsf{Pred}})$ is a total map $\varphi : A \to B$ such that*

(1) whenever $f_{\mathcal{A}}(a_1, \dots, a_n)$ is defined (in $\mathcal{A}$), then $f_{\mathcal{B}}(\varphi(a_1), \dots, \varphi(a_n))$ is defined (in $\mathcal{B}$) and $\varphi(f_{\mathcal{A}}(a_1, \dots, a_n)) = f_{\mathcal{B}}(\varphi(a_1), \dots, \varphi(a_n))$, for all $f \in \Sigma$;
(2) for every $P \in \mathsf{Pred}$ with arity n and every $a_1, \dots, a_n \in \mathcal{A}$, $(a_1, \dots, a_n) \in P_{\mathcal{A}}$ if and only if $(\varphi(a_1), \dots, \varphi(a_n)) \in P_{\mathcal{B}}$.

Definition 7 (Weak Validity). *Let $\mathcal{A}$ be a partial Π-algebra and $\beta : X \to A$ a valuation for its variables. We define weak validity w.r.t. $(\mathcal{A}, \beta)$ as follows:*

(1) $(\mathcal{A}, \beta) \models_w t \approx u$ if (i) both $\beta(t)$ and $\beta(u)$ are defined and equal; or (ii) at least one of the terms $\beta(t)$, $\beta(u)$ is undefined.
(2) $(\mathcal{A}, \beta) \models_w t \not\approx u$ if (i) both $\beta(t)$ and $\beta(u)$ are defined but different; or (ii) at least one of the terms $\beta(t)$, $\beta(u)$ is undefined.
(3) $(\mathcal{A}, \beta) \models_w P(t_1, \dots, t_n)$ if (i) $\beta(t_1), \dots, \beta(t_n)$ are all defined and $(\beta(t_1), \dots, \beta(t_n)) \in P_{\mathcal{A}}$; or (ii) at least one $\beta(t_i)$, $1 \leq i \leq n$, is undefined.
(4) $(\mathcal{A}, \beta) \models_w \neg P(t_1, \dots, t_n)$ if (i) $\beta(t_1), \dots, \beta(t_n)$ are all defined and $(\beta(t_1), \dots, \beta(t_n)) \notin P_{\mathcal{A}}$; or (ii) at least one $\beta(t_i)$, $1 \leq i \leq n$, is undefined.

$(\mathcal{A}, \beta)$ weakly satisfies a clause C *(notation: $(\mathcal{A}, \beta) \models_w C$) if it satisfies at least one literal in C. $\mathcal{A}$ is a* weak partial model *of a set of clauses $\mathcal{K}$ if $(\mathcal{A}, \beta) \models_w C$ for every valuation β and every clause C in $\mathcal{K}$.*

If $\mathcal{T} = \mathcal{T}_0 \cup \mathcal{K}$ is an extension of a Π_0-theory $\mathcal{T}_0$ with new function symbols in Σ and (augmented) clauses $\mathcal{K}$, we denote by $\mathsf{PMod}_w(\Sigma, \mathcal{T})$ the set of weak partial models of $\mathcal{T}$ whose Σ_0-functions are total.

3 Recognizing Ψ-Local Theory Extensions

In [21] we proved that if all weak partial models of an extension $\mathcal{T}_0 \cup \mathcal{K}$ of a base theory $\mathcal{T}_0$ with total base functions can be embedded into a total model of the extension, then the extension is local. In [14] we lifted these results to Ψ-locality. We recall these results and then extend them to obtain semantical *characterizations* of various types of Ψ-locality. In what follows, let $\mathcal{T}_0$ be a Π_0-theory, and $\mathcal{T}_0 \subseteq \mathcal{T}_0 \cup \mathcal{K} = \mathcal{T}$ a theory extension with functions in Σ and (augmented) clauses $\mathcal{K}$ and let Ψ be as in Definition 3.

Definition 8. *Let $\mathcal{A} = (A, \{f_{\mathcal{A}}\}_{f \in \Sigma_0 \cup \Sigma}, \{P_{\mathcal{A}}\}_{P \in \mathsf{Pred}})$ be a partial Π^C-structure with total Σ_0-functions. We denote by Π^A the extension of the signature Π with constants from A. We denote by $\mathcal{D}(\mathcal{A})$ the following set of ground Π^A-terms:*

$$\mathcal{D}(\mathcal{A}) := \{f(a_1, ..., a_n) \mid f \in \Sigma, a_i \in A, i = 1, \dots, n, f_{\mathcal{A}}(a_1, ..., a_n) \text{ is defined }\}.$$

Let $\mathsf{PMod}^{\Psi}_w(\Sigma, \mathcal{T})$ be the class of all weak partial models $\mathcal{A}$ of $\mathcal{T}_0 \cup \mathcal{K}$ in which the Σ-functions are partial, all other functions are total and all terms in $\Psi(\mathsf{est}(\mathcal{K}, \mathcal{D}(\mathcal{A})))$ are defined (in the extended structure $\mathcal{A}^A$ with constants from A).

We consider the following embeddability properties of partial algebras:

(Emb^{Ψ}_w) Every $\mathcal{A} \in \mathsf{PMod}^{\Psi}_w(\Sigma, \mathcal{T})$ weakly embeds into a total model of $\mathcal{T}$.

(Comp^{Ψ}_w) Every $\mathcal{A} \in \mathsf{PMod}^{\Psi}_w(\Sigma, \mathcal{T})$ weakly embeds into a total model $\mathcal{B}$ of $\mathcal{T}$ such that $\mathcal{A}|_{\Pi_0}$ and $\mathcal{B}|_{\Pi_0}$ are isomorphic.

Variants ($\mathsf{Comp}^{\Psi}_{w,f}$) and ($\mathsf{Emb}^{\Psi}_{w,f}$) can be obtained by requiring embeddability only for extension functions with a finite domain of definition.

When establishing links between locality and embeddability we require that the extension clauses in $\mathcal{K}$ are *flat* (*quasi-flat*) and *linear* w.r.t. Σ-functions.

Definition 9. *We distinguish between ground and non-ground clauses.*

Non-ground clauses: *An extension clause D is* quasi-flat *when all symbols below a Σ-function symbol in D are variables or ground Π_0-terms. D is* flat *when all symbols below a Σ-function symbol in D are variables. D is* linear *if whenever a variable occurs in two terms of D which start with Σ-functions, the terms are identical, and no such term contains two occurrences of a variable.*

Ground clauses: *A ground clause D is* flat *if all symbols below a Σ-function in D are constants. A ground clause D is* linear *if whenever a constant occurs in two terms in D whose root symbol is in Σ, the two terms are identical, and if no term which starts with a Σ-function contains two occurrences of the same constant.*

Definition 10. *With the above notations, let Ψ be a map associating with $\mathcal{K}$ and a set of Π^C-ground terms T a set $\Psi_{\mathcal{K}}(T)$ of Π^C-ground terms. We call $\Psi_{\mathcal{K}}$ a* term closure operator *if the following holds for all sets of ground terms T, T':*

(1) $\mathrm{est}(\mathcal{K}, T) \subseteq \Psi_{\mathcal{K}}(T)$,
(2) $T \subseteq T' \Rightarrow \Psi_{\mathcal{K}}(T) \subseteq \Psi_{\mathcal{K}}(T')$,
(3) $\Psi_{\mathcal{K}}(\Psi_{\mathcal{K}}(T)) \subseteq \Psi_{\mathcal{K}}(T)$,
(4) for any map $h : C \to C$, $\bar{h}(\Psi_{\mathcal{K}}(T)) = \Psi_{\bar{h}\mathcal{K}}(\bar{h}(T))$, where $\bar{h}$ is the canonical extension of h to extension ground terms.

In [14] we proved that if Ψ is a term closure operator then condition (Comp_w^{Ψ}) implies (ELoc^{Ψ}), provided the extension clauses are flat and linear. An analogous proof shows that (Emb_w^{Ψ}) implies (Loc^{Ψ}).[4] This allowed us to identify many examples of Ψ-local theory extensions. In [14] we showed that (i) a decidability result for the array property fragment in [3] is due to the Ψ-locality (for a certain Ψ) of the corresponding extensions of the many-sorted combination of Presburger arithmetic (for indices) with the given theory of elements, and (ii) a fragment of the theory of pointer structures studied in [18] satisfies a Ψ-locality property.

4 Semantical Characterizations of Locality

The aim of this section is to obtain semantical characterizations of the notions of Ψ-locality studied here. We first show that Ψ-locality implies Ψ-embeddability.

Theorem 11. *Assume that $\mathcal{K}$ is a family of Σ-flat clauses in the signature Π.*

(1) If $\mathcal{T}_0$ is a first-order theory and the extension $\mathcal{T}_0 \subseteq \mathcal{T}{=}\mathcal{T}_0 \cup \mathcal{K}$ satisfies (Loc^{Ψ}) then every model in $\mathsf{PMod}_w^{\Psi}(\Sigma, \mathcal{T})$ weakly embeds into a total model of $\mathcal{T}$.
(2) If $\mathcal{T}_0 \subseteq \mathcal{T}{=}\mathcal{T}_0 \cup \mathcal{K}$ satisfies (ELoc^{Ψ}) then every $\mathcal{A} \in \mathsf{PMod}_w^{\Psi}(\Sigma, \mathcal{T})$ weakly embeds into a total model $\mathcal{B}$ of $\mathcal{T}$ such that restriction of this embedding to the reducts to Π_0 of $\mathcal{A}$, $\mathcal{B}$ preserves the truth of all first-order Π_0-formulae.

[4] It is easy to see that if $\mathcal{T}_0$ is a first-order theory then in order to prove locality it is sufficient to restrict to countable partial models in the embeddability conditions.

Theorem 11(2) indicates that for characterizing extended Ψ-locality we need a notion weaker than completability. Therefore, instead of condition (Comp_w^Ψ) we now consider embeddings that are elementary w.r.t. the base language.

Definition 12. *A map $\varphi : \mathcal{A} \to \mathcal{B}$ is an* elementary embedding *iff it preserves and reflects* all *formulas, i.e., for every formula $F(x_1, \dots, x_n)$ with free variables $x_1, \dots, x_n$ and all elements $a_1, \dots, a_n$ from A,*

$$\mathcal{A} \models F(a_1, \dots, a_n) \textit{ if and only if } \mathcal{B} \models F(\varphi(a_1), \dots, \varphi(a_n)).$$

If φ is the inclusion, we say that $\mathcal{A}$ is an elementary substructure of $\mathcal{B}$ *(notation: $\mathcal{A} \preccurlyeq \mathcal{B}$). Two structures $\mathcal{A}, \mathcal{B}$ are* elementarily equivalent *(notation: $\mathcal{A} \equiv \mathcal{B}$) if they satisfy the same sentences.*

Note that if there is an elementary embedding between two structures, then they are elementarily equivalent in particular. We consider the following property.

(EEmb_w) For every $\mathcal{A} \in \mathsf{PMod}_w(\Sigma, \mathcal{T})$ there is a total model $\mathcal{B}$ of $\mathcal{T}$
and a weak embedding $\varphi : \mathcal{A} \to \mathcal{B}$
such that the embedding $\varphi : \mathcal{A}|_{\Pi_0} \to \mathcal{B}|_{\Pi_0}$ is elementary.

The definition generalizes in a natural way to a notion (EEmb_w^Ψ), parameterized by a closure term operator Ψ by requiring that the embeddability condition holds for all $\mathcal{A} \in \mathsf{PMod}_w^\Psi(\Sigma, \mathcal{T})$ with domain of definition closed under Ψ, and to corresponding finite embeddability conditions $(\mathsf{EEmb}_{w,f}^\Psi)$ analogous to $(\mathsf{Emb}_{w,f}^\Psi)$. Since every isomorphism is an elementary embedding we have the implications $(\mathsf{Comp}_w) \to (\mathsf{EEmb}_w) \to (\mathsf{Emb}_w)$ and $(\mathsf{Comp}_w^\Psi) \to (\mathsf{EEmb}_w^\Psi) \to (\mathsf{Emb}_w^\Psi)$.

A *model complete* theory is one that has the property that all embeddings between its models are elementary. So if we choose a model complete base theory then (EEmb_w) and (Emb_w) coincide. To give examples of model complete base theories note first that every theory which allows quantifier elimination (QE) is model complete (cf. [12], Theorem 7.3.1).

Example 13. The following theories have QE and are therefore model complete.

(1) Presburger arithmetic with congruence mod. n $(\equiv_n)$, $n = 2, 3, ...$ ([6], p.197).
(2) Rational linear arithmetic in the signature $\{+, 0, \leq\}$ ([27]).
(3) Real closed ordered fields ([12], 7.4.4), e.g., the real numbers.
(4) Algebraically closed fields ([5], Ex. 3.5.2; Rem. p.204; [12], Ch. 7.4, Ex. 2).
(5) Finite fields ([12], Ch. 7.4, Example 2).
(6) The theory of acyclic lists in the signature $\{\mathrm{car}, \mathrm{cdr}, \mathrm{cons}\}$ ([16,9]).

Not all model complete theories allow QE: the theory of real closed fields (without $<$) is model complete but does not admit quantifier elimination (cf. [5], 3.5.19, and the subsequent remark on p.204).

Theorem 14. *Let $\mathcal{T}_0$ be a Π_0-theory, $\Pi = (\Sigma_0 \cup \Sigma, \mathsf{Pred})$ and let $\mathcal{K}$ be a set of universally closed, linear and quasi-flat clauses in the signature Π and let $\Psi_{\mathcal{K}}$ be a term closure operator with the property that for every flat set of ground terms T, $\Psi(T)$ is flat.*

$$\begin{tikzcd}(\mathsf{Emb}^\Psi_w) \arrow[r, shift left] & (\mathsf{Loc}^\Psi) \arrow[l, shift left] \\ (\mathsf{EEmb}^\Psi_w) \arrow[u] \arrow[r, shift left] & (\mathsf{ELoc}^\Psi) \arrow[u] \arrow[l, shift left]\end{tikzcd}$$

Fig. 1. Relations between locality and embeddability

(1) If the extension $\mathcal{T}_0 \subseteq \mathcal{T}_0 \cup \mathcal{K}$ satisfies (Emb^Ψ_w) then it satisfies (Loc^Ψ).
(2) If the extension $\mathcal{T}_0 \subseteq \mathcal{T}_0 \cup \mathcal{K}$ satisfies (EEmb^Ψ_w) then it satisfies (ELoc^Ψ).

The results above give us the following relations (Fig. 1) between these properties under the conditions on Ψ and $\mathcal{K}$ in the statements of Theorem 14 and 11. The results naturally adapt to yield links between $((\mathsf{E})\mathsf{Loc}^\Psi_f)$ and $((\mathsf{E})\mathsf{Emb}^\Psi_{w,f})$.

Comments. We can generalize these results even further and refer to versions of locality resp. embeddability parameterized by a fragment $\mathcal{F}$ of the theory $\mathcal{T}_0$ (containing the ground clause fragment): In condition $(\mathsf{EEmb}^\Psi_w(\mathcal{F}))$ we require that every $\mathcal{A} \in \mathsf{PMod}^\Psi_w(\Sigma, \mathcal{T})$ weakly embeds into a total model of $\mathcal{T}$ such that the restriction of the embedding on Π_0 preserves and reflects truth of formulae in $\mathcal{F}$, possibly with parameters in $\mathcal{A}$, and by allowing in $(\mathsf{ELoc}^\Psi(\mathcal{F}))$ that all clauses in $\mathcal{K}$ and in G are $\mathcal{F}$-augmented clauses (cf. Definition 3 in Section 2). Due to space constraints, we do not present these extensions in detail here.

5 Locality Transfer Results

We present some locality transfer results.

Note. Considering an extension Π^C of a signature Π with fresh constants $C = \{c_i \mid i \in I\}$, we denote by $(\mathcal{A}, \bar{b})$ the expansion of a Π-structure $\mathcal{A}$ to a structure for the extended language Π^C where c_i is interpreted as b_i, for all i. Supposing that we have Π-structures $\mathcal{A}, \mathcal{B}$ and constants c_a for every element $a \in \mathcal{A}$, note that a map $\varphi : \mathcal{A} \to \mathcal{B}$ is an elementary embedding if and only if $(\mathcal{A}, \bar{a}) \equiv (\mathcal{B}, \varphi\bar{a})$.

We use the following theorem which can be seen as a generalization of Robinson's joint consistency theorem.

Theorem 15 ([12], 5.5.1). *Let Π_1, Π_2 be signatures, $\Pi = \Pi_1 \cap \Pi_2$, $\mathcal{B}$ a Π_1-structure, $\mathcal{C}$ a Π_2-structure and $\bar{a}$ a sequence of elements in both $\mathcal{B}$ and $\mathcal{C}$ such that $(\mathcal{B}|_\Pi,\ \bar{a}) \equiv (\mathcal{C}|_\Pi,\ \bar{a})$. Then there is a $(\Pi_1 \cup \Pi_2)$-structure $\mathcal{D}$ such that $\mathcal{B} \preccurlyeq \mathcal{D}|_{\Pi_1}$ and an elementary embedding $g : \mathcal{C} \to \mathcal{D}|_{\Pi_2}$ with $g(a_i) = a_i$ for every a_i in $\bar{a}$.*

$$\begin{tikzcd} & \mathcal{D} & \\ \mathcal{B} \arrow[ur, "\preccurlyeq"] & & \mathcal{C} \arrow[ul, "g : \preccurlyeq"'] \\ & \bar{a} \arrow[ul, "\subseteq"] \arrow[ur, "\subseteq"'] & \end{tikzcd} \quad \text{if} \quad (\mathcal{B}|_\Pi,\ \bar{a}) \equiv (\mathcal{C}|_\Pi,\ \bar{a})$$

An application of Theorem 15 is the transfer of elementary embeddings.

Theorem 16 ((EEmb) Transfer). *Let $\Pi_0 = (\Sigma_0, \mathsf{Pred})$ be a signature, $\mathcal{T}_0$ a theory in Π_0, Σ_1 and Σ_2 two disjoint sets of new function symbols, $\Pi_i := (\Sigma_0 \cup \Sigma_i, \mathsf{Pred})$, $i = 1, 2$. Assume that $\mathcal{T}_2$ is a Π_2-theory with $\mathcal{T}_0 \subseteq \mathcal{T}_2$, and $\mathcal{K}$ is a set of universally closed Π_1-clauses. If the extension $\mathcal{T}_0 \subseteq \mathcal{T}_0 \cup \mathcal{K}$ enjoys (EEmb_w) then so does the extension $\mathcal{T}_2 \subseteq \mathcal{T}_2 \cup \mathcal{K}$. In particular, if $\mathcal{K}$ is (quasi)-flat and linear then extension $\mathcal{T}_2 \subseteq \mathcal{T}_2 \cup \mathcal{K}$ satisfies condition (ELoc).*

If all variables in clauses in $\mathcal{K}$ occur below Σ_1-functions, and ground satisfiability is decidable in $\mathcal{T}_2$, then ground satisfiability is decidable in $\mathcal{T}_2 \cup \mathcal{K}$.

The result extends in a natural way to the case of finite embeddability $(\mathsf{EEmb}_{w,f})$. Theorem 16 is a very useful result, which allows us to identify a large number of local extensions. We illustrate its applicability on one example.

Example 17. Let Lat be the theory of lattices and $\mathcal{T}_1 = \mathsf{Lat} \cup \mathsf{Mon}_f$, where $\mathsf{Mon}_f = \{\forall x, y\ (x \leq y \rightarrow f(x) \leq f(y))\}$ is the monotonicity of a new function symbol f. Using techniques similar to the ones used in [24] we can prove that the extension $\mathsf{Lat} \subseteq \mathsf{Lat} \cup \mathsf{Mon}_f$ satisfies condition $(\mathsf{Comp}_{w,f})$ hence also $(\mathsf{EEmb}_{w,f})$. Let $\mathcal{T}$ be any extension of the theory of lattices (this can be the theory of distributive lattices, Heyting algebras, Boolean algebras, any theory with a total order – e.g. the (ordered) theory of integers or of reals, etc.). By Theorem 16, $\mathcal{T} \subseteq \mathcal{T} \cup \mathsf{Mon}_f$ satisfies condition $(\mathsf{EEmb}_{w,f})$, hence the extended locality condition (ELoc_f).

For model complete base theories Theorem 16 specializes as follows.

Corollary 18. *Let Π_0 be a signature, $\mathcal{T}_0$ a model complete theory in Π_0, and Σ_1 and Σ_2 two disjoint sets of new function symbols. Let $\Pi_i = (\Sigma_0 \cup \Sigma_i, \mathsf{Pred})$, $i = 1, 2$. Let $\mathcal{K}$ be a set of flat and linear Π_1-clauses and $\mathcal{T}_2$ be an arbitrary Π_2-theory with $\mathcal{T}_0 \subseteq \mathcal{T}_2$. If the extension $\mathcal{T}_0 \subseteq \mathcal{T}_0 \cup \mathcal{K}$ is local then the extension $\mathcal{T}_2 \subseteq \mathcal{T}_2 \cup \mathcal{K}$ is local as well.*

5.1 Locality and Model Completeness

A model complete theory can sometimes be regarded as the completion of another theory with the same universal fragment. Recall that the *diagram* of a first-order structure $\mathcal{A}$ is the set of all literals true in the extension $(\mathcal{A}, \bar{a})$ of $\mathcal{A}$ where we have a constant for each element of A.

Definition 19. *A theory $\mathcal{T}^*$ is called a* model completion *of $\mathcal{T}$ if (i) $\mathcal{T}$ and $\mathcal{T}^*$ are co-theories (i.e. every model of $\mathcal{T}$ can be extended to a model of $\mathcal{T}^*$ and vice versa), (ii) $\mathcal{T}^*$ is model complete and (iii) for every model $\mathcal{A}$ of $\mathcal{T}$, $\mathcal{T}^* \cup \Delta_A$ is complete where Δ_A is the diagram of $\mathcal{A}$.*

Example 20. Below we present some examples of model completions:

(1) The theory of algebraically closed fields is the model completion of the theory of fields. This was the motivating example for developing the theory of model completions ([5], Examples 3.5.2, 3.5.12; Remark 3.5.6 ff; [12], 7.3).

(2) The theory of dense total orders without endpoints is the model completion of the theory of total orders ([9]).

(3) The theory of atomless Boolean algebras is the model-completion of Boolean algebras ([5], Example 3.5.12, cf. also p.196).

(4) Universal Horn theories in finite signatures have a model completion if they are locally finite and have the amalgamation property (e.g., graphs, posets) ([28]).

Theorem 21. *Let $\mathcal{T}_0$ be a theory. Assume that $\mathcal{T}_0$ has a model completion $\mathcal{T}_0^*$ such that $\mathcal{T}_0 \subseteq \mathcal{T}_0^*$. Let $\mathcal{T} = \mathcal{T}_0 \cup \mathcal{K}$ be an extension of $\mathcal{T}_0$ with new function symbols Σ whose properties are axiomatized by a set of flat and linear clauses $\mathcal{K}$ (all of which contain symbols in Σ).*

(1) Assume that:
(i) Every model of $\mathcal{T}_0 \cup \mathcal{K}$ embeds[5] *into a model of $\mathcal{T}_0^* \cup \mathcal{K}$.*
(ii) $\mathcal{T}_0 \cup \mathcal{K}$ is a local extension of $\mathcal{T}_0$.
Then $\mathcal{T}_0^* \subseteq \mathcal{T}_0^* \cup \mathcal{K}$ satisfies condition (EEmb_w)*, hence if $\mathcal{K}$ is a set of quasi-flat and linear augmented clauses also condition* (ELoc) *as extension of $\mathcal{T}_0^*$.*

(2) If all variables in $\mathcal{K}$ occur below an extension function and $\mathcal{T}_0^* \cup \mathcal{K}$ is a local extension of $\mathcal{T}_0^*$ then $\mathcal{T}_0 \cup \mathcal{K}$ is a local extension of $\mathcal{T}_0$.

The results extend in a natural way to Ψ-locality and to finite versions of embeddability and locality.

Example 22. We show that the extension of the theory TOrd of total orderings with a strictly monotone function, i.e. a function f satisfying the axiom:

$$\mathsf{SMon}(f) \quad \forall x, y (x < y \rightarrow f(x) < f(y))$$

satisfies condition (Loc_f). To show this, note that the model completion TOrd^* of TOrd is the theory of dense total orderings without endpoints. We show that the extension $\mathsf{TOrd}^* \subseteq \mathsf{TOrd}^* \cup \mathsf{SMon}(f)$ satisfies condition (ELoc_f). Indeed, let $\mathcal{A} = (A, \leq, f)$ be a partial model of $\mathsf{TOrd}^* \cup \mathsf{SMon}(f)$ where the domain of definition of f is finite, say $\{a_1, \dots, a_n\} \subseteq A$ where $a_1 < a_2 < \dots < a_n$. W.l.o.g. we can assume that A is countable. Let $b_i = f(a_i) \in A$, $1 \leq i \leq n$. Then $b_1 < b_2 < \dots < b_n$. Let $A_0 = \{x \in A \mid x < a_1\}$, $A_i = \{x \in A \mid a_i < x < a_{i+1}\}$, for $1 \leq i \leq n-1$, and $A_n = \{x \in A \mid a_n < x\}$, and $B_0 = \{x \in A \mid x < b_1\}, B_i = \{x \in A \mid b_i < x < b_{i+1}\}$, for $1 \leq i \leq n-1$, and $B_n = \{x \in A \mid b_n < x\}$. All these sets are countable models of TOrd^* hence isomorphic (since the theory of dense total orderings without endpoints is ω-categorical). We can use these isomorphisms to extend the partial map to a strictly monotone map from A to A. This extends a result established in [15].

Similarly, we can prove that the extension of the pure theory of equality with a function f satisfying $\mathsf{Inj}(f)$ $\forall x, y\, (x \neq y \rightarrow f(x) \neq f(y))$ is local.

[5] If $\mathcal{T}_0$ is universal, this is the notion of compatibility defined in [9].

6 Combinations of Local Theories

We now identify situations in which the union of two local extensions of a common base theory is again a local extension of the base theory. This was first studied in [22] and [23]. Here, we extend some of the results in [22] and [23] by using instead of the completability of partial models the condition (EEmb_w), and also embeddability conditions parameterized by term closure operators.

6.1 Case 1: Both Theories Satisfy (EEmb_w)

We first show that extended locality is preserved when combining theories.

Lemma 23. *Let Π_0 be a signature, $\mathcal{T}_0$ a Π_0-theory, Σ_1 and Σ_2 two disjoint sets of fresh function symbols and $\mathcal{K}_i$ a set of universally closed Π_i-clauses (where $\Pi_i = (\Sigma_0 \cup \Sigma_i, \mathsf{Pred})$) for $i = 1, 2$. If both extensions $\mathcal{T}_0 \subseteq \mathcal{T}_0 \cup \mathcal{K}_i$, $i = 1, 2$, satisfy* (EEmb_w) *then so does the extension $\mathcal{T}_0 \subseteq \mathcal{T}_0 \cup \mathcal{K}_1 \cup \mathcal{K}_2$. If $\mathcal{K}_1 \cup \mathcal{K}_2$ is (quasi)-flat and linear then the extension $\mathcal{T}_0 \subseteq \mathcal{T}_0 \cup \mathcal{K}_1 \cup \mathcal{K}_2$ is local.*

Proof: This is a consequence of Theorem 15. □

If $\mathcal{T}_0$ is a model complete base theory then (EEmb_w) and (Emb_w) coincide.

Corollary 24. *Let $\mathcal{T}_0$ be a model complete Π_0-theory, Σ_1 and Σ_2 two disjoint sets of fresh function symbols and $\mathcal{K}_i$ a set of universally closed Π_i-clauses for $i = 1, 2$. If both extensions $\mathcal{T}_0 \subseteq \mathcal{T}_0 \cup \mathcal{K}_i$, $i = 1, 2$, satisfy* (Emb_w) *then so does the extension $\mathcal{T}_0 \subseteq \mathcal{T}_0 \cup \mathcal{K}_1 \cup \mathcal{K}_2$. In particular, if $\mathcal{K}_1 \cup \mathcal{K}_2$ is (quasi)-flat and linear then the extension $\mathcal{T}_0 \subseteq \mathcal{T}_0 \cup \mathcal{K}_1 \cup \mathcal{K}_2$ is local.*

If the base theory $\mathcal{T}_0$ is complete, (EEmb_w) follows from (Emb_w). Recall that a theory $\mathcal{T}$ is complete if it is consistent and it holds for every sentence φ that $\mathcal{T} \models \varphi$ or $\mathcal{T} \models \neg\varphi$. Equivalently, a theory $\mathcal{T}$ is complete if it is consistent and all its models are elementarily equivalent.

Corollary 25. *Let $\mathcal{T}_0$ be a complete theory in Σ_0, Σ_1 and Σ_2 two disjoint sets of fresh function symbols and $\mathcal{K}_i$ a set of clauses in $\Sigma_0 \cup \Sigma_i$ for $i = 1, 2$. If both extensions $\mathcal{T}_0 \subseteq \mathcal{T}_0 \cup \mathcal{K}_i$, $i = 1, 2$, satisfy* (Emb_w) *then so does the extension $\mathcal{T}_0 \subseteq \mathcal{T}_0 \cup \mathcal{K}_1 \cup \mathcal{K}_2$. In particular, if $\mathcal{K}_1 \cup \mathcal{K}_2$ is (quasi)-flat and linear then the extension $\mathcal{T}_0 \subseteq \mathcal{T}_0 \cup \mathcal{K}_1 \cup \mathcal{K}_2$ is local.*

Example 26. The following theories are complete.
(1) Presburger arithmetic ([20]).
(2) Real closed fields ([12], Thm. 2.7.2 and subsequent Remark (b).).
(3) The theory of algebraically closed fields of characteristic 0 or p (p prime) ([5], Prop. 1.4.10ff.; Example 3.5.9 and page 197).
(4) Divisible torsion-free Abelian groups in which all elements have order p (p prime) (ibid.).

Remark 27. Completeness and model completeness do not imply one another. For instance, algebraically closed fields are model complete but not complete (cf. [5], p.188). On the other hand, dense linear orders with endpoints are complete but not model complete (cf. [5], p.187).

6.2 Case 2: One Theory Satisfies (EEmb_w)

We consider combinations of theory extensions among which one satisfies condition (EEmb_w). We extend Theorem 19 of [23] to handle this situation.

Theorem 28. *Let $\mathcal{T}_0$ be a theory in the signature Π_0, Σ_1 and Σ_2 two disjoint sets of new function symbols, and $\Pi_i = (\Sigma_0 \cup \Sigma_i, \mathsf{Pred})$. Let $\mathcal{K}_i$ be a set of Π_i-clauses for $i = 1, 2$, and $\mathcal{T}_i := \mathcal{T}_0 \cup \mathcal{K}_i$, $i = 1, 2$. Assume that:*

(1) $\mathcal{T}_0 \subseteq \mathcal{T}_1$ satisfies (EEmb_w),
(2) $\mathcal{T}_0 \subseteq \mathcal{T}_2$ satisfies (Emb_w) and
(3) $\mathcal{K}_1$ is Σ_1-flat in which all variables are shielded, i.e., all variables occur below some Σ_1-function.

Then the extension $\mathcal{T}_0 \subseteq \mathcal{T}_0 \cup \mathcal{K}_1 \cup \mathcal{K}_2$ satisfies (Emb_w). If $\mathcal{K}_1 \cup \mathcal{K}_2$ is (quasi)-flat and linear then the extension $\mathcal{T}_0 \subseteq \mathcal{T}_0 \cup \mathcal{K}_1 \cup \mathcal{K}_2$ is local.

6.3 Combinations of Ψ-Local Theory Extensions

We now study combinations of Ψ_i-local extensions (with different Ψ_i's) over a common base theory. For a partial algebra $\mathcal{A}$ and a term closure operator Ψ, let us write $\Psi_{\mathcal{K}}(\mathcal{A})$ for the set $\Psi_{\mathcal{K}}(\mathcal{D}(\mathcal{A}))$ in what follows. The following lemma lifts the argument in [23] (cf. also [22]) to Ψ-locality.

Lemma 29. *Let $\mathcal{T}_0$ be a Π_0-theory, Σ_1 and Σ_2 two disjoint sets of new function symbols, $\Pi_i = (\Sigma_0 \cup \Sigma_i, \mathsf{Pred})$, and $\mathcal{K}_i$ a set of universally closed Π_i-clauses, for $i = 1, 2$. Let $\mathcal{T}_i = \mathcal{T}_0 \cup \mathcal{K}_i$, $i = 1, 2$. Let $\Psi_{\mathcal{K}_1}$ be a closure operator w.r.t. (Π_1^C)-terms. Let $\mathcal{A}$ be a $(\Pi_1 \cup \Pi_2)$-structure such that $\mathcal{A}|_{\Pi_1} \in \mathsf{PMod}_w(\Sigma, \mathcal{T})$, $\mathcal{B}$ be a total model of $\mathcal{T}_2$ and $\chi : \mathcal{A}|_{\Pi_2} \rightarrow \mathcal{B}$ be a weak Π_2-embedding. Assume that:*

(1) $\mathcal{K}_1$ is Σ_1-flat,
(2) all variables of $\mathcal{K}_1$ appear below an extension function,
(3) all terms in $\Psi_{\mathcal{K}_1}(\mathcal{A}|_{\Pi_1})$ are defined in $(\mathcal{A}, \{a \mid a \in A\})$.

Then χ and $\mathcal{B}$ can be extended s.t. $\hat{\chi}{:}\mathcal{A} {\rightarrow} \hat{\mathcal{B}}$ is a weak $\Pi_1 {\cup} \Pi_2$-embedding, $\hat{\mathcal{B}}|_{\Pi_2} = B|_{\Pi_2} {\models} \mathcal{T}_2$, $\hat{\mathcal{B}}|_{\Pi_1} {\in} \mathsf{PMod}_w(\Sigma, \mathcal{T})$, and all terms of $\Psi_{\mathcal{K}_1}(\mathcal{B}|_{\Pi_1})$ are defined in $\mathcal{B}$.

Theorem 30. *Let $\mathcal{T}_0$ be a theory in the signature Π_0, Σ_1 and Σ_2 two disjoint sets of fresh function symbols. Let Π_i defined as above, and let $\mathcal{K}_i$ be a set of universally closed Π_i-clauses for $i = 1, 2$. Let $\mathcal{T}_i := \mathcal{T}_0 \cup \mathcal{K}_i$, $i = 1, 2$. Let Ψ_i be term closure operators on ground Π_i^C-terms, $i = 1, 2$. Suppose that*

(1) $\mathcal{T}_0$ is a $\forall\exists$ theory,
(2) $\mathcal{K}_i$ is Σ_i-flat and $\mathcal{T}_0 \subseteq \mathcal{T}_i$ satisfies condition $(\mathsf{Emb}_w^{\Psi_i})$ for $i = 1, 2$,
(3) all variables are shielded in $\mathcal{K}_i$, i.e., all variables occur below an extension function, $i = 1, 2$.

Then $\mathcal{T}_0 \subseteq \mathcal{T}_0 \cup \mathcal{K}_1 \cup \mathcal{K}_2$ has $(\mathsf{Emb}_w^{\Psi_1 \cup \Psi_2})$ where $(\Psi_1 \cup \Psi_2)(\Gamma) := \Psi_1(\Gamma) \cup \Psi_2(\Gamma)$.

Proof: Let $\mathcal{A}$ be a partial model of $\mathcal{T}_0 \cup \mathcal{K}_1 \cup \mathcal{K}_2$ with total Σ_0-functions such that the terms in $(\Psi_1 \cup \Psi_2)(\mathcal{A})$ are all defined. We need to embed $\mathcal{A}$ into a total model of $\mathcal{T}_0 \cup \mathcal{K}_1 \cup \mathcal{K}_2$. We build up this total model inductively by repeatingly using Lemma 29 and embeddability to get a diagram

$$\begin{tikzcd}
 & \overset{\mathcal{T}_1,\mathcal{T}_2^w}{\mathcal{B}_1} \arrow[dr] & & \overset{\mathcal{T}_1,\mathcal{T}_2^w}{\mathcal{B}_3} \arrow[dr] & & \cdots \\
\underset{\mathcal{T}_1^w,\mathcal{T}_2^w}{\mathcal{A}} \arrow[ur] & & \underset{\mathcal{T}_1^w,\mathcal{T}_2}{\mathcal{B}_2} \arrow[ur] & & \underset{\mathcal{T}_1^w,\mathcal{T}_2}{\mathcal{B}_4} \arrow[ur] &
\end{tikzcd}$$

where all the arrows are weak $(\Pi_1 \cup \Pi_2)$-inclusions, $\mathcal{B}_{2k} \models \mathcal{T}_2$, $\mathcal{B}_{2k} \models_w \mathcal{T}_1$ and $\mathcal{B}_{2k+1} \models \mathcal{T}_1$, $\mathcal{B}_{2k+1} \models_w \mathcal{T}_2$ We can construct the inductive limit $\mathcal{B}_\omega$ of this chain, having as carrier the union of the carriers B_i, and the functions defined by:

$$f_{\mathcal{B}_\omega}(\bar{b}) := f_{\mathcal{B}_i}(\bar{b}) \quad \text{if } \exists i \text{ s. t. } f_{\mathcal{B}_i}(\bar{b}) \text{ is defined.}$$

The functions are total and well-defined by the definition of $\mathcal{B}_\omega$ and because all maps in the diagram are weak embeddings. We obtain the total $\Pi_1 \cup \Pi_2$-structure $\mathcal{B}_\omega$. Since $\mathcal{T}_0$ is $\forall\exists$, $\mathcal{B}_\omega$ is a model of $\mathcal{T}_0$ (Chang-Łos-Suszko theorem). It is easy to check that $\mathcal{B}_\omega$ is a total model of $\mathcal{T}_0 \cup \mathcal{K}_1 \cup \mathcal{K}_2$. □

Corollary 31. *With the above notations, additionally assume that the $\mathcal{K}_i$ are Σ_i-linear for $i = 1, 2$. Then for any closure term operator Ψ_3 with $\Psi_3 \supseteq (\Psi_1 \cup \Psi_2)$ it holds that $\mathcal{T}_0 \subseteq \mathcal{T}_0 \cup \mathcal{K}_1 \cup \mathcal{K}_2$ is a Ψ_3-local extension. Hence, under conditions (1),(3) if $\mathcal{K}_i$ are flat and linear and $\mathcal{T}_0 \subseteq \mathcal{T}_0 \cup \mathcal{K}_i$ satisfies $(\mathsf{Emb}_w^{\Psi_i})$, for $i = 1, 2$ then $\mathcal{T}_0 \subseteq \mathcal{T}_0 \cup \mathcal{K}_1 \cup \mathcal{K}_2$ satisfies $(\mathsf{Emb}_w^{\Psi_3})$ for every Ψ_3 with $\Psi_3 \supseteq (\Psi_1 \cup \Psi_2)$.*

We now analyze the locality of combinations of local extensions of a base theory which share some of the extension clauses (a consequence of Theorem 30).

Theorem 32. *Let $\mathcal{T}_0$ be a theory with signature $\Pi_0 = (\Sigma_0, \mathsf{Pred})$. Let $\mathcal{K}$ be a set of universally closed, flat and linear clauses in the signature $(\Sigma_0 \cup \Sigma, \mathsf{Pred})$; and let Σ_1, Σ_2 be sets of new function symbols such that $\Sigma_1 \cap \Sigma_2 = \emptyset$ and $\Sigma \cap \Sigma_i = \emptyset$, $i = 1, 2$. Let $\mathcal{K}_i, i = 1, 2$, be sets of universally closed, flat and linear $(\Sigma_0 \cup \Sigma \cup \Sigma_i, \mathsf{Pred})$-clauses such that each clause in $\mathcal{K}_i$ contains at least one Σ_i-symbol. Assume that $\mathcal{T}_0 \subseteq \mathcal{T}_0 \cup \mathcal{K} \cup \mathcal{K}_i$ is local, for $i = 1, 2$. Then:*

(1) The extension $\mathcal{T}_0 \subseteq \mathcal{T}_0 \cup \mathcal{K}$ is local.
(2) The extension $\mathcal{T}_0 \cup \mathcal{K} \subseteq \mathcal{T}_0 \cup \mathcal{K} \cup \mathcal{K}_i$ is local.
(3) If $\mathcal{T}_0$ is a $\forall\exists$-theory and all variables are shielded in $\mathcal{K}_i$, $i = 1, 2$, then the extension $\mathcal{T}_0 \subseteq \mathcal{T}_0 \cup \mathcal{K} \cup \mathcal{K}_1 \cup \mathcal{K}_2$ is local.
(4) The extension $\mathcal{T}_0 \cup \mathcal{K} \cup \mathcal{K}_1 \subseteq \mathcal{T}_0 \cup \mathcal{K} \cup \mathcal{K}_1 \cup \mathcal{K}_2$ is local.

Implementation. These results were established with the goal of having simple, modular ways of recognizing locality and Ψ-locality and for giving and implementing efficient decision procedures for theory extensions and combinations. At the moment H-PILoT [13] handles combinations of local theories as follows:

By Theorems 30 and 32, a theory extension $\mathcal{T}_0 \subseteq \mathcal{T}_0 \cup \mathcal{K}_1 \cup \cdots \cup \mathcal{K}_n$, where the clauses $\mathcal{K}_i$ specify function symbols in mutually disjoint signatures Σ_i and where all variables appear below an extension function, is local and can equivalently be considered as a chain of local extensions $\mathcal{T}_0 \subseteq \mathcal{T}_0 \cup \mathcal{K}_1 \subseteq \cdots \subseteq \mathcal{T}_0 \cup \bigcup_{i=1}^{n} \mathcal{K}_i$, provided each extension $\mathcal{T}_0 \subseteq \mathcal{T}_0 \cup \mathcal{K}_i$ is local.

7 Conclusions

In this paper we gave semantical characterizations of locality conditions parameterized by closure operators on ground terms. These operators capture in a theoretical way the type of instances of the axioms which are needed for guaranteeing completeness for ground satisfiability problems in extensions of a theory with sets of clauses. The conditions we imposed on the closure operators we consider allow us to address within the framework of Ψ-locality a large number of theories related to data structures and some theories which occur e.g. in relationship with description logics. Based on this, we identified several situations in which locality results can be transferred from one theory extension to another, some of them with a model theoretical flavor. We then studied possibilities of combining local theory extensions. The results we obtained allow us to identify in a simple and structured way an even larger number of local theory extensions interesting for applications. These theoretical results have been used for extending the H-PILoT prover.

Acknowledgments. We thank the referees for their helpful comments. This work was partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center “Automatic Verification and Analysis of Complex Systems” (SFB/TR 14 AVACS). See `www.avacs.org` for more information.

References

1. Baader, F., Tinelli, C.: Deciding the word problem in the union of equational theories. Information and Computation 178(2), 346–390 (2002)
2. Basin, D.A., Ganzinger, H.: Automated complexity analysis based on ordered resolution. Journal of the ACM 48(1), 70–109 (2001)
3. Bradley, A.R., Manna, Z., Sipma, H.: What's decidable about arrays? In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 427–442. Springer, Heidelberg (2005)
4. Burris, S.: Polynomial time uniform word problems. Mathematical Logic Quarterly 41, 173–182 (1995)
5. Chang, C.C., Keisler, J.J.: Model Theory. North-Holland, Amsterdam (1990)
6. Enderton, H.B.: A Mathematical Introduction to Logic, 2nd edn. Harcourt Academic Press, London (2002)
7. Ganzinger, H.: Relating semantic and proof-theoretic concepts for polynomial time decidability of uniform word problems. In: Logic in Computer Science, LICS'01, pp. 81–92. IEEE Computer Society Press, Los Alamitos (2001)
8. Ganzinger, H., Sofronie-Stokkermans, V., Waldmann, U.: Modular proof systems for partial functions with weak equality. In: Basin, D., Rusinowitch, M. (eds.) IJCAR 2004. LNCS (LNAI), vol. 3097, pp. 168–182. Springer, Heidelberg (2004)

9. Ghilardi, S.: Model-theoretic methods in combined constraint satisfiability. Journal of Automated Reasoning 33(3-4), 221–249 (2004)
10. Givan, R., McAllester, D.A.: New results on local inference relations. In: Nebel, B., Rich, C., Swartout, W.R. (eds.) Knowledge Representation and Reasoning, KR'92, pp. 403–412 (1992)
11. Givan, R., McAllester, D.A.: Polynomial-time computation via local inference relations. ACM Transactions on Comp. Logic 3(4), 521–541 (2002)
12. Hodges, W.: A Shorter Model Theory. Cambridge University Press, Cambridge (1997)
13. Ihlemann, C., Sofronie-Stokkermans, V.: System description: H-PILoT. In: Schmidt, R.A. (ed.) CADE 2009. LNCS (LNAI), vol. 5663, pp. 131–139. Springer, Heidelberg (2009)
14. Ihlemann, C., Jacobs, S., Sofronie-Stokkermans, V.: On local reasoning in verification. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 265–281. Springer, Heidelberg (2008)
15. Jacobs, S.: Hierarchic Decision Procedures for Verification. PhD. Thesis, Univ. des Saarlandes (2010)
16. Mal'cev, A.I.: Axiomatizable classes of locally free algebras of various types. The Metamathematics of Algebraic Systems. In: Collected Papers: 1936-1967, Studies in Logic and the Foundation of Mathematics, ch. 23, vol. 66. North-Holland, Amsterdam (1971)
17. McAllester, D.A.: Automatic recognition of tractability in inference relations. Journal of the ACM 40(2), 284–303 (1993)
18. McPeak, S., Necula, G.C.: Data structure specifications via local equality axioms. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 476–490. Springer, Heidelberg (2005)
19. Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. ACM Transactions on Programming Languages and Systems 1(2), 245–257 (1979)
20. Presburger, M.: Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. Comptes Rendus du Premier Congrès des Mathématiciens des Pays Slaves, 92–101 (1929)
21. Sofronie-Stokkermans, V.: Hierarchic reasoning in local theory extensions. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 219–234. Springer, Heidelberg (2005)
22. Sofronie-Stokkermans, V.: On combinations of local theory extensions. In: Proceedings of the Workshop on Programming Logics in memory of Harald Ganzinger, WPLHG'05 (to appear, 2005), http://arxiv.org/abs/0810.2653
23. Sofronie-Stokkermans, V.: Hierarchical and modular reasoning in complex theories: The case of local theory extensions. In: Konev, B., Wolter, F. (eds.) FroCos 2007. LNCS (LNAI), vol. 4720, pp. 47–71. Springer, Heidelberg (2007)
24. Sofronie-Stokkermans, V., Ihlemann, C.: Automated reasoning in some local extensions of ordered structures. Journal of Multiple-Valued Logics and Soft Computing 13(4-6), 397–414 (2007)
25. Tinelli, C., Ringeissen, C.: Unions of non-disjoint theories and combinations of satisfiability procedures. Theoretical Computer Science 290(1), 291–353 (2003)
26. Tinelli, C., Zarba, C.: Combining non-stably infinite theories. Journal of Automated Reasoning 34(3), 209–238 (2005)
27. Weispfenning, V.: The complexity of linear problems in fields. Journal of Symbolic Computation 5(1/2), 3–27 (1988)
28. Wheeler, W.H.: Model-companions and definability in existentially complete structures. Israel Journal of Mathematics 25, 305–330 (1976)

Global Caching for Coalgebraic Description Logics[⋆]

Rajeev Goré[1], Clemens Kupke[2], Dirk Pattinson[2], and Lutz Schröder[3]

[1] Computer Science Laboratory, The Australian National University
[2] Department of Computing, Imperial College London
[3] DFKI Bremen and Department of Computer Science, Universität Bremen

Abstract. Coalgebraic description logics offer a common semantic umbrella for extensions of description logics with reasoning principles outside relational semantics, e.g. quantitative uncertainty, non-monotonic conditionals, or coalitional power. Specifically, we work in coalgebraic logic with global assumptions (i.e. a general TBox), nominals, and satisfaction operators, and prove soundness and completeness of an associated tableau algorithm of optimal complexity EXPTIME. The algorithm uses the (known) tableau rules for the underlying modal logics, and is based on on global caching, which raises hopes of practically feasible implementation. Instantiation of this result to concrete logics yields new algorithms in all cases including standard relational hybrid logic.

Introduction

Description Logics (DLs) [2], which may be regarded as notational variants of various extensions of modal logic, constitute one of the most important formalisms in the area of logic-based knowledge representation. Two key features of DLs are support for *nominals*, i.e. the ability to reason about particular individual states of the model rather than about subsets only, and support for reasoning using global assumptions, a so-called (general) TBox. The combination of these two features corresponds roughly to (i.e. is slightly less expressive than) reasoning with global assumptions in hybrid logic [1].

On the other hand, reasoning about real-life problems typically calls for modal principles beyond the standard existential and universal constraints that correspond in DL notation to the boxes and diamonds of modal logic. Some of these principles, such as qualified number constraints, are already incorporated in many DLs, while support for others such as quantitative uncertainty ('with likelihood at least p'), non-monotonic conditionals ('if a then normally b'), and strategic aspects ('coalition C of agents can force a') is less well-developed. In particular, the complexity of hybrid or description logics including these features was largely unknown until it was shown recently that TBox reasoning in such logics is, under weak assumptions, in EXPTIME and, hence, typically EXPTIME complete [20] The generic framework that makes results at this level of generality possible is *coalgebraic logic*, which relies on the principle of encapsulating the type of systems underlying the semantics of a given modal or hybrid logics (neighbourhood frames, Kripke frames, Markov chains, game frames etc.) as coalgebras for a set functor.

[⋆] The second and third author supported by EPSRC grant EP/F031173/1. The last author supported by BMBF grant FormalSafe (FKZ 01IW07002).

From a practical point of view, the generic algorithm presented in [20] has two drawbacks: it relies on decision procedures for infinite games, which guarantee EXPTIME complexity but also induce average-case exponential run time; and moreover it starts by guessing the theories of named states, which is practically infeasible. In the present work, we drastically improve on this by presenting an EXPTIME tableau algorithm that uses *global caching*. Tableau-based methods have been successful in providing efficient decision procedures for modal and description logics [12], and in particular are employed in the leading current DL reasoners. They are relatively easy to implement but often do not meet known upper complexity bounds for a given logic. For example, the traditional tableau algorithm for the $\mathcal{ALC}$ requires double exponential time in the worst case, and known alternative EXPTIME tableaux are highly complex [8]. Global caching has recently been proposed as a way of establishing tableau-based decision procedures that do meet known optimality bounds, while at the same time offering good practical efficiency and room for heuristic optimization. It has been successfully applied to the description logics $\mathcal{ALC}$ and $\mathcal{ALCI}$ [10,11] and, more recently, to coalgebraic modal logic with global assumptions [9].

The fundamental idea of global caching is that tableau sequents should be arranged in a directed graph rather than in a tree. In this way one is able to avoid unnecessary repetitions of calculations that would normally happen on various branches of a tree-shaped tableau. The challenge in extending global caching to hybrid logics is to manage the theories of named nodes in a consistent way across the tableau without resorting to backtracking, which is what global caching tries to avoid. Our algorithm achieves precisely this. It instantiates to new tableau-based decision procedures for a wide range of hybrid logics such as probabilistic and graded hybrid logics and hybrid coalition logic. Even its incarnation in the traditional relational realm seems to be a new decision procedure for global reasoning in hybrid K (or $\mathcal{ALCO}$ extended with satisfaction operators; see [4,3] for an overview of existing tableau systems for hybrid K), to our knowledge the first backtracking-free tableau-based procedure for hybrid K that matches the known EXPTIME bound [1]. Compared to other approaches in the literature [4] we note that our tableaux are *unlabelled* which allows us to avoid backtracking, as we do not need to pay special attention to labels, and in turn entails the optimal complexity bound.

1 Global Caching for Hybrid Logics, Informally

We proceed to give a brief motivation of the main ideas of our algorithm, using two very simple examples; the full description of the algorithm is given in Section 4. We feel entitled to work in the standard example of hybrid logic over Kripke frames, as our algorithm appears to be new even in this basic case, but we emphasize that our method applies in the much wider setting of coalgebraic hybrid logic. Recall that the main features that distinguish hybrid logic from plain modal logic (which comes with an operator $\Diamond$ 'there exists a successor state such that', corresponding to existential restrictions in description logic) are *nominals* which designate individual states, and *satisfaction operators* $@_i$ 'state i satisfies ...'. The main problem in designing tableaux calculi for hybrid logics that are amenable to global caching is that the satisfaction operators are global in nature, i.e. all states in the tableau have to agree on the truth values of formulas of the form $@_i A$. In complexity proofs, it is unproblematic to just guess these truth values before building the tableau [14,20], but of course this is not a feasible approach in the design of a reasoning algorithm that aims for efficient average-case behaviour. Standard tableau algorithms for hybrid logic (e.g. [3]) explore possible truth values for @-formulas via backtracking; however, the driving idea behind the global caching approach is precisely to minimize backtracking.

Our solution to this problem is roughly as follows. We view the construction of a tableau in the standard way as a game between two players, Eloise ($\exists$) and Abelard ($\forall$), where $\exists$ tries to prove satisfiability of the target formulas, and $\forall$ unsatisfiability. A typical move by $\exists$ in a standard purely modal tableau would be, e.g., to select a disjunct from a disjunction, while a typical move by $\forall$ would be, e.g., to pick a diamond formula $\Diamond A$ for which $\exists$ then has to establish satisfiability of a new state labelled A. In the hybrid setting, $\exists$ may stumble upon formulas $@_i\psi$ in this process; she may choose to just ignore these for the moment, but will later be forced to prove satisfiability of all such formulas that she runs across. Our main idea is now to collect such @-formulas along a potential winning strategy of $\exists$ and propagate this collection back through the tableau. Collections of @-formulas, which we call @-constraints, are attached to standard nodes via special links, along (one of) which $\exists$ may be forced to move to prove satisfiability.

As our first example, we have the following tableau for the sequent $\Diamond @_i p, \Diamond(\Diamond @_i q \vee \Diamond \neg p)$ (with the comma read conjunctively).

$$\begin{tikzcd}
\boxed{@_i p, @_i \neg p} & & \Diamond @_i p, \Diamond(\Diamond @_i q \vee \Diamond @_i \neg p) \arrow[ll, dotted, "\exists"'] \arrow[rr, dotted, "\exists"] \arrow[dl, "\forall"'] \arrow[d, "\forall"] & & \boxed{@_i p, @_i q} \arrow[d, "\forall"] \\
\boxed{@_i p} & \Diamond @_i p \arrow[l, dotted, "\exists"'] \arrow[d, "\forall"] & \Diamond @_i q \vee \Diamond @_i \neg p \arrow[r, dotted, "\exists"] \arrow[dr, dotted, "\exists"] \arrow[ddl, "\exists"'] \arrow[ddr, "\exists"] & \boxed{@_i q} & i, p, q, @_i p, @_i q \arrow[u, dotted, "\exists"] \\
\boxed{@_i p} & @_i p \arrow[l, dotted, "\exists"'] & & \boxed{@_i \neg p} & \\
\boxed{@_i q} & \Diamond @_i q \arrow[l, dotted, "\exists"'] \arrow[d, "\forall"] & & \Diamond @_i \neg p \arrow[r, dotted, "\exists"] \arrow[d, "\forall"'] & \boxed{@_i \neg p} \\
\boxed{@_i q} & @_i q \arrow[l, dotted, "\exists"'] & & @_i \neg p \arrow[r, dotted, "\exists"] & \boxed{@_i \neg p}
\end{tikzcd}$$

The solid arrows above indicate unfolding of tableau rules. At the root node, $\forall$ has two challenges, corresponding to the two $\Diamond$-formulas, that $\exists$ can answer in each case, and she can choose between two disjuncts at the node below the root. Ignoring @-constraints, both choices would demonstrate satisfiability of the root. The @-constraints are represented by dotted lines above, and are propagated from the bottom nodes that contain @-formulas. At the node below the root, $\exists$ can demonstrate satisfiability if she can satisfy *either* of the two constraints, which are propagated to the root node, and induce (stronger) constraints that also account for the the second (left) $\Diamond$-formula. At the root, each constraint represents a set of assumptions that need to be met in order for $\exists$ to demonstrate satisfiability, and the ability to establish satisfiability of the constraint on the right finally gives satisfiability of the root node. Note that we could have declared satisfiability even without unfolding the left disjunct at the node below the root, as the satisfiability of a node hinges on the satisfiability of only *one* constraint. The situation is different in the next example, a tableau for the sequent $\Diamond @_i\bot, \Diamond A$, where A is some complex formula:

$$\begin{tikzcd}
 & \Diamond @_i\bot, \Diamond A \arrow[r, dotted, "\exists"] \arrow[d, "\forall"] & \boxed{@_i\bot, \bullet} \arrow[d, "\forall"] \\
\boxed{@_i\bot} & @_i\bot \arrow[l, dotted, "\exists"'] & i, \bot, @_i\bot
\end{tikzcd}$$

Here, the root formula is clearly unsatisfiable, and $\forall$ can challenge the satisfiability of the root by requiring successors for each of the $\Diamond$-formulas. Unfolding the left $\Diamond$, this induces an unsatisfiable @-constraint that is propagated to the root, but as the right $\Diamond$ is not yet unfolded, this constraint is incomplete, indicated by $\bullet$ – further unfolding and propagation would still require satisfiability of $@_i\bot$. Challenging the satisfiability of this (incomplete) constraint, $\forall$ demonstrates unsatisfiability of the root node, even without fully unfolding the tableau.

2 Syntax and Semantics of Coalgebraic Hybrid Logic

To make our treatment parametric in the concrete syntax of any given modal logic, we fix a modal similarity type Λ consisting of modal operators with associated arities throughout. Throughout the paper, Prop and N denote two denumerable disjoint sets of *propositional variables* and *nominals*, respectively. We will only be considering formulas in negation normal form, and abbreviate $\overline{\mathsf{P}} = \{\overline{p} \mid p \in \mathsf{P}\}$ and $\overline{\Lambda} = \{\overline{\heartsuit} \mid \heartsuit \in \Lambda\}$ where the derived dual modal operator $\overline{\heartsuit}$ has the same arity as $\heartsuit$. The set $\mathcal{H}(\Lambda)$ of *hybrid Λ-formulas* is given by the grammar

$$\mathcal{H}(\Lambda) \ni A, B ::= x \mid A \wedge B \mid A \vee B \mid \heartsuit(A_1, \dots, A_n) \mid \overline{\heartsuit}(A_1, \dots, A_n) \mid @_i A$$

where $x \in \mathsf{P} \cup \overline{\mathsf{P}} \cup \mathsf{N} \cup \overline{\mathsf{N}}$ is a possibly negated propositional variable or nominal, $\heartsuit \in \Lambda$ is n-ary and $i \in \mathsf{N}$ is a nominal. We write

$$(\Lambda \cup \overline{\Lambda})(F) = \{\heartsuit(A_1, \dots, A_n) \mid \heartsuit \in \Lambda \cup \overline{\Lambda}\ n\text{-ary}, A_1, \dots, A_n \in F\}$$

for the set of all formulas that can be constructed by applying a (possibly dualised) modal operator to elements of a set F of formulas. A *Λ-tableau-sequent*, short *Λ-sequent* or just *sequent*, is a finite set of Λ-formulas that we read conjunctively, and we write $\mathcal{S}(\Lambda)$ for the set of Λ-sequents. A *Λ-state* is a Λ-sequent that neither contain a top-level propositional connective nor a pair $x, \overline{x}$ of complementary propositional variables or nominals, and we write $\mathsf{State}(\Lambda)$ for the set of Λ-states. We define the negation $\overline{A}$ of a formula $A \in \mathcal{H}(\Lambda)$ by $\overline{\overline{p}} = p$, $\overline{(A \wedge B)} = \overline{A} \vee \overline{B}$, $\overline{A \vee B} = \overline{A} \wedge \overline{B}$, $\overline{\heartsuit(A_1, \dots A_n)} = \overline{\heartsuit}(\overline{A_1}, \dots, \overline{A_n})$, $\overline{\overline{\heartsuit}(A_1, \dots, A_n)} = \heartsuit(\overline{A_1}, \dots, \overline{A_n})$ and $\overline{@_i A} = @_i \overline{A}$. We use the standard definitions for the other propositional connectives $\rightarrow, \leftrightarrow, \vee$. The set of nominals occurring in a formula A is denoted by $\mathsf{N}(A)$. This extends to sets of formulas, and $\mathsf{N}(\Gamma) = \bigcup\{\mathsf{N}(A) \mid A \in \Gamma\}$. A formula of the form

$@_i A$ is called an *@-formula*, and a formula that does not begin with @ is called *plain*. Semantically, nominals i denote individual states in a model, and an @-formula $@_i A$ stipulates that A holds at i.

To reflect parametricity in the particular underlying logic also semantically, we equip hybrid logics with a *coalgebraic semantics* extending the standard coalgebraic semantics of modal logics [15]: we fix an endofunctor $T : \mathsf{Set} \to \mathsf{Set}$ throughout that is equipped with an assigment of an *n-ary predicate lifting* $[\![\heartsuit]\!]$ for every n-ary modal operator $\heartsuit \in \Lambda$, i.e. a set-indexed family of mappings $([\![\heartsuit]\!]_X : \mathcal{P}(X)^n \to \mathcal{P}(TX))_{X \in \mathsf{Set}}$ that satisfies

$$[\![\heartsuit]\!]_X \circ (f^{-1})^n = (Tf)^{-1} \circ [\![\heartsuit]\!]_Y$$

for all $f : X \to Y$. In categorical terms, $[\![\heartsuit]\!]$ is a natural transformation $\mathcal{Q}^n \to \mathcal{Q} \circ T^{op}$ where $\mathcal{Q} : \mathsf{Set}^{op} \to \mathsf{Set}$ is the contravariant powerset functor.

In this setting, T-coalgebras play the roles of *frames*. A *T-coalgebra* is a pair (C, γ) where C is a set of *states* and $\gamma : C \to TC$ is the *transition function*. If clear from the context, we identify a T-coalgebra (C, γ) with its state space C. A *(hybrid) T-model* (C, γ, π) consists of a T-coalgebra (C, γ) together with a *hybrid valuation* π, i.e. a map $\mathsf{P} \cup \mathsf{N} \to \mathcal{P}(C)$ that assigns singleton sets to all nominals $i \in \mathsf{N}$. We often identify the singleton set $\pi(i)$ with its unique element.

The semantics of $\mathcal{H}(\Lambda)$ is a satisfaction relation $\models$ between states $c \in C$ in hybrid T-models $M = (C, \gamma, \pi)$ and formulas $A \in \mathcal{H}(\Lambda)$, inductively defined as follows. For $x \in \mathsf{N} \cup \mathsf{P}$ and $i \in \mathsf{N}$, put

$$c, M \models x \text{ iff } c \in \pi(x) \qquad c, M \models @_i A \text{ iff } \pi(i), M \models A.$$

Modal operators are interpreted using their associated predicate liftings, that is,

$$c, M \models \heartsuit(A_1, \dots, A_n) \iff \gamma(c) \in [\![\heartsuit]\!]_C([\![A_1]\!]_M, \dots, [\![A_n]\!]_M)$$

where $\heartsuit \in \Lambda$ n-ary and $[\![A]\!]_M = \{c \in C \mid M, c \models A\}$ denotes the truth-set of A relative to M and $[\![\overline{\heartsuit}]\!]_C(A_1, \dots, A_n) = C \setminus [\![\heartsuit]\!](C \setminus A_1, \dots, C \setminus A_n)$ for the case of dual operators. If Ξ is a set of formulas (the global assumptions, or TBox), we write $\mathsf{Mod}(\Xi)$ for the class of models $M = (C, \gamma, \pi)$ such that $M, c \models A$ for all $A \in \Xi$ and all $c \in C$. A formula A is *satisfiable in* $\mathsf{Mod}(\Xi)$ if it is satisfied in some state in some model of $\mathsf{Mod}(\Xi)$.

The distinguishing feature of the coalgebraic approach to hybrid and modal logics is the parametricity in both the logical language and the notion of frame: concrete instantiations of the general framework, in other words a choice of modal operators Λ, an endofunctor T and an assigment of predicate liftings, capture the semantics of a wide range of modal logics, as witnessed by the following examples.

Example 1. 1. The hybrid version of the modal logic K, *hybrid K* for short, has a single unary modal operator $\Box$, and we write $\overline{\Box} = \Diamond$. Hybrid K is interpreted over coalgebras for the powerset functor $\mathcal{P}$ that takes a set X to its powerset $\mathcal{P}(X)$ and $[\![\Box]\!]_X(A) = \{B \in \mathcal{P}(X) \mid B \subseteq A\}$. It is clear that $\mathcal{P}$-coalgebras $(C, \gamma : C \to \mathcal{P}(C))$ are in 1-1 correspondence with Kripke frames, and that the coalgebraic definition of satisfaction specialises to the usual semantics of the box operator.

2. *Graded hybrid logic* has modal operators $\Diamond_k$ 'in more than k successors, it holds that', where we write $\overline{\Diamond}_k = \Box_k$. It is interpreted over the functor $\mathcal{B}$ that takes a set X to the set $\mathcal{B}(X)$ of multisets over X, i.e. maps $B : X \to \mathbb{N} \cup \{\infty\}$, by $[\![\Diamond_k]\!]_X(A) = \{B \in \mathcal{B}(X) \mid \sum_{x \in A} B(x) > k\}$. This captures the semantics of graded modalities over *multigraphs* [7], which are precisely the $\mathcal{B}$-coalgebras. One can encode the description logic $\mathcal{ALCOQ}$ (which features *qualified number restrictions* $\geq nR$ and has a relational semantics) into multi-agent graded hybrid logic with multigraph semantics by adding formulas $\neg\Diamond_1 i$ for all occurring nominals i to the TBox.

3. *Probabilistic hybrid logic*, the hybrid extension of probabilistic modal logic [13], has modal operators L_p 'in the next step, it holds with probability at least p that', for $p \in [0,1] \cap \mathbb{Q}$. It is interpreted over the functor D_ω that maps a set X to the set of finitely-supported probability distributions on X by putting $[\![L_p]\!]_X(A) = \{P \in D_\omega(X) \mid PA \geq p\}$. Coalgebras for D_ω are just Markov chains.

3 Tableau Rules for Coalgebraic Logics

We now introduce the (type of) tableau rules we will be working with. Clearly, these rules have to relate syntax and semantics in an appropriate way, and we cannot expect to prove as much as soundness, let alone completeness, without the rules satisfying appropriate coherence conditions, which we introduce later. We begin with the propositional part of the calculus, for which it is convenient to unfold propositional connectives in a single step. This process is called *saturation* and is given by a map sat that is defined inductively by the clauses

$$\begin{aligned} \mathsf{sat}(\Delta') &= \{\Delta'\} & \qquad \mathsf{sat}(A \vee B, \Gamma) &= \mathsf{sat}(A, \Gamma) \cup \mathsf{sat}(B, \Gamma) \\ \mathsf{sat}(x, \overline{x}, \Gamma) &= \emptyset & \qquad \mathsf{sat}(A \wedge B, \Gamma) &= \mathsf{sat}(A, B, \Gamma) \end{aligned}$$

where $A, B \in \mathcal{H}(\Lambda)$ are formulas, Γ is a sequent, $x \in \mathsf{P} \cup \mathsf{N}$ is a propositional variable or a nominal and $\Delta' \in \mathsf{State}(\Lambda)$ is a state, i.e. contains neither complementary propositional variables nor top-level propositional connectives. As we interpret hybrid formulas over the class of *all* (coalgebraic) hybrid models, it suffices to use modal rules of a rather specific form where the premise contains only modalised formulas and the conclusion is purely propositional in terms of the arguments of the modalities. Rules of this type are called *one-step rules* and have been used in the context of tableau calculi in [9,6] and originate from the (dual) sequent rules of [19].

Definition 2. A *one-step tableau rule* over Λ is a tuple $(\Gamma_0, \Gamma_1, \ldots, \Gamma_n)$, written as $\Gamma_0/\Gamma_1 \ldots \Gamma_n$, where $\Gamma_0 \subseteq (\Lambda \cup \overline{\Lambda})(\mathsf{P} \cup \overline{\mathsf{P}})$ and $\Gamma_i \subseteq \mathsf{P} \cup \overline{\mathsf{P}}$ so that every variable that occurs in the conclusion $\Gamma_1 \ldots \Gamma_n$ also occurs in the premise Γ_0, and every propositional variable occurs at most once in the premise Γ_0.

We can think of one-step rules as a syntactic representation of the inverse image $\gamma^{-1} : \mathcal{P}(TC) \to \mathcal{P}(C)$ of a generic coalgebra map $\gamma : C \to TC$ in that the premise describes a property of successors, whereas the conclusion describes states. The requirement that propositional variables do not occur twice in the premise is a mere technical convenience, and can be met by introducing premises that state the equivalence of variables. While this rigid format of one-step rules suffices to completely axiomatise all coalgebraic logics [17], it does not accommodate frame conditions like transitivity ($\Box p \to \Box\Box p$), which require separate consideration.

Example 3. One-step rules that axiomatise the logics in Example 1 can be found (in the form of proof rules) in [16,19]. Continuing Example 1, we single out hybrid K and (hybrid) graded modal logic.

1. Hybrid K is axiomatised by the set $\mathcal{R}_K$ of one-step rules that contain
$$(K)\frac{\Diamond p_0, \Box p_1, \ldots, \Box p_n}{p_0, \ldots, p_n} \qquad \text{for all } n \geq 0.$$
2. The rules $\mathcal{R}_\mathcal{B}$ for graded modal logic contain
$$(G)\frac{\Diamond_{k_1} p_1, \ldots, \Diamond_{k_n} p_n, \Box_{l_1} q_1, \ldots, \Box_{l_m} q_m}{\sum_{j=1}^m s_j \overline{q}_j - \sum_{i=1}^n r_i p_i < 0}$$

where $n, m \in \mathbb{N}$ and $r_i, s_j \in \mathbb{N} \setminus \{0\}$ satisfy the side condition $\sum_{i=1}^n r_i(k_i + 1) \geq 1 + \sum_{j=1}^m s_j l_j$. The conclusion of (G) is to be read as arithmetic of characteristic functions, and expands into a disjunctive normal form with only positive literals [19].

While the above are examples of one-step rules, the generic treatment of a larger class of modal logic requires that we abstract away from concretely given rule sets. This is achieved by formalising *coherence conditions* that link the rules with the coalgebraic semantics. The following terminology is handy to formalise these conditions:

Definition 4. Suppose that X is a set, $\mathsf{P} \subseteq \mathsf{Prop}$ is a set of variables and $\tau : \mathsf{P} \to \mathcal{P}(X)$ is a valuation. The interpretation of a propositional sequent $\Gamma \subseteq \mathsf{P} \cup \overline{\mathsf{P}}$ relative to (X, τ) is given by $[\![\Gamma]\!]_{X,\tau} = \bigcap\{\tau(p) \mid p \in \Gamma\} \cap \bigcap\{X \setminus \tau(p) \mid \overline{p} \in \Gamma\} \subseteq X$. Modalised sequents $\Gamma \subseteq (\Lambda \cup \overline{\Lambda})(\mathsf{P} \cup \overline{\mathsf{P}})$ are interpreted, again relative to (X, τ), as subsets of TX by

$$[\![\Gamma]\!]_{TX,\tau} = \bigcap\{[\![\heartsuit]\!]_X([\![p_1]\!]_{X,\tau}, \ldots, [\![p_n]\!]_{X,\tau}) \mid \heartsuit(p_1, \ldots, p_n) \in \Gamma\}$$

where $p_1, \ldots, p_n \in \mathsf{P} \cup \overline{\mathsf{P}}$ and $\heartsuit \in \Lambda \cup \overline{\Lambda}$.

The coherence conditions can be formulated solely in terms of (the interpretation of) propositional and modal sequents. In particular, we do not consider models for the logic under scrutiny.

Definition 5. Suppose that $\mathcal{R}$ is a set of one-step tableau rules. We say that $\mathcal{R}$ is *one-step tableau sound* (resp. *one-step tableau complete*) with respect to T if, for all $\mathsf{P} \subseteq \mathsf{Prop}$, all finite $\Gamma \subseteq (\Lambda \cup \overline{\Lambda})(\mathsf{P} \cup \overline{\mathsf{P}})$, all sets X and valuations $\tau : \mathsf{P} \to \mathcal{P}(X)$: $[\![\Gamma]\!]_{TX,\tau} \neq \emptyset$ only if (if) for all rules $\Gamma_0/\Gamma_1 \ldots \Gamma_n \in \mathcal{R}$ and all renamings $\sigma : \mathsf{P} \to \mathsf{P}$ (such that $A \neq B$ implies $A\sigma \neq B\sigma$ for all $A, B \in \Gamma_0$) with $\Gamma_0\sigma \subseteq \Gamma$, we have that $[\![\Gamma_i\sigma]\!]_{X,\tau} \neq \emptyset$ for some $1 \leq i \leq n$.

This means that a rule set is sound and complete if a modalised sequent is satisfiable iff every one-step rule applicable to it has at least one satisfiable conclusion. We note that the rule sets given in Example 3 are both one-step sound and one-step complete for their respective interpretations. This is argued, in the dual case of sequent rules, in [19].

4 Caching Graphs for Coalgebraic Hybrid Logic

Caching graphs address the problem of deciding the validity of a sequent Γ_0 under a finite set of global assumptions (TBox) Ξ both of which we fix throughout. We write $\mathcal{C}$ for the *closure* of Γ_0, Ξ, i.e. the smallest set of formulas that contains Γ, Ξ and is closed under taking subformulas, their (involutive) negations and prefixing of plain formulas (that do not begin with $@$) with $@_i$ where $i \in \mathsf{N}(\mathcal{C})$; we then work with *sequents over* $\mathcal{C}$, i.e. subsets of $\mathcal{C}$, but mostly omit explicit mention of $\mathcal{C}$.

For a given one-step sound and complete set $\mathcal{R}$ of one-step rules, this allows us to consider the set $T(\mathcal{R})$ that consists of the rule instances that are needed to expand sequents over $\mathcal{C}$.

Definition 6. The set $T(\mathcal{R})$ of *tableau rules relative to* $\mathcal{C}$ consists of the rules $\Gamma/\mathsf{sat}(\Gamma)$ for $\Gamma \in \mathcal{C}$ and the rules

$$\frac{\Gamma\sigma, \Gamma'}{\Delta_1\sigma, \Xi \quad \dots \quad \Delta_n\sigma, \Xi}$$

where $\Gamma/\Delta_1, \dots, \Delta_n \in \mathcal{R}$, $\sigma : \mathsf{P} \to \mathcal{C}$ is a substitution such that $\Gamma\sigma \subseteq \mathcal{C}$ and σ does not identify elements of Γ, and $\Gamma' \subseteq \mathcal{C}$.

In other words, the set $T(\mathcal{R})$ of tableau rules relative to $\mathcal{C}$ consists of all substitution instances of one-step rules where we allow an additional sequent Γ' in the premise to absorb weakening, and the set Ξ of global assumptions is added to every conclusion. Informally, the conclusions of a modal rule specify properties of successor states, and adding Ξ to each conclusion ensures that successor states also validate Ξ, leading to a model that globally validates the TBox.

While the tableau rules are used to expand sequents, a second type of sequent is needed to deal with the @-formulas: since @-formulas are either globally true or globally false, they need to be propagated across the tableau, and their validity needs to be ascertained. This is the role of @-constraints that we now introduce, together with rules that govern their expansion.

Definition 7 (@-Constraints). An @-*constraint* over $\mathcal{C}$ is a finite set of @-formulas in $\mathcal{C}$ that may additionally include the symbol $\bullet$. The expansion of @-constraints is governed by the rules $T(@)$ that contain, for each @-constraint Υ over $\mathcal{C}$, the following @-*expansion rules*

$$\frac{\Upsilon}{i, \Upsilon/@_i, \Upsilon \setminus \{\bullet\}, \Xi}$$

where $i \in \mathsf{N}(\Upsilon)$ is a nominal occurring in Υ and $\Upsilon/@_i = \{A \mid @_i A \in \Upsilon\}$.

The role of @-constraints is to record those formulas that are required to be *globally* valid (and therefore need to satisfy the global assumptions Ξ)to guarantee the satisfiability of a particular sequent. To check whether a particular @-constraint is satisfiable, we therefore need to check, for each applicable @-expansion rule, the consistency of the conclusion. The role of $\bullet$ as an element of an @-constraint is to denote an (as yet) unknown constraint to be induced by a sequent that is yet to be expanded. We next introduce caching graphs, which provide the fundamental data structure that allows for the propagation of these constraints.

Definition 8. A *caching graph* over $\mathcal{C}$ is a tuple $G = (S, C, L_M, L_@, \lambda_S, \lambda_C)$ where

- S and C are sets of sequents and @-constraints respectively
- $L_M = (L_M^\forall, L_M^\exists)$ is a pair of relations with $L_M^\forall \subseteq S \times \mathcal{P}(S)$ and $L_M^\exists \subseteq \mathcal{P}(S) \times S$
- $L_@ = (L_@^\forall, L_@^\exists)$ is a pair of relations with $L_@^\forall \subseteq C \times S$ and $L_@^\exists \subseteq S \times C$; we require that $L_@^\exists$ is *upclosed*, i.e. $(\Gamma, \Upsilon) \in L_@^\exists$ and $\Upsilon \subseteq \Upsilon'$ imply $(\Gamma, \Upsilon') \in L_@^\exists$
- $\lambda_S : S \to \{A, E, U, X\}$ and $\lambda_C : C \to \{T, D\}$ are labelling functions.

We denote the *upclosure* (under $\subseteq$) of a set $\mathcal{B}$ of @-constraints by $\uparrow \mathcal{B}$.

We think of L_M as the "modal links" where $L_M^\forall$ links sequents to sets of conclusions of modal rules, and $L_M^\exists$ links conclusions (sets of sequents) to their individual elements. This reflects the fact that to declare a sequent Γ satisfiable, we need to select, for all rules applicable to this sequent (all $(\Gamma, \Psi) \in L_M^\forall$) one conclusion $\Delta \in \Psi$ (there exists $(\Psi, \Delta) \in L_M^\exists$) which is recursively satisfiable. Similarly, $L_@$ encodes the global constraints (which we later propagate) that ensure that a sequent is satisfiable. For example for the sequent $\Gamma = @_i A \vee @_j B$ to be satisfiable, *either* the @-constraint $@_i A$ *or* the @-constraint $@_j B$ needs to be satisfiable, so that $\{(\Gamma, @_i A), (\Gamma, @_i B)\} \subseteq L_@^\exists$, and we ask for the existence of a $L_@^\exists$-successor of Γ. The required upclosure corresponds to the fact that – in order to satisfy an @-constraint – it suffices to satisfy any larger constraint. Technically, requiring upclosure simplifies the definition of @-propagation below. To guarantee the satisfiability of @-constraints, we link every @-constraint to a set of sequents (one for each nominal) that stipulate the validity of these constraints. For example, the @-constraint $\Upsilon = @_i A, @_j B$ requires that A holds at i and B holds at j which stipulates that both $\Upsilon_i = i, A, \Upsilon$ and $\Upsilon_j = j, B, \Upsilon$ should be satisfiable. This is represented by stipulating that $\{(\Upsilon, \Upsilon_i), (\Upsilon, \Upsilon_j)\} \subseteq L_@^\forall$ where again $\forall$ indicates universal choice.

The role of the labelling functions is essentially for bookkeeping. The label $\lambda_S(\Gamma)$ of a sequent Γ indicates whether the sequent is satisfiable (E: a winning position for the existential player), unsatisfiable (A: a winning position for the universal player), unknown (U) or unexpanded (X). Similarly, the label of an @-constraint Υ indicates that this constraint is expanded (D for done) or unexpanded (T for todo).

Remark 9. In a concrete implementation of caching graphs, it is sufficient to represent the upwards closed sets $L_@^\exists(\Gamma) = \{\Upsilon \mid (\Gamma, \Upsilon) \in L_@^\exists\}$ by means of a set of generators, which dramatically reduces the size of caching graphs.

We now introduce a set of transitions between caching graphs that correspond to expansion of both sequents and @-constraints, propagation of @-constraints and updating of winning positions. We begin with sequent expansion.

Definition 10 (Sequent Expansion). Suppose $G = (S, C, L_M, L_@, \lambda_C, \lambda_S)$ is a caching graph. We put $G \to_E G'$ and say that G' arises from G through *sequent expansion* if there is an unexpanded sequent $\Gamma \in S$ (i.e. $\lambda_S(\Gamma) = X$), and G' arises from G by inserting all relations (Γ, Ψ) where $\Gamma/\Psi \in T(\mathcal{R})$ is a rule into $L_M^\forall$, all ensuing relations (Ψ, Δ) with $\Delta \in \Psi$ to $L_M^\forall$, updating S to contain new sequents Δ that have been encountered in this procedure and setting their status to unexpanded (i.e. $\lambda_S(\Delta) = X$), equipping them with all @-constraints that contain $\bullet$ and finally marking Γ as unknown ($\lambda_S(\Gamma) = U$).

The situation is somewhat dual for @-constraints, where the universal links are added by a simple expansion process, but the existential links arise via propagation. Given that satisfiability of a sequent is conditional on the satisfiability of one of the associated @-constraints, we need a mechanism to check their satisfiability. In a nutshell, for an @-constraint to be satisfiable, we need to check, for each nominal, that the formulas deemed to be valid at this nominal are jointly satisfiable. While this results in an (ordinary) sequent, this process may uncover more constraints, and we therefore need to remember the set of @-constraints that we started out with. Formally, expansion of @-constraints takes the following form:

Definition 11 (@-Expansion). Suppose $G = (S, C, L_M, L_@, \lambda_S, \lambda_C)$ is a caching graph. We put $G \to_{@E} G'$ and say that G' arises from G through *@-expansion* if there exists a 'todo'-constraint $\varUpsilon \in C$ (i.e. $\lambda_C(\varUpsilon) = T$ and G' arises from G by inserting all sequents Γ for which $\varUpsilon/\Gamma \in T(@)$ to $L^{\exists}_{\forall}(\varUpsilon)$ and adding all new sequents to S, marking them as unexpanded $(\lambda_C(\Gamma) = X$, equipping new sequents with all @-constraints containing $\bullet$, and finally marking $\varUpsilon$ as done $(\lambda_C(\varUpsilon) = D)$.

Informally, every @-constraint $\varUpsilon$ specifies, for each nominal i, a set of formulas that are to be valid at i, which are collected in the @-demands. As the expansion of these formulas may unearth further @-formulas (possibly involving nominals distinct from i), the original @-constraint $\varUpsilon$ is remembered in the @-demand. For the existential @-links the situation is more complicated, as @-links emanating from a sequent describe constraints (sets of @-prefixed formulas) that need to be satisfied for the sequent to be satisfiable. As @-prefixed formulas are either globally true or globally false, these constraints must hold at *all* points of a putative model. This necessitates distributing those constraints from one node of a tableau graph to the others. In general, every tableau node comes with a finite number of @-constraints where each particular constraint represents one requirement under which the associated sequent is satisfiable, such $@_i A$ or $@_j B$ for the above-mentioned example sequent $@_i A \vee @_j B$. As a consequence, we need to analyse the universal / existential branching structure of the caching graph during the propagation phase. As we are dealing with a possibly circular graph (due to the global assumptions), propagation is formalised as a *greatest* fixpoint computation.

Definition 12 (@-Propagation). Let $\mathcal{A}$ denote the set of all @-constraints that can be formed in the closed set $\mathcal{C}$. Suppose $G = (S, C, L_M, L_@, \lambda_S, \lambda_C)$ is a caching graph and $R \subseteq S \times \mathcal{A}$. The set $C_R(\Gamma)$ of *R-constraints* of Γ consists of all @-constraints of the form $\varUpsilon_1, \dots, \varUpsilon_k$ such that for some $(\Delta_1, \dots, \Delta_k) \in \prod_{(\Gamma,\Psi) \in L^{\forall}_M} \Psi$, $(\Delta_i, \varUpsilon_i) \in R$ and $\lambda_S(\Delta_i) \neq A$ for $i = 1, \dots, k$. In other words, an R-constraint of Γ collects, for every rule applicable to Γ, one constraint of one rule conclusion. In particular, $C_R(\Gamma) = \emptyset$ if $(\Gamma/\emptyset) \in L^{\forall}_M$ (i.e. Γ is inconsistent) and $C_R(\Gamma) = \{\emptyset\}$ in case no rule is applicable to Γ. Recall that $L^{\exists}_@$ is maintained as an upclosed relation of type $S \times \mathcal{A}$. Thus, let $\mathcal{R} = \{R \subseteq L^{\exists}_@ \mid R \text{ upclosed}\}$, and define a monotone operator $W_@ : \mathcal{R} \to \mathcal{R}$ by

$$W_@(R)(\Gamma) = \begin{cases} \uparrow \{\{\bullet\}\} & (\lambda_S(\Gamma) = X) \\ \uparrow \{@\Gamma, \Theta_1, \Theta_2, \mid \Theta_1 \in R(\Gamma), \Theta_2 \in C_R(\Gamma)\} & \text{(otherwise)} \end{cases}$$

where $@\Gamma = \{@_i A \mid i, A \subseteq \Gamma \text{ and } A \text{ not an @-formula}\} \cup \{A \in \Gamma \mid A \text{ an @-formula}\}$ so that to an expanded sequent, we associate its own @-formulas together with one previously computed constraint and one constraint that is propagated upwards from its children. We put $G \to_{@P} G'$ if $G' = (S, C', L_M, L'_@, \lambda_S, \lambda'_C)$ where

- $C' = C \cup \{\Upsilon \mid \exists \Gamma \in S\ ((\Gamma, \Upsilon) \in \nu W_@)\}$,
- ${L^{\exists}_@}' = \nu W_@$ and ${L^{\forall}_@}' = L^{\forall}_@$
- $\lambda'_C(\Upsilon) = \lambda_C(\Upsilon)$ if $\Upsilon \in C$ and $\lambda'_C(\Upsilon) = T$, otherwise

and say that G' arises from G through $@$-*propagation*.

Some comments are in order regarding the above definition of @-propagation. Updating $L^{\exists}_@$ requires us to compute an upclosed relation $R \subseteq S \times \mathcal{A}$; because constraints grow monotonically (due to sequent expansion), we will have $R \subseteq L^{\exists}_@$ (possibly throwing out some smaller constraints). To propagate @-constraints from the children of a sequent Γ up to Γ itself, note that Γ is satisfiable if for all applicable rules Γ/Ψ, there exists at least one satisfiable conclusion in $\Delta \in \Psi$. In particular, one of the @-constraints associated with Δ needs to be satisfiable. In other words, for Γ to be satisfiable it is necessary to be able to simultaneously select one @-constraint $\Upsilon \in R(\Delta)$ from one of the conclusions $\Delta \in \Psi$ for each of the rules Γ/Ψ that are applicable to Γ. If we think of R as defining an over-approximation of @-constraints, we keep for each Γ only those constraints that contain the @-formulas of Γ and one of the R-constraints of Γ. This is precisely the effect of one application of $W_@$, and we compute the greatest fixpoint of $W_@$ to propagate this information across cycles in the tableau graph.

The final crucial step is the updating of winning positions. Here, the intuition is that a given sequent is satisfiable if we can select a *complete* set of @-constraints so that all @-demands of this set are satisfiable – i.e. for each of the @-demands, we must be able to (recursively) pinpoint a complete set of @-constraints for which the same condition holds. We call a set of @-constraints complete if it represents full information, that is, collects all constraints that ensure that – if these constraints are satisfied – the sequent under consideration does not have a closed tableau. This is where $\bullet$ comes in: @-constraints that do not include $\bullet$ are complete. On the other hand, $\forall$ can win from a given sequent if all of the (possibly still incomplete) @-constraints (recursively) have at least one unsatisfiable @-demand.

Definition 13 (Position Propagation). Suppose that $G = (S, C, L_M, L_@, \lambda_S, \lambda_C)$ is a caching graph. By abuse of notation, write

$$U = \lambda_S^{-1}(U) \quad E = \lambda_S^{-1}(E) \text{ and } A = \lambda_S^{-1}(A)$$

for the sets of sequents that are labelled with U, E and A respectively. Define two monotone operators $M, W : \mathcal{P}(U) \to \mathcal{P}(U)$ by

$$M(X) = \{\Gamma \in U \mid \forall (\Gamma, \Upsilon) \in L^{\exists}_@\ \exists (\Upsilon, \Delta) \in L^{\forall}_@\ (\Delta \in X \cup A)\}$$
$$W(X) = \{\Gamma \in U \mid \exists (\Gamma, \Upsilon) \in L^{\exists}_@\ (\bullet \notin \Upsilon) \text{ and } \forall (\Upsilon, \Delta) \in L^{\forall}_@\ (\Delta \in X \cup E)\}.$$

We put $G \to_P G'$ if $G' = (S, C, L_M, L_@, \lambda'_S, \lambda_C)$ where $\lambda'_S(\Gamma) = A$ if $\Gamma \in A \cup \mu M$, $\lambda'_S(\Gamma) = E$ if $\Gamma \in E \cup \nu W$ and $\lambda'_S(\Gamma) = \lambda_S(\Gamma)$, otherwise, and say that G' arises from G through *position propagation*.

We now have all ingredients in place to describe the algorithm for deciding the satisfiability of a sequent $\Gamma \subseteq \mathcal{H}(\Lambda)$. This algorithm non-deterministically applies expansion, propagation and update steps until the initial sequent is either marked A (unsatisfiable) or E (satisfiable).

Algorithm 14. Decide whether Γ_0 is satisfiable in $\mathsf{Mod}(\Xi)$.

```
1. Initialise: put $G = (\{\Gamma_0, \Xi\}, \uparrow \{\{\bullet\}\}, \emptyset, L_@, \lambda_S, \lambda_C)$ where
   – $\lambda_S(\Gamma_0, \Xi) = X$, and $\lambda_C(\Upsilon) = T$ everywhere;
   – $L_@^\exists$ is total and $L_@^\forall = \emptyset$.
2. While $(\lambda_S(X)^{-1} \neq \emptyset)$ or $(\lambda_C^{-1}(T) \neq \emptyset)$ do
   (a) choose $G'$ with $G \to_E G'$ or $G \to_{@E} G'$ and let $G := G'$;
   (b) (optional) choose $G'$ with $G \to_{@P} G'$ and let $G := G'$;
   (c) (optional)
         – choose $G'$ with $G \to_P G'$ and let $G := G'$;
         – return 'yes' if $\lambda_S(\Gamma) = E$ and 'no' if $\lambda_S(\Gamma) = A$.
3. Find $G'$ with $G \to_{@P} G'$, let $G := G'$, and continue with Step 2.
4. Find $G'$ with $G \to_P G'$ and let $G := G'$.
5. Return 'yes' if $\lambda_S(\Gamma) = E$ and 'no' if $\lambda_S(\Gamma) = A$.
```

In the above formulation, the algorithm nondeterministically expands sequents or @-constraints and interleaves @-propagation and position update. Since @-propagation may create new @-constraints, we need to make sure that all @-constraints are eventually created, which is ensured by going back to Step 2 after @-propagation in Step 3. This procedure terminates, after at most exponentially many steps, as there are at most exponentially many @-constraints and sequents (measured in the size of the initial sequent Γ_0 and the TBox Ξ), and the final position update ensures that all sequents are marked accordingly. Note that we may terminate at any time after the initial sequent has been marked as either satisfiable or unsatisfiable after a position update.

5 Correctness and Completeness

We begin by showing that a sequent marked as satisfiable by Algorithm 14 is indeed satisfiable. This necessitates the construction of a satisfying model, which is based on a *named tableau graph*. Simply put, a named tableau graph consists of sequents Γ so that for every rule applicable to Γ, one of the conclusions occurs in the tableau graph, and is connected to Γ. In order to also satisfy @-formulas, we require that the tableau graph be *named*, as introduced next.

Definition 15. A *tableau graph* over a finite set S of sequents is a graph $G_T = (S, L)$ where $L \subseteq (S \times \mathcal{P}(S)) \cup (\mathcal{P}(S) \times S)$ is such that

- for all $\Gamma \in S$ and all $\Gamma/\Psi \in T(\mathcal{R})$ there exists $\Delta_{(\Gamma,\Psi)} \in \Psi$ such that $L = \{(\Gamma, \Psi) \mid \Gamma/\Psi \in T(\mathcal{R})\} \cup \{(\Psi, \Delta_{(\Gamma,\Psi)}) \mid \Gamma/\Psi \in T(\mathcal{R})\}$.

We say that (S, L) is a *named* tableau graph if additionally

- for each $i \in N$, there exists exactly one $\Gamma_i \in S$ with $i \in \Gamma$, and
- for all $\Gamma \in S$ and all $@_i A \in \Gamma$ we have $A \in \Gamma_i$.

The crucial stepping stone in the correctness proof for Algorithm 14 is the fact that we can construct a satisfying model based on a named tableau graph.

Lemma 16 (Model Existence Lemma). *If G_T is a named tableau graph over a set S of sequents, there exists a coalgebra structure $\sigma : W \to TW$ on the set of states contained in S and a hybrid valuation $\pi : \mathsf{N} \cup \mathsf{P} \to \mathcal{P}(W)$ such that $\Gamma \in [\![A]\!]_{(W,w,\pi)}$ for all $A \in \Gamma$.*

In order to be marked as satisfiable by a position update step in Algorithm 14, we need to be able to select a $\bullet$-free @-constraint for every satisfiable sequent. This entails the existence of a tableau graph for this sequent, and we will later merge these graphs.

Lemma 17. *Suppose that during the execution of Algorithm 14, $(\Gamma, \Upsilon) \in L^{\exists}_{@}$ with $\bullet \notin \Upsilon$ for a caching graph G. Then there exists a (not necessarily named) tableau graph (S, T) with $\Gamma \in S$.*

The first part of the correctness assertion can now be established as follows:

Lemma 18 (Completeness). *Every sequent marked 'satisfiable' by Algorithm 14 is satisfiable in* $\mathsf{Mod}(\Xi)$.

The second half of the correctness of Algorithm 14 needs the following preliminary lemma that shows that satisfiable sequents have satisfiable @-constraints.

Lemma 19. *Throughout the construction of the caching graph $G = (S, C, L_M, L_@, \lambda_S, \lambda_C)$ by Algorithm 14, it holds that for every satisfiable sequent $\Gamma \in S$ there exists an @-constraint $\Upsilon \in C$ such that $(\Gamma, \Upsilon) \in L^{\exists}_{@}$, $@\Gamma \subseteq \Upsilon$, and $\Upsilon \setminus \{\bullet\}$ is satisfiable.*

With the help of the last lemma, the second half of correctness of Algorithm 14 can now be established as follows:

Lemma 20. *Every sequent marked 'unsatisfiable' by Algorithm 14 is unsatisfiable in* $\mathsf{Mod}(\Xi)$.

Finally, we need to establish that Algorithm 14 in fact marks the initial sequent Γ_0, Ξ as either satisfiable or unsatisfiable, which only requires proof if Algorithm 14 terminates in Step 5. This rests on the final position update, and we first show that every @-constraint is 'morally' complete, that is, can be turned into a complete @-constraint by removing $\bullet$. This trivialises the condition $\bullet \notin \Upsilon$ in the definition of position update, which is used in the following lemma.

Lemma 21. *If Algorithm 14 terminates in Step 5, then the following holds for all $\Gamma \in S$: If $(\Gamma, \Upsilon) \in L^{\exists}_{@}$ then $(\Gamma, \Upsilon \setminus \{\bullet\}) \in L^{\exists}_{@}$.*

Finally, we show that Algorithm 14 always delivers a result (which is correct by Lemma 20 and Lemma 18), and thus finally confirm correctness.

Lemma 22. *When Algorithm 14 terminates in Step 5, each sequent is either marked satisfiable or unsatisfiable.*

Correctness of Algorithm 14 is a consequence of Lemma 18, Lemma 20 and Lemma 22.

Theorem 23. *For any given sequent Γ, Algorithm 14 delivers the answer 'yes' if Γ is satisfiable in* $\mathsf{Mod}(\Xi)$ *and the answer 'no' otherwise.*

6 Complexity

We proceed to analyse the runtime of the global caching algorithm, under suitable sanity assumptions on the set of modal rules. Specifically, in order to ensure that executions of the algorithm run in exponential time, we need to assume, as in [9,20], that our set $\mathcal{R}$ of one-step rules is EXPTIME-*tractable* in the following sense. To begin, a *demand* of a sequent Δ is a sequent $\Gamma_i\sigma$, $i \geq 1$, where $\Gamma_0/\Gamma_1 \dots \Gamma_n \in \mathcal{R}$ is a modal rule and σ is a substitution such that $\Gamma_0\sigma \subseteq \Delta$ and σ does not identify any two formulas in Γ_0. In this case, σ *matches* $\Gamma_0/\Gamma_1 \dots \Gamma_n$ to Δ. Then, we say that $\mathcal{R}$ is EXPTIME-tractable if there exists a coding of the rules such that all demands of a sequent can be generated by rules with codes of polynomially bounded size, validity of codes and membership of a sequent in the set of premises of a coded rule are decidable in EXPTIME, and matching substitutions for a given rule code/sequent pair can be enumerated in exponential time.

Theorem 24. *If the given set* $\mathcal{R}$ *of one-step rules is* EXPTIME*-tractable, then every execution of the global caching algorithm (Algorithm 14) runs in* EXPTIME.

We emphasize explicitly that, although the global caching algorithm is non-deterministic, it does not have any inherent non-determinism: *every* terminating execution yields the correct answer, so that the non-determinism works in favour of the implementer, who now has a chance to achieve improved average case behaviour by using suitable heuristics in his strategy for choosing expansion, propagation, and update steps. In particular, the above theorem does reprove the tight EXPTIME bound from [20].

7 Conclusions

We have presented an optimal tableau algorithm for hybrid modal logic over arbitrary TBoxes that is applicable to all (hybrid) logics with coalgebraic semantics. Instantiated to the modal logic K or a multi-modal variant, such as the description logic $\mathcal{ALCO}$, this provides, to our knowledge, the first purely syntax driven and backtracking-free tableau algorithm that realizes optimal (EXPTIME) complexity bounds. However, the scope of the coalgebraic framework is much broader, and the built-in parametricity uniformly provides us with optimal tableau-based decision procedures, e.g., for hybrid graded modal logic (or the description logic $\mathcal{ALCOQ}$), hybrid probabilistic modal logic, or hybrid logics for coalitional power in games. The compositionality of coalgebraic logics [18] in particular allows us to obtain optimal tableau algorithms for logics that mix the above features (see [20] for examples). The most pressing research concern at this point is, of course, experimental evaluation, which is the subject of ongoing work, and we plan to extend the CoLoSS system [5] that already implements the (modal) proof rules for a large variety of logics.

References

1. Areces, C., ten Cate, B.: Hybrid logics. In: Blackburn, P., van Benthem, J., Wolter, F. (eds.) Handbook of Modal Logic. Elsevier, Amsterdam (2007)
2. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F. (eds.): The Description Logic Handbook. Cambridge University Press, Cambridge (2003)

3. Bolander, T., Blackburn, P.: Termination for hybrid tableaus. J. Log. Comput. 17, 517–554 (2007)
4. Bolander, T., Braüner, T.: Tableau-based decision procedures for hybrid logic. J. Log. Comput. 16(6), 737–763 (2006)
5. Calin, G., Myers, R., Pattinson, D., Schröder, L.: CoLoSS: The coalgebraic logic satisfiability solver (system description). In: Methods for Modalities, M4M-5. ENTCS. Elsevier, Amsterdam (to appear, 2008)
6. Cîrstea, C., Kupke, C., Pattinson, D.: EXPTIME tableaux for the coalgebraic μ-calculus. In: Grädel, E., Kahle, R. (eds.) CSL 2009. LNCS, vol. 5771, pp. 179–193. Springer, Heidelberg (2009)
7. D'Agostino, G., Visser, A.: Finality regained: A coalgebraic study of Scott-sets and multisets. Arch. Math. Logic 41, 267–298 (2002)
8. Donini, F.M., Massacci, F.: EXPTIME tableaux for ALC. Artif. Intell. 124, 87–138 (2000)
9. Gore, R., Kupke, C., Pattinson, D.: Optimal tableau algorithms for coalgebraic logics. In: Tools and Algorithms for the Construction and Analysis of Systems, TACAS 10. LNCS. Springer, Heidelberg (2010)
10. Goré, R., Nguyen, L.: EXPTIME tableaux for $\mathcal{ALC}$ using sound global caching. In: Description Logics, DL '07, CEUR Workshop Proceedings, vol. 250 (2007)
11. Goré, R., Nguyen, L.: EXPTIME tableaux with global caching for description logics with transitive roles, inverse roles and role hierarchies. In: Olivetti, N. (ed.) TABLEAUX 2007. LNCS (LNAI), vol. 4548, pp. 133–148. Springer, Heidelberg (2007)
12. Horrocks, I., Patel-Schneider, P.F.: Optimising description logic subsumption. J. Logic Comput. 9, 267–293 (1999)
13. Larsen, K., Skou, A.: Bisimulation through probabilistic testing. Inf. Comput. 94, 1–28 (1991)
14. Myers, R., Pattinson, D., Schröder, L.: Coalgebraic hybrid logic. In: de Alfaro, L. (ed.) FOSSACS 2009. LNCS, vol. 5504, pp. 137–151. Springer, Heidelberg (2009)
15. Pattinson, D.: Coalgebraic modal logic: Soundness, completeness and decidability of local consequence. Theoret. Comput. Sci. 309, 177–193 (2003)
16. Pattinson, D., Schröder, L.: Cut elimination in coalgebraic logics. Inf. Comput. (to appear)
17. Schröder, L.: A finite model construction for coalgebraic modal logic. J. Log. Algebr. Prog. 73, 97–110 (2007)
18. Schröder, L., Pattinson, D.: Modular algorithms for heterogeneous modal logics. In: Arge, L., Cachin, C., Jurdziński, T., Tarlecki, A. (eds.) ICALP 2007. LNCS, vol. 4596, pp. 459–471. Springer, Heidelberg (2007)
19. Schröder, L., Pattinson, D.: PSPACE bounds for rank-1 modal logics. ACM Trans. Comput. Log. 10, 13:1–13:33 (2009); Earlier version in LICS'06 (2006)
20. Schröder, L., Pattinson, D., Kupke, C.: Nominals for everyone. In: International Joint Conferences on Artificial Intelligence, IJCAI 09, pp. 917–922. AAAI Press, Menlo Park (2009)
21. Thomas, W.: On the synthesis of strategies in infinite games. In: Mayr, E.W., Puech, C. (eds.) STACS 1995. LNCS, vol. 900, pp. 1–13. Springer, Heidelberg (1995)

Tractable Extensions of the Description Logic $\mathcal{EL}$ with Numerical Datatypes

Despoina Magka, Yevgeny Kazakov, and Ian Horrocks

Oxford University Computing Laboratory
Wolfson Building, Parks Road, OXFORD, OX1 3QD, UK
{despoina.magka,yevgeny.kazakov,ian.horrocks}@comlab.ox.ac.uk

Abstract. We consider extensions of the lightweight description logic (DL) $\mathcal{EL}$ with numerical datatypes such as naturals, integers, rationals and reals equipped with relations such as equality and inequalities. It is well-known that the main reasoning problems for such DLs are decidable in polynomial time provided that the datatypes enjoy the so-called convexity property. Unfortunately many combinations of the numerical relations violate convexity, which makes the usage of these datatypes rather limited in practice. In this paper, we make a more fine-grained complexity analysis of these DLs by considering restrictions not only on the kinds of relations that can be used in ontologies but also on their occurrences, such as allowing certain relations to appear only on the left-hand side of the axioms. To this end, we introduce a notion of safety for a numerical datatype with restrictions (NDR) which guarantees tractability, extend the $\mathcal{EL}$ reasoning algorithm to these cases, and provide a complete classification of safe NDRs for natural numbers, integers, rationals and reals.

Keywords: description logic, computational complexity, datatypes.

1 Introduction and Motivation

Description logics (DLs) [1] provide a logical foundation for modern ontology languages such as OWL[1] and OWL 2 [2]. $\mathcal{EL}^{++}$ [3] is a lightweight DL for which reasoning is tractable (i.e., can be performed in time that is polynomial w.r.t. the size of the input), and that offers sufficient expressivity for a number of life-sciences ontologies, such as SNOMED CT [4] or the Gene Ontology [5]. Among other constructors, $\mathcal{EL}^{++}$ supports limited usage of datatypes. In DL, datatypes (also called concrete domains) can be used to define new concepts by referring to particular values, such as strings or integers. For example, the concept $\mathsf{Human} \sqcap \exists \mathsf{hasAge}.(<, 18) \sqcap \exists \mathsf{hasName}.(=, \text{“}\mathsf{Alice}\text{”})$ describes humans whose age is less than 18 and whose name is “Alice”. Datatypes are characterised first by the domain their values can come from and also by the relations that can be used to constrain possible values. In our example, $(<, 18)$ refers to the domain of natural numbers and uses the relation “$<$” to constrain possible values to

[1] http://www.w3.org/2004/OWL

those less than 18, while (=, "Alice") refers to the domain of strings and uses the relation "=" to constrain the value to "Alice".

In order to ensure that reasoning remains polynomial, $\mathcal{EL}^{++}$ allows only for datatypes which satisfy a condition called p-admissibility [3]. In an nutshell, this condition ensures that the satisfiability of datatype constraints can be solved in polynomial time, and that concept disjunction cannot be expressed using datatype concepts. For example, if we were to allow both $\leq$ and $\geq$ for integers, then we could express $A \sqsubseteq B \sqcup C$ by formulating the axioms $\mathsf{A} \sqsubseteq \exists \mathsf{R}.(\leq, 5)$, $\exists \mathsf{R}.(\leq, 2) \sqsubseteq \mathsf{B}$ and $\exists \mathsf{R}.(\geq, 2) \sqsubseteq \mathsf{C}$. Thus, allowing both $\leq$ and $\geq$ has the same effect as extending $\mathcal{EL}^{++}$ with disjunction, which is well known to cause intractability [3]. Similarly, we can show that p-admissibility prevents from having both $\leq$ and $=$ or both $\geq$ and $=$ in the language. For this reason, the EL Profile of OWL 2, which is based on $\mathcal{EL}^{++}$, admits only equality (=) in datatype expressions.

In this paper, we demonstrate how these restrictions can be significantly relaxed without loosing tractability. As a motivating example, consider the following two axioms which might be used, e.g., in a pharmacy-related ontology:

$$\mathsf{Panadol} \sqsubseteq \exists \mathsf{contains}.(\mathsf{Paracetamol} \sqcap \exists \mathsf{mgPerTablet}.(=, 500)) \tag{1}$$

$$\begin{aligned}&\mathsf{Patient} \sqcap \exists \mathsf{hasAge}.(<, 6) \sqcap \\ &\quad \exists \mathsf{hasPrescription}.\exists \mathsf{contains}.(\mathsf{Paracetamol} \sqcap \exists \mathsf{mgPerTablet}.(>, 250)) \sqsubseteq \bot\end{aligned} \tag{2}$$

Axiom (1) states that the drug Panadol contains 500 mg of paracetamol per tablet, while axiom (2) states that a drug that contains more than 250 mg of paracetamol per tablet must not be prescribed to a patient younger than 6 years old. The ontology could be used, for example, to support clinical staff who want to check whether Panadol can be prescribed to a 3-year-old patient. This can easily be achieved by checking whether the following concept is satisfiable w.r.t. the ontology:

$$\mathsf{Patient} \sqcap \exists \mathsf{hasAge}.(=, 3) \sqcap \exists \mathsf{hasPrescription}.\mathsf{Panadol} \tag{3}$$

Unfortunately, this is not possible using $\mathcal{EL}^{++}$, because axioms (1) and (2) involve both equality (=) and inequalities (<, >), and this violates the p-admissibility restriction. In this paper we demonstrate that it is, however, possible to express axioms (1) and (2) and concept (3) in a tractable extension of $\mathcal{EL}$. A polynomial classification procedure can then be used to determine the satisfiability of (3) w.r.t. the ontology by checking if adding an axiom

$$X \sqsubseteq \mathsf{Patient} \sqcap \exists \mathsf{hasAge}.(=, 3) \sqcap \exists \mathsf{hasPrescription}.\mathsf{Panadol}$$

for some new concept name X would entail $X \sqsubseteq \bot$.

Our idea is based on the intuition that equality in (1) and (3) serves a different purpose than inequalities do in (2). Equality in (1) and (3) is used to state a *fact* (the content of a drug and the age of a patient) whereas inequalities in (2) are used to trigger a *rule* (what happens if a certain quantity of drug is prescribed to a patient of a certain age). In other words, equality is used *positively* and inequalities are used *negatively.* It seems reasonable to assume that positive usages of datatypes will typically involve equality since a fact can usually be precisely stated. On the other hand, it seems reasonable to assume that negative occurrences of datatypes will typically involve equality as well as inequalities since a rule usually applies to a range of situations. In this paper, we make a fine-grained study of datatypes in $\mathcal{EL}$ by considering restrictions not only on the kinds of relations included in a datatype, but also on whether the relations can be used positively or negatively.

The main contributions of this paper can be summarised as follows:

1. We introduce the notion of a *Numerical Datatype with Restrictions* (*NDR*) that specifies the domain of the datatype, the datatype relations that can be used positively and the datatype relations that can be used negatively.
2. We extend the $\mathcal{EL}$ reasoning algorithm [3] to provide a polynomial reasoning procedure for an extension of $\mathcal{EL}$ with NDRs, and we prove that this procedure is sound for any NDR.
3. We introduce the notion of a *safe NDR*, show that every extension of $\mathcal{EL}$ with a safe NDR is tractable, and prove that our reasoning procedure is complete for any safe NDR.
4. Finally, we provide a complete classification of safe NDRs for the cases of natural numbers, integers, rationals and reals. Notably, we demonstrate that the numerical datatype restrictions can be significantly relaxed by allowing arbitrary numerical relations to occur negatively—not only equality as currently specified in the OWL 2 EL Profile. As argued earlier, this combination is of particular interest to ontology engineering, and is thus a strong candidate for the next extension of the EL Profile in OWL 2.

This work is based on a Master's thesis [6].

2 Preliminaries

In this section we introduce an extension of $\mathcal{EL}^{\perp}$ [3] with numerical datatypes which we denote by $\mathcal{EL}^{\perp}(\mathcal{D})$. In the DL literature the notion of a datatype is better known as a concrete domain [7]; we call them datatypes to be more consistent with OWL 2 [2]. The syntax of $\mathcal{EL}^{\perp}(\mathcal{D})$ uses a set of *concept names* N_C, a set of *role names* N_R and a set of *feature names* N_F. $\mathcal{EL}^{\perp}(\mathcal{D})$ is parametrised with a *numerical domain* $\mathcal{D}$, such that $\mathcal{D} \subseteq \mathbb{R}$, where $\mathbb{R}$ is the set of real numbers. N_C, N_R and N_F are countably infinite sets and, additionally, pairwise disjoint.

Definition 1 ($\mathcal{D}$-Datatype Restriction). *We call (s, y), where $y \in \mathcal{D}$ and $s \in \{<, \leq, >, \geq, =\}$, a $\mathcal{D}$-*datatype restriction *(or simply a* datatype restriction *if the domain $\mathcal{D}$ is clear from the context). Given a domain $\mathcal{D}$, a $\mathcal{D}$-datatype restriction $r = (s, y)$ and an $x \in \mathcal{D}$, we say that x satisfies r and we write $r(x)$ iff $(x, y) \in s$, where s is interpreted as the corresponding standard relation on real numbers.*

Table 1. Concept descriptions in $\mathcal{EL}^{\bot}(\mathcal{D})$

Name	Syntax	Semantics
Concept name	C	$C^{\mathcal{I}}$
Top	$\top$	$\Delta^{\mathcal{I}}$
Bottom	$\bot$	$\emptyset$
Conjunction	$C \sqcap D$	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$
Existential restriction	$\exists R.C$	$\{x \in \Delta^{\mathcal{I}} \mid \exists y \in \Delta^{\mathcal{I}}:\ (x,y) \in R^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\}$
Datatype restriction	$\exists F.r$	$\{x \in \Delta^{\mathcal{I}} \mid \exists v \in \mathcal{D}:\ (x,v) \in F^{\mathcal{I}} \wedge r(v)\}$

Intuitively, a datatype restriction is used to specify a subset of the numerical domain so that one can form new concepts that refer to elements of this subset. The set of concepts is recursively defined using the constructors listed in the middle column of Table 1, where C and D are concepts, $R \in N_R$, $F \in N_F$ and r is a $\mathcal{D}$-datatype restriction. We typically use the capital letters A, B to refer to concept names and the capital letters C, D or E to refer to concepts. We also set the abbreviations $N_C^{\top} = N_C \cup \{\top\}$ and $N_C^{\top,\bot} = N_C \cup \{\top, \bot\}$.

An *axiom* α *in* $\mathcal{EL}^{\bot}(\mathcal{D})$ or simply *an axiom* α is an expression of the form $C \sqsubseteq D$, where C and D are concepts. An $\mathcal{EL}^{\bot}(\mathcal{D})$-*ontology* $\mathcal{O}$ or simply *an ontology* $\mathcal{O}$ is a set of axioms. We say that a concept E occurs in a concept C iff E is used as a concept in the construction of C. Moreover, a concept E is said to *positively (negatively) occur* in an axiom $C \sqsubseteq D$ iff it occurs in the concept D (C); we alternatively say that we have a *positive (negative) occurrence* of E.

An interpretation of $\mathcal{EL}^{\bot}(\mathcal{D})$ is a pair $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$, where $\Delta^{\mathcal{I}}$ is a non-empty set which we call the *domain of the interpretation* and $\cdot^{\mathcal{I}}$ is the *interpretation function*. The interpretation function maps each concept name A to a subset $A^{\mathcal{I}}$ of $\Delta^{\mathcal{I}}$, each role name $R \in N_R$ to a relation $R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$ and each feature name $F \in N_F$ to a relation $F^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \mathcal{D}$. Note that we do not require the interpretation of features to be functional. In this respect, they correspond to the data properties in OWL 2 [2]. The constructors of $\mathcal{EL}^{\bot}(\mathcal{D})$ are interpreted as indicated in the right column of Table 1. For an axiom α, where $\alpha = C \sqsubseteq D$, we write $\mathcal{I} \models \alpha$ and we say that *an interpretation* $\mathcal{I}$ *satisfies an axiom* α, iff $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$. If $\mathcal{I} \models \alpha$ for every $\alpha \in \mathcal{O}$, then $\mathcal{I}$ *is a model of* $\mathcal{O}$ and we write $\mathcal{I} \models \mathcal{O}$. Additionally, if every model $\mathcal{I}$ of $\mathcal{O}$ satisfies the axiom α then we say that $\mathcal{O}$ *entails* α and we write $\mathcal{O} \models \alpha$. We define the *signature of an ontology* $\mathcal{O}$ as the set $sig(\mathcal{O})$ of concept, role and feature names that occur in $\mathcal{O}$. We say that a concept, role or feature name X is *fresh w.r.t. an ontology* $\mathcal{O}$ iff $X \notin sig(\mathcal{O})$.

One of the most common reasoning tasks w.r.t. an ontology $\mathcal{O}$ is the classification of an ontology $\mathcal{O}$, that is computing all axioms of the form $A \sqsubseteq B$, where A, $B \in N_C^{\top,\bot}$ and $\mathcal{O} \models A \sqsubseteq B$. The set of these axioms is called the *taxonomy* of the ontology $\mathcal{O}$.

We say that an axiom in $\mathcal{EL}^{\bot}(\mathcal{D})$ is in normal form if it has one of the forms NF1-NF6 of the left part of Table 2, where A, A_1, A_2, $B \in N_C^{\top}$, $B' \in N_C^{\top,\bot}$, $R \in N_R$, $F \in N_F$ and r is a $\mathcal{D}$-datatype restriction. Given an $\mathcal{EL}^{\bot}(\mathcal{D})$-ontology, if the normalization rules of the right part of Table 2 are applied, we obtain an

Table 2. Normal form of axioms and normalization rules for $\mathcal{EL}^{\perp}(\mathcal{D})$

Normal forms		Normalization rules
NF1	$A \sqsubseteq B'$	$C \sqcap H \sqsubseteq E \rightarrow \{H \sqsubseteq A_f, C \sqcap A_f \sqsubseteq E\}$
NF2	$A_1 \sqcap A_2 \sqsubseteq B$	$\exists R.G \sqsubseteq D \rightarrow \{G \sqsubseteq A_f, \exists R.A_f \sqsubseteq D\}$
NF3	$A \sqsubseteq \exists R.B$	$G \sqsubseteq H \rightarrow \{G \sqsubseteq A_f, A_f \sqsubseteq H\}$
NF4	$\exists R.B \sqsubseteq A$	$C \sqsubseteq \exists R.H \rightarrow \{C \sqsubseteq \exists R.A_f, A_f \sqsubseteq H\}$
NF5	$A \sqsubseteq \exists F.r$	$B \sqsubseteq C \sqcap D \rightarrow \{B \sqsubseteq C, B \sqsubseteq D\}$
NF6	$\exists F.r \sqsubseteq A$	$\perp \sqsubseteq C \rightarrow \emptyset$

ontology which contains only axioms in normal form [3]. For the rules of Table 2, we have that $B \in N_C^{\top}$, $G, H \notin N_C^{\top}$, $R \in N_R$, C, D, E, G and H are concepts and A_f is a fresh concept name w.r.t. the so far transformed ontology.

3 Numerical Datatypes with Restrictions

In this section we introduce the notion of a Numerical Datatype with Restrictions (NDR) which specifies which datatype relations can be used positively and negatively. We then present a polynomial consequence-based classification procedure for $\mathcal{EL}^{\perp}$ extended with NDRs and prove its soundness. Finally we prove that the procedure is complete provided that the NDR satisfies special safety requirements.

Definition 2 (Numerical Datatype with Restrictions). *A* numerical datatype with restrictions (NDR) *is a triple* $(\mathcal{D}, O_+, O_-)$*, where* $\mathcal{D} \subseteq \mathbb{R}$ *is a numerical domain and* $O_+, O_- \subseteq \{<, \leq, >, \geq, =\}$ *is the set of* positive *and, respectively,* negative relations*. An axiom in* $\mathcal{EL}^{\perp}(\mathcal{D})$ *is an axiom in* $\mathcal{EL}^{\perp}(\mathcal{D}, O_+, O_-)$ *if for every positive (negative) occurrence of a concept* $\exists F.(s, y)$ *in the axiom,* $s \in O_+$ *(*$s \in O_-$*). An* $\mathcal{EL}^{\perp}(\mathcal{D}, O_+, O_-)$-ontology *is a set of axioms in* $\mathcal{EL}^{\perp}(\mathcal{D}, O_+, O_-)$*.*

Subsequently, we describe when a datatype restriction is inconsistent and when one datatype restriction implies another (w.r.t. a domain $\mathcal{D}$). These definitions of inconsistency and implication for datatype restrictions are necessary for the formulation of the inference rules, which we then briefly present.

3.1 The Classification Procedure and Soundness

We are going to describe a classification procedure for $\mathcal{EL}^{\perp}(\mathcal{D}, O_+, O_-)$, which is closely related to the procedure for $\mathcal{EL}^{++}$ [3]. In order to formulate inference rules for datatypes we need to introduce notation for satisfiability of a datatype restriction and implication between datatype restrictions.

Definition 3. *For two* $\mathcal{D}$*-datatype restrictions* r_+ *and* r_-*, we write* $r_+ \rightarrow_{\mathcal{D}} \perp$ *iff there is no* $x \in \mathcal{D}$ *such that* $r_+(x)$ *holds. Otherwise, we write* $r_+ \not\rightarrow_{\mathcal{D}} \perp$*. We write that* $r_+ \rightarrow_{\mathcal{D}} r_-$ *iff* $r_+(x)$ *implies* $r_-(x)$*,* $\forall x \in \mathcal{D}$*. Otherwise, we write* $r_+ \not\rightarrow_{\mathcal{D}} r_-$*.*

Table 3. Reasoning rules in $\mathcal{EL}^{\perp}(\mathcal{D})$

IR1	$\dfrac{}{A \sqsubseteq A}$		**IR2**	$\dfrac{}{A \sqsubseteq \top}$		**CR1**	$\dfrac{A \sqsubseteq B}{A \sqsubseteq C'}$	$B \sqsubseteq C' \in \mathcal{O}$
CR2	$\dfrac{A \sqsubseteq B \quad A \sqsubseteq C}{A \sqsubseteq D}$	$B \sqcap C \sqsubseteq D \in \mathcal{O}$				**CR3**	$\dfrac{A \sqsubseteq B}{A \sqsubseteq \exists R.C}$	$B \sqsubseteq \exists R.C \in \mathcal{O}$
CR4	$\dfrac{A \sqsubseteq \exists R.B \quad B \sqsubseteq C}{A \sqsubseteq D}$	$\exists R.C \sqsubseteq D \in \mathcal{O}$				**CR5**	$\dfrac{A \sqsubseteq \exists R.B \quad B \sqsubseteq \bot}{A \sqsubseteq \bot}$	

ID1	$\dfrac{}{A \sqsubseteq \bot}$	$A \sqsubseteq \exists F.r_+ \in \mathcal{O}\ ,\ r_+ \rightarrow_{\mathcal{D}} \bot$	$A, B, C, D \in N_C^{\top}$
CD1	$\dfrac{A \sqsubseteq \exists F.r_+}{A \sqsubseteq B}$	$\exists F.r_- \sqsubseteq B \in \mathcal{O}\ ,\ r_+ \rightarrow_{\mathcal{D}} r_-$	$C' \in N_C^{\top,\bot}$
CD2	$\dfrac{A \sqsubseteq B}{A \sqsubseteq \exists F.r_+}$	$B \sqsubseteq \exists F.r_+ \in \mathcal{O}$	$R \in N_R, F \in N_F$

We assume that deciding whether $r_+ \rightarrow_{\mathcal{D}} \bot$ and $r_+ \rightarrow_{\mathcal{D}} r_-$ can be done in polynomial time. It is easy to see that this is the case when $\mathcal{D}$ is the set of natural numbers, integers, reals or rationals for the set of relations $\{<, \leq, >, \geq, =\}$.

The classification procedure for $\mathcal{EL}^{\perp}(\mathcal{D})$ takes as input an $\mathcal{EL}^{\perp}(\mathcal{D})$-ontology $\mathcal{O}$ whose axioms are in normal form and applies the inference rules in Table 3 to derive new axioms of the form NF1, NF3 and NF5 in Table 2. The rules are applied to already derived axioms and use axioms in $\mathcal{O}$ and properties $r_+ \rightarrow_{\mathcal{D}} \bot$ and $r_+ \rightarrow_{\mathcal{D}} r_-$ as side-conditions. The procedure terminates when no new axiom can be derived. It is easy to see that the procedure runs in polynomial time because there are only polynomially many axioms of the form NF1, NF3 and NF5 possible over $sig(\mathcal{O})$. It can be easily proved that the procedure is sound because the rules derive logical consequences of the axioms.

Theorem 1 (Soundness). *Let $\mathcal{O}$ be an $\mathcal{EL}^{\perp}(\mathcal{D})$-ontology consisting of axioms in normal form and $\mathcal{O}'$ consists of all axioms that are derivable using the rules of Table 3 for $\mathcal{O}$. Every model $\mathcal{I}$ of $\mathcal{O}$ is a model of $\mathcal{O}'$ as well.*

Proof. For every axiom $\alpha \in \mathcal{O}'$, we prove that $\mathcal{I} \models \alpha$ by induction on the length of the derivation of α.

Induction base: If α is obtained using rules **IR1** and **IR2** then $\mathcal{I} \models \alpha$ trivially. Suppose that $\alpha = A \sqsubseteq \bot$ is obtained using rule **ID1**. In this case, $A \sqsubseteq \exists F.r_+ \in \mathcal{O}$ and since $\mathcal{I} \models \mathcal{O}$ then $A^{\mathcal{I}} \subseteq (\exists F.r_+)^{\mathcal{I}}$. Since $r_+ \rightarrow_{\mathcal{D}} \bot$ we have $(\exists F.r_+)^{\mathcal{I}} = \emptyset$. Therefore, $A^{\mathcal{I}} \subseteq \emptyset$ and so $\mathcal{I} \models A \sqsubseteq \bot$.

Induction step: For the cases when axiom α is obtained using rules **CR1-CR5** (that do not involve datatypes) the proof is identical with the case of $\mathcal{EL}^{++}$ [3]. Suppose that $\alpha = A \sqsubseteq B$ is obtained using **CD1** from $A \sqsubseteq \exists F.r_+$. Then by induction hypothesis, $A^{\mathcal{I}} \subseteq (\exists F.r_+)^{\mathcal{I}}$. Since $\mathcal{I} \models \mathcal{O}$, $(\exists F.r_-)^{\mathcal{I}} \subseteq B^{\mathcal{I}}$ and from $r_+ \rightarrow_{\mathcal{D}} r_-$, we have that $A^{\mathcal{I}} \subseteq B^{\mathcal{I}}$. So, $\mathcal{I} \models A \sqsubseteq B$. Suppose that $\alpha = A \sqsubseteq \exists F.r_+$

is obtained using **CD2** from $A \sqsubseteq B$. Then by induction hypothesis, $A^{\mathcal{I}} \subseteq B^{\mathcal{I}}$. Since $\mathcal{I} \models \mathcal{O}$, $B^{\mathcal{I}} \subseteq (\exists F.r_+)^{\mathcal{I}}$ and, so, $A^{\mathcal{I}} \subseteq (\exists F.r_+)^{\mathcal{I}}$. So, $\mathcal{I} \models A \sqsubseteq \exists F.r_+$.

3.2 Completeness and Safe NDRs

The completeness proof is based on the canonical model construction similarly as for $\mathcal{EL}^{++}$ [3]. In order to deal with datatypes in the canonical model we introduce a notion of a datatype *constraint*. Intuitively, a constraint specifies which datatype restrictions should hold in a model and which should not.

Definition 4 (Constraint). *A* constraint *over* $(\mathcal{D}, O_+, O_-)$ *is defined as a pair of sets* (S_+, S_-)*, such that* $S_+ = \{(s_+^1, y_1), \ldots, (s_+^n, y_n)\}$ *with* $s_+^i \in O_+$*,* $S_- = \{(s_-^1, z_1), \ldots, (s_-^m, z_m)\}$ *with* $s_-^j \in O_-$*,* y_i*,* $z_j \in \mathcal{D}$*,* $(s_+^i, y_i) \nrightarrow_{\mathcal{D}} (s_-^j, z_j)$ *and* $(s_+^i, y_i) \nrightarrow_{\mathcal{D}} \bot$ *for* $1 \leq i \leq n$*,* $1 \leq j \leq m$ *and* m*,* $n \geq 0$*.*

Definition 5. *A constraint* (S_+, S_-) *over* $(\mathcal{D}, O_+, O_-)$ *is* satisfiable *iff there exists a* solution of (S_+, S_-) *that is a set* $V \subseteq \mathcal{D}$ *such that every* $r_+ \in S_+$ *is satisfied by at least one* $v \in V$ *but no* $r_- \in S_-$ *is satisfied by any* $v \in V$*.*

Our model construction procedure works only for the cases where we can ensure that every constraint over a numerical domain is satisfiable. This leads us to a notion of safety for an NDR.

Definition 6 (NDR Safety). *Let* $(\mathcal{D}, O_+, O_-)$ *be an NDR.* $(\mathcal{D}, O_+, O_-)$ *is* safe *iff every constraint over* $(\mathcal{D}, O_+, O_-)$ *is satisfiable.*

We define strong and weak convexity for NDRs and prove that an NDR is safe iff it is weakly convex.

Definition 7 (Strong and Weak Convexity). *The NDR* $(\mathcal{D}, O_+, O_-)$ *is* strongly convex *when for every* $r_+^i = (s_+^i, y_i)$ *and* $r_-^j = (s_-^j, z_j)$*, with* $s_+^i \in O_+$*,* $s_-^j \in O_-$ *and* y_i*,* $z_j \in \mathcal{D}$ *(*$1 \leq i \leq n$*,* $1 \leq j \leq m$*), if* $\bigwedge_{i=1}^n r_+^i \rightarrow_{\mathcal{D}} \bigvee_{j=1}^m r_-^j$*, then there exists an* r_-^j *(*$1 \leq j \leq m$*) such that* $\bigwedge_{i=1}^n r_+^i \rightarrow_{\mathcal{D}} r_-^j$*.* $(\mathcal{D}, O_+, O_-)$ *is* weakly convex *when the implication holds for* $n = 1$*.*

For example the NDR $(\mathbb{Z}, \{<, >\}, \{=\})$ is weakly convex but not strongly convex. It is weakly convex since the implications $((<, y) \rightarrow_{\mathbb{Z}} \bigvee_{j=1}^m (=, z_j))$ and $((>, y) \rightarrow_{\mathbb{Z}} \bigvee_{j=1}^m (=, z_j))$ never hold. However, it is not strongly convex: it is $(>, 2) \wedge (<, 5) \rightarrow_{\mathbb{Z}} (=, 3) \vee (=, 4)$, but also $(>, 2) \wedge (<, 5) \nrightarrow_{\mathbb{Z}} (=, 3)$ and $(>, 2) \wedge (<, 5) \nrightarrow_{\mathbb{Z}} (=, 4)$.

Lemma 1. $(\mathcal{D}, O_+, O_-)$ *is safe iff it is weakly convex.*

Proof. We assume that $(\mathcal{D}, O_+, O_-)$ is not weakly convex and we prove that it is non-safe. Since it is not weakly convex we have that for some $r_+ \rightarrow_{\mathcal{D}} \bigvee_{j=1}^m r_-^j$ there exists no r_-^j such that $r_+ \rightarrow_{\mathcal{D}} r_-^j$. In order to prove non-safety it is sufficient to define a constraint which is not satisfiable. We define (S_+, S_-), with $S_+ = \{r_+\}$ and $S_- = \{r_-^j\}_{j=1}^m$. (S_+, S_-) is indeed a constraint because $r_+ \nrightarrow_{\mathcal{D}} \bot$ (otherwise $r_+ \rightarrow_{\mathcal{D}} r_-^j$ is true for every r_-^j) and for every r_-^j, $r_+ \nrightarrow_{\mathcal{D}} r_-^j$ (otherwise $r_+ \rightarrow_{\mathcal{D}} r_-^j$ is true for at least one r_-^j). Additionally, it is not satisfiable, because from $r_+ \rightarrow_{\mathcal{D}} \bigvee_{j=1}^m r_-^j$ there can be found no x such that $r_+(x)$ and $\bigwedge_{j=1}^m \neg r_-^j(x)$.

We prove that if $(\mathcal{D}, O_+, O_-)$ is not safe, then it is not weakly convex. Since it is not safe then there exists a non-satisfiable constraint (S_+, S_-), where $S_+ = \{r_+^i\}_{i=1}^n$ and $S_- = \{r_-^j\}_{j=1}^m$. If $S_- = \emptyset$, then since $r_+^i \nrightarrow_{\mathcal{D}} \bot$ for $1 \leq i \leq n$, there is a solution $V = \{x_i \mid 1 \leq i \leq n\}$ for (S_+, S_-). Thus, $S_- \neq \emptyset$. If $S_+ = \emptyset$ then there is the solution $V = \emptyset$ for (S_+, S_-). Thus, $S_+ \neq \emptyset$. Since (S_+, S_-) is a constraint, then $r_+^i \nrightarrow_{\mathcal{D}} r_-^j$ for $1 \leq i \leq n$ and $1 \leq j \leq m$. Since (S_+, S_-) is not satisfiable for every $1 \leq i \leq n$ there exists no x such that $r_+^i(x)$ and $\bigwedge_{j=1}^m \neg r_-^j(x)$, that is if $r_+^i(x)$ then $r_-^j(x)$ holds for at least one r_-^j or, otherwise written, $r_+^i \rightarrow_{\mathcal{D}} \bigvee_{j=1}^m r_-^j$. From this and $r_+^i \nrightarrow_{\mathcal{D}} r_-^j$ for every r_-^j, $(\mathcal{D}, O_+, O_-)$ is not weakly convex. □

Theorem 2 (Completeness). *Let $(\mathcal{D}, O_+, O_-)$ be a safe NDR, let $\mathcal{O}$ be an $\mathcal{EL}^\bot(\mathcal{D}, O_+, O_-)$-ontology containing axioms in normal form and let $\mathcal{O}'$ be the saturation of $\mathcal{O}$ under the rules of Table 3. For every A, $B \in (N_C^\top \cap sig(\mathcal{O}))$, if $\mathcal{O} \models A \sqsubseteq B$, then $A \sqsubseteq B \in \mathcal{O}'$ or $A \sqsubseteq \bot \in \mathcal{O}'$.*

Proof. The proof is analogous to the completeness proof of the subsumption algorithm for $\mathcal{EL}^{++}$ [3]; we build a canonical model $\mathcal{I}$ for $\mathcal{O}$ using $\mathcal{O}'$ and show that if $A \not\sqsubseteq B \in \mathcal{O}'$ and $A \not\sqsubseteq \bot \in \mathcal{O}'$ then $\mathcal{I} \not\models A \sqsubseteq B$.
For every $A \in N_C$, $F \in N_F$, define $S_+(A, F)$ and $S_-(A, F)$, as follows:

$$S_+(A, F) = \{r_+ \mid A \sqsubseteq \exists F.r_+ \in \mathcal{O}', A \sqsubseteq \bot \notin \mathcal{O}'\} \tag{3}$$

$$S_-(A, F) = \{r_- \mid \exists F.r_- \sqsubseteq B \in \mathcal{O}, A \sqsubseteq B \notin \mathcal{O}'\} \tag{4}$$

We now show that $(S_+(A, F), S_-(A, F))$ is a constraint over $(\mathcal{D}, O_+, O_-)$. First we prove that $r_+ \nrightarrow_{\mathcal{D}} \bot$, $\forall r_+ \in S_+(A, F)$, which is true because otherwise due to rule **ID1** it would be $A \sqsubseteq \bot \in \mathcal{O}'$, in contradiction to the definition of $S_+(A, F)$. Additionally, there is no $r_+ \in S_+(A, F)$ and $r_- \in S_-(A, F)$ such that $r_+ \rightarrow_{\mathcal{D}} r_-$, otherwise from $A \sqsubseteq \exists F.r_+ \in \mathcal{O}'$, $\exists F.r_- \sqsubseteq B \in \mathcal{O}$ and **CD1** it would be $A \sqsubseteq B \in \mathcal{O}'$ which contradicts the definition of $S_-(A, F)$. Since $(S_+(A, F), S_-(A, F))$ is a constraint over $(\mathcal{D}, O_+, O_-)$ and $(\mathcal{D}, O_+, O_-)$ is safe, there exists a solution $V(A, F) \subseteq \mathcal{D}$ of $(S_+(A, F), S_-(A, F))$. We now construct the canonical model $\mathcal{I}$:

$$\Delta^{\mathcal{I}} = \{x_A \mid A \in (N_C^\top \cap sig(\mathcal{O})), A \sqsubseteq \bot \notin \mathcal{O}'\} \tag{5}$$

$$B^{\mathcal{I}} = \{x_A \mid x_A \in \Delta^{\mathcal{I}}, A \sqsubseteq B \in \mathcal{O}'\} \tag{6}$$

$$R^{\mathcal{I}} = \{(x_A, x_B) \mid A \sqsubseteq \exists R.B \in \mathcal{O}', x_A, x_B \in \Delta^{\mathcal{I}}\} \tag{7}$$

$$F^{\mathcal{I}} = \{(x_A, v) \mid v \in V(A, F)\} \tag{8}$$

We prove that $\mathcal{I} \models \mathcal{O}$ by showing that $\mathcal{I} \models \alpha$, when α takes one of the NF1-NF6.

NF1 $A \sqsubseteq B$: We need to prove $A^{\mathcal{I}} \subseteq B^{\mathcal{I}}$. Take an $x \in A^{\mathcal{I}}$. By (6), $x = x_C$ such that $C \sqsubseteq A \in \mathcal{O}'$. From $A \sqsubseteq B \in \mathcal{O}$ and since $\mathcal{O}'$ is closed under **CR1**, we have $C \sqsubseteq B \in \mathcal{O}'$. Hence $x = x_C \in B^{\mathcal{I}}$ by (6).

If $B = \bot$, then we need to show that $A^{\mathcal{I}} = \emptyset$. If there exists $x \in A^{\mathcal{I}}$, then by (6) $x = x_C$ such that $C \sqsubseteq A \in \mathcal{O}'$. Since $\mathcal{O}'$ is closed under **CR1** and $A \sqsubseteq \bot \in \mathcal{O}'$, we have $C \sqsubseteq \bot \in \mathcal{O}'$. Thus, $x = x_C \notin \Delta^{\mathcal{I}}$ by (5), which contradicts our assumption that $x \in A^{\mathcal{I}}$.

We examine separately the case when $A = \top$. We have that $x_A \in \Delta^{\mathcal{I}}$ and we need to show that $x_A \in B^{\mathcal{I}}$. From rule **IR2**, we have that $A \sqsubseteq \top \in \mathcal{O}'$. From rule **CR1**, $A \sqsubseteq B \in \mathcal{O}'$; since $x_A \in \Delta^{\mathcal{I}}$ and $A \sqsubseteq B \in \mathcal{O}'$ we get $x_A \in B^{\mathcal{I}}$ by (6).

NF2 $A_1 \sqcap A_2 \sqsubseteq B$: We prove $(A_1 \sqcap A_2)^{\mathcal{I}} \subseteq B^{\mathcal{I}}$. Take an $x \in (A_1 \sqcap A_2)^{\mathcal{I}}$; then, $x \in A_1^{\mathcal{I}}$, $x \in A_2^{\mathcal{I}}$ and by (6) $x = x_A$ for some concept name A such that $A \sqsubseteq A_1 \in \mathcal{O}'$ and $A \sqsubseteq A_2 \in \mathcal{O}'$. Since $A \sqsubseteq A_1 \in \mathcal{O}'$, $A \sqsubseteq A_2 \in \mathcal{O}'$ and $A_1 \sqcap A_2 \sqsubseteq B \in \mathcal{O}$ closure under rule **CR2** gives $A \sqsubseteq B \in \mathcal{O}'$ and, therefore, $x \in B^{\mathcal{I}}$, by (6).

NF3 $A \sqsubseteq \exists R.B$: We show $A^{\mathcal{I}} \subseteq (\exists R.B)^{\mathcal{I}}$; take an $x \in A^{\mathcal{I}}$. By (6), $x = x_C$ where $C \sqsubseteq A \in \mathcal{O}'$. Since $A \sqsubseteq \exists R.B \in \mathcal{O}$ and $\mathcal{O}'$ is closed under **CR3**, we have $C \sqsubseteq \exists R.B \in \mathcal{O}'$. Since $x_C \in \Delta^{\mathcal{I}}$, we have $C \sqsubseteq \bot \notin \mathcal{O}'$ and, hence, $B \sqsubseteq \bot \notin \mathcal{O}'$ by **CR5**. Thus, $x_B \in \Delta^{\mathcal{I}}$ and $(x_C, x_B) \in R^{\mathcal{I}}$ by (7). Since $B \sqsubseteq B \in \mathcal{O}'$ by **IR1**, we have $x_B \in B^{\mathcal{I}}$ by (6). Thus, $x = x_C \in (\exists R.B)^{\mathcal{I}}$.

NF4 $\exists R.B \sqsubseteq A$: We prove $(\exists R.B)^{\mathcal{I}} \subseteq A^{\mathcal{I}}$; take an $x \in (\exists R.B)^{\mathcal{I}}$. Then, there exists $y \in \Delta^{\mathcal{I}}$ such that $(x, y) \in R^{\mathcal{I}}$ and $y \in B^{\mathcal{I}}$. By (7) and (6) $x = x_C$ and $y = x_D$ such that $C \sqsubseteq \exists R.D \in \mathcal{O}'$ and $D \sqsubseteq B \in \mathcal{O}'$ respectively. Since $\exists R.B \sqsubseteq A \in \mathcal{O}$ and $\mathcal{O}'$ is closed under **CR4**, we have $C \sqsubseteq A \in \mathcal{O}'$. Hence, $x = x_C \in A^{\mathcal{I}}$ by (6).

NF5 $A \sqsubseteq \exists F.r_+$: We show that $A^{\mathcal{I}} \subseteq (\exists F.r_+)^{\mathcal{I}}$; take an $x \in A^{\mathcal{I}}$. By (6), there exists a concept name C such that $x = x_C$ and $C \sqsubseteq A \in \mathcal{O}'$. Since $A \sqsubseteq \exists F.r_+ \in \mathcal{O}$ and $\mathcal{O}'$ is closed under **CD2**, we have $C \sqsubseteq \exists F.r_+ \in \mathcal{O}'$. We use (3) and (4) to build $(S_+(C, F), S_-(C, F))$; we have $r_+ \in S_+(C, F)$. By (8) we have $(x_C, v) \in F^{\mathcal{I}}$ for every $v \in V(C, F)$. Since $r_+ \in S_+(C, F)$, there exists $v \in V(C, F)$ such that v satisfies r_+ and, hence, $x = x_C \in (\exists F.r_+)^{\mathcal{I}}$.

NF6 $\exists F.r_- \sqsubseteq B$: We prove that $(\exists F.r_-)^{\mathcal{I}} \subseteq B^{\mathcal{I}}$; take an $x \in (\exists F.r_-)^{\mathcal{I}}$. By (5), there exists a concept name C such that $x = x_C$. We use (3) and (4) and construct $(S_+(C, F), S_-(C, F))$. Since $x_C \in (\exists F.r_-)^{\mathcal{I}}$, by (8), there exists $v \in V(C, F)$, such that $r_-(v)$ and $V(C, F)$ is a solution for $(S_+(C, F), S_-(C, F))$. Hence, $r_- \notin S_-(C, F)$, and so, $C \sqsubseteq B \in \mathcal{O}'$ by (4). Now by (6) and $C \sqsubseteq B \in \mathcal{O}'$, we have that $x_C \in B^{\mathcal{I}}$.

We now show that if $A \sqsubseteq B \notin \mathcal{O}'$ and $A \sqsubseteq \bot \notin \mathcal{O}'$, then $\mathcal{O} \not\models A \sqsubseteq B$ by proving $\mathcal{I} \not\models A \sqsubseteq B$ (since $\mathcal{I} \models \mathcal{O}$). $A^{\mathcal{I}} \not\subseteq B^{\mathcal{I}}$ holds, because $x_A \in \Delta^{\mathcal{I}}$ (from $A \sqsubseteq \bot \notin \mathcal{O}'$ and (5)), $x_A \in A^{\mathcal{I}}$ (from $A \sqsubseteq A \in \mathcal{O}'$ using rule **IR1** and (6)) and $x_A \notin B^{\mathcal{I}}$ (from $A \sqsubseteq B \notin \mathcal{O}'$ and (6)). □

4 Maximal Safe NDRs for $\mathbb{N}$

In this section we present a full classification of safe NDRs for natural numbers; for the current section we assume that every constraint is over the domain $\mathbb{N}$

Table 4. Maximal safe NDRs for $\mathbb{N}$

NDR	O_+	O_-
NDR$_1$	$\{=\}$	$\{<, \leq, >, \geq, =\}$
NDR$_2$	$\{<, \leq, >, \geq, =\}$	$\{<, \leq\}$
NDR$_3$	$\{<, \leq, >, \geq, =\}$	$\{>, \geq\}$
NDR$_4$	$\{>, \geq, =\}$	$\{<, \leq, =\}$

Table 5. Transformations $C_1 \Rightarrow C_2$ preserving constraints and their satisfiability for $\mathbb{N}$, where S_-, S_+ and S are sets of datatype restrictions and $y_1 \leq y_2$, $z_1 \leq z_2$

$C_1 = (S \cup S_+^1, S_-)$, $C_2 = (S \cup S_+^2, S_-)$		$C_1 = (S_+, S \cup S_-^1)$, $C_2 = (S_+, S \cup S_-^2)$	
S_+^1	S_+^2	S_-^1	S_-^2
$\{(<, y)\}$	$\{(\leq, y-1)\}$	$\{(<, z)\}$	$\{(\leq, z-1)\}$
$\{(>, y)\}$	$\{(\geq, y+1)\}$	$\{(>, z)\}$	$\{(\geq, z+1)\}$
$\{(\leq, y_1), (\leq, y_2)\}$	$\{(\leq, y_1)\}$	$\{(\leq, z_1), (\leq, z_2)\}$	$\{(\leq, z_2)\}$
$\{(\geq, y_1), (\geq, y_2)\}$	$\{(\geq, y_2)\}$	$\{(\geq, z_1), (\geq, z_2)\}$	$\{(\geq, z_1)\}$
$\{(=, y_1), (\leq, y_2)\}$	$\{(=, y_1)\}$	$\{(=, z_1), (\leq, z_2)\}$	$\{(\leq, z_2)\}$
$\{(\geq, y_1), (=, y_2)\}$	$\{(=, y_2)\}$	$\{(\geq, z_1), (=, z_2)\}$	$\{(\geq, z_1)\}$
		$\{(<, 0)\}$	$\emptyset$

($0 \in \mathbb{N}$). Table 4 lists all maximal safe NDRs for $\mathbb{N}$. We prove that: (i) every NDR in Table 4 is safe, (ii) extending any of these NDRs with a new relation leads to non-safety and (iii) every safe NDR is contained in some NDR in Table 4.

Table 5 presents some basic transformations that preserve satisfiability of constraints.

Lemma 2. *Let C_1 and C_2 be as defined in Table 5 and $(\mathbb{N}, O_+, O_-)$ be an NDR. Then (i) C_1 is a constraint over $(\mathbb{N}, O_+, O_-)$ iff C_2 is a constraint over $(\mathbb{N}, O_+, O_-)$ and (ii) if C_1 and C_2 are both constraints over $(\mathbb{N}, O_+, O_-)$, then C_1 is satisfiable iff C_2 is satisfiable.*

Corollary 1. *Let $\mathsf{NDR}_i = (\mathbb{N}, O_+^i, O_-^i)$, with $1 \leq i \leq 4$. For every (S_+^1, S_-^1) over NDR_i there exists a constraint (S_+^2, S_-^2) over NDR_i, $y_1, \ldots, y_n \in \mathbb{N}$ and $z_1, \ldots, z_m \in \mathbb{N}$ such that:*

$$\begin{aligned} S_+^2 &\subseteq \{(\leq, y_1), (=, y_2), \ldots, (=, y_{n-1}), (\geq, y_n)\} \\ S_-^2 &\subseteq \{(\leq, z_1), (=, z_2), \ldots, (=, z_{m-1}), (\geq, z_m)\} \end{aligned}$$

where $y_1 < \ldots < y_n$, $z_1 < \ldots < z_m$, $z_1 < y_1$, $z_m > y_n$, $y_i \neq z_j$ $(2 \leq i \leq n-1$, $2 \leq j \leq m-1$, m, $n \geq 0)$ and (S_+^1, S_-^1) over NDR_i is satisfiable iff (S_+^2, S_-^2) over NDR_i is satisfiable.

The proof of Lemma 2 and Corollary 1 is trivial by a routine check of all cases.

Lemma 3. *Every NDR in Table 4 is safe.*

Table 6. Examples of non-safe NDRs for $\mathbb{N}$ where $(s_+, y) \rightarrow_{\mathbb{N}} (s_-^1, z_1) \vee (s_-^2, z_2)$, $(s_+, y) \nrightarrow_{\mathbb{N}} (s_-^1, z_1)$ and $(s_+, y) \nrightarrow_{\mathbb{N}} (s_-^2, z_2)$

$\{s_+\}$	$\{s_-^1, s_-^2\}$	y	z_1	z_2
$\{<\}$, $\{\leq\}$	$\{<, \geq\}$, $\{\leq, >\}$, $\{\leq, \geq\}$	3	1	1
$\{<\}$, $\{\leq\}$	$\{<, >\}$	3	2	1
$\{>\}$, $\{\geq\}$	$\{<, \geq\}$, $\{\leq, >\}$, $\{\leq, \geq\}$	1	3	3
$\{>\}$, $\{\geq\}$	$\{<, >\}$	1	3	2
$\{>\}$	$\{=, \geq\}$	1	2	3
$\{>\}$	$\{=, >\}$	1	2	2
$\{\geq\}$	$\{=, \geq\}$	1	1	2
$\{\geq\}$	$\{=, >\}$	1	1	1
$\{<\}$	$\{=\}$	3	1	2
$\{\leq\}$	$\{=\}$	2	1	2

Proof. We prove Lemma 3 by building a solution V for every constraint over NDRs in Table 4. By Corollary 1 we can assume w.l.o.g. the following restrictions for (S_+, S_-) and construct the corresponding solution V:

NDR_1: For S_+ we have that $S_+ \subseteq \{(=, y_1), \ldots, (=, y_n)\}$ and for S_- that $S_- \subseteq \{(\leq, z_1), (=, z_2), \ldots, (=, z_{m-1}), (\geq, z_m)\}$ with $z_1 < y_1 < \ldots < y_n < z_m$, $z_1 < \ldots < z_m$ and $y_i \neq z_j$ $(1 \leq i \leq n,\ 2 \leq j \leq m-1)$. $V = \{y_1, \ldots, y_n\}$.

NDR_2: For S_+ we have that $S_+ \subseteq \{(\leq, y_1), (=, y_2), \ldots, (=, y_{n-1}), (\geq, y_n)\}$ and for S_- that $S_- \subseteq \{(\leq, z_1)\}$ with $z_1 < y_1 < \ldots < y_n$. $V = \{y_1, \ldots, y_n\}$.

NDR_3: For S_+ we have that $S_+ \subseteq \{(\leq, y_1), (=, y_2), \ldots, (=, y_{n-1}), (\geq, y_n)\}$ and for S_- that $S_- \subseteq \{(\geq, z_1)\}$ with $y_1 < \ldots < y_n < z_1$. $V = \{y_1, \ldots, y_n\}$.

NDR_4: For S_+ we have that $S_+ \subseteq \{(=, y_1), \ldots, (=, y_{n-1}), (\geq, y_n)\}$ and for S_- that $S_- \subseteq \{(\leq, z_1), (=, z_2), \ldots, (=, z_m)\}$ with $y_1 < \ldots < y_n$, $z_1 < \ldots < z_m$, $z_1 < y_1$ and $y_i \neq z_j$ $(1 \leq i \leq n-1,\ 2 \leq j \leq m)$. $V = \{y_1, \ldots, y_{n-1}, y_n'\}$, where $y_n' = \max(y_n, z_m) + 1$. □

Lemma 4. *Let* $(\mathbb{N}, O_+, O_-)$ *be an NDR. Then:*

(a) If $O_+ \cap \{<, \leq, >, \geq\} \neq \emptyset$, $O_- \cap \{<, \leq\} \neq \emptyset$ *and* $O_- \cap \{>, \geq\} \neq \emptyset$, *then* $(\mathbb{N}, O_+, O_-)$ *is non-safe.*
(b) If $O_+ \cap \{>, \geq\} \neq \emptyset$, $O_- \cap \{>, \geq\} \neq \emptyset$ *and* $\{=\} \subseteq O_-$, *then* $(\mathbb{N}, O_+, O_-)$ *is non-safe.*
(c) If $O_+ \cap \{<, \leq\} \neq \emptyset$ *and* $\{=\} \subseteq O_-$, *then* $(\mathbb{N}, O_+, O_-)$ *is non-safe.*

Proof. In order to prove that the NDR is non-safe it suffices, from Lemma 1 to prove that it is not weakly convex. We provide restrictions (s_+, y), (s_-^1, z_1) and (s_-^2, z_2), such that $s_+ \in O_+$, s_-^1, $s_-^2 \in O_-$ and $(s_+, y) \rightarrow_{\mathbb{N}} (s_-^1, z_1) \vee (s_-^2, z_2)$, $(s_+, y) \nrightarrow_{\mathbb{N}} (s_-^1, z_1)$, $(s_+, y) \nrightarrow_{\mathbb{N}} (s_-^2, z_2)$ that consist a violation of the weak convexity condition. Table 6 provides the counterexamples; the first four, next four and last two lines refer to Lemma 4(a), 4(b) and 4(c) respectively. □

Lemma 5. *Every NDR in Table 4 is maximal safe, that is if any relation is added to* O_+ *or* O_- *it becomes non-safe.*

Proof. We examine all cases of adding a new relation to NDRs in Table 4:
<u>NDR_1:</u> If any of the $<, \leq, >, \geq$ is added to O_+, then NDR_1 becomes non-safe due to Lemma 4(a).
<u>NDR_2:</u> If $>$ or $\geq$ is added to O_-, then non-safety is due to Lemma 4(a). When $=$ is added to O_- then NDR_2 becomes non-safe due to Lemma 4(c).
<u>NDR_3:</u> If $<$ or $\leq$ is added to O_-, then non-safety is due to Lemma 4(a). When $=$ is added to O_- then NDR_3 becomes non-safe due to Lemma 4(c).
<u>NDR_4:</u> If $>$ or $\geq$ is added to O_-, then non-safety is due to Lemma 4(b). For adding $<$ or $\leq$ to O_+, non-safety is due to Lemma 4(c). □

It remains to be proved that every safe NDR is contained in some NDR in Table 4. In the following, we assume that O^i_+ and O^i_- are defined such that $\mathsf{NDR}_i = (\mathbb{N}, O^i_+, O^i_-)$ with $1 \leq i \leq 4$.

Lemma 6. *If $(\mathbb{N}, O_+, O_-)$ is a safe NDR, then $O_+ \subseteq O^i_+$ and $O_- \subseteq O^i_-$ for some i $(1 \leq i \leq 4)$.*

Proof. The proof is by case analysis of possible relations in O_+ and O_-.
<u>Case 1:</u> $O_+ \cap \{<, \leq, >, \geq\} = \emptyset$. In this case, $O_+ \subseteq O^1_+$ and $O_- \subseteq O^1_-$.
<u>Case 2:</u> $O_+ \cap \{<, \leq, >, \geq\} \neq \emptyset$. If $O_- \cap \{<, \leq\} \neq \emptyset$ and $O_- \cap \{>, \geq\} \neq \emptyset$ at the same time, then from Lemma 4(a), the NDR is non-safe. Therefore, we examine two cases: either $O_- \subseteq \{>, \geq, =\}$ or $O_- \subseteq \{<, \leq, =\}$.
<u>Case 2.1:</u> $O_- \subseteq \{>, \geq, =\}$. We further distinguish whether $O_- \subseteq \{>, \geq\}$ or $\{=\} \subseteq O_-$.
<u>Case 2.1.1:</u> $O_- \subseteq \{>, \geq\} = O^3_-$ and $O_+ \subseteq O^3_+$.
<u>Case 2.1.2:</u> $\{=\} \subseteq O_-$. By Lemma 4(c) it should be $O_+ \subseteq \{>, \geq, =\} = O^4_+$ otherwise the NDR is non-safe. If $O_- \cap \{>, \geq\} \neq \emptyset$ then the NDR is non-safe by Lemma 4(b); otherwise $O_- = \{=\} \subseteq O^4_-$.
<u>Case 2.2:</u> $O_- \subseteq \{<, \leq, =\} = O^4_-$. If $O_+ \subseteq \{>, \geq, =\}$, then $O_+ \subseteq O^4_+$. Otherwise, $O_+ \cap \{<, \leq\} \neq \emptyset$ and we distinguish cases whether $O_- \subseteq \{<, \leq\}$ or $\{=\} \in O_-$.
<u>Case 2.2.1:</u> $O_- \subseteq \{<, \leq\} = O^2_-$ and $O_+ \subseteq O^2_+$.
<u>Case 2.2.2:</u> $\{=\} \in O_-$. In this case, S is non-safe by Lemma 4(c). □

5 Maximal Safe NDRs for $\mathbb{Z}$, $\mathbb{R}$ and $\mathbb{Q}$

In this section we present the classification of maximal safe NDRs for the cases of integers, reals and rationals. Due to space restriction, we do not provide the full proofs. The interested reader can find details in the technical report [8]. The proofs are analogous to the case of natural numbers. In the following, we provide a brief explanation for the results.

Table 7 provides the safe NDRs for integers. When we compare the results with Table 4 we notice two new maximal safe NDRs, namely NDR_2 and NDR_6. The reason is that integers do not have a minimal element such as 0 in the case of naturals. In particular positive occurrences of $<$ (or $\leq$) and negative occurrence of $=$ are no longer dangerous (e.g. $(\leq, 1) \not\twoheadrightarrow_{\mathbb{Z}} (=, 1) \vee (=, 0)$ does not hold anymore).

Table 7. Maximal safe NDRs for $\mathbb{Z}$

NDR	O_+	O_-
NDR_1	$\{=\}$	$\{<, \leq, >, \geq, =\}$
NDR_2	$\{<, \leq, >, \geq, =\}$	$\{=\}$
NDR_3	$\{<, \leq, >, \geq, =\}$	$\{<, \leq\}$
NDR_4	$\{<, \leq, >, \geq, =\}$	$\{>, \geq\}$
NDR_5	$\{>, \geq, =\}$	$\{<, \leq, =\}$
NDR_6	$\{<, \leq, =\}$	$\{>, \geq, =\}$

Table 8. Maximal safe NDRs for $\mathbb{R}$ and $\mathbb{Q}$

NDR	O_+	O_-
NDR_1	$\{=\}$	$\{<, \leq, >, \geq, =\}$
NDR_2	$\{<, \leq, >, \geq, =\}$	$\{\leq, =\}$
NDR_3	$\{<, \leq, >, \geq, =\}$	$\{\geq, =\}$
NDR_4	$\{<, \leq, >, \geq, =\}$	$\{<, \leq\}$
NDR_5	$\{<, \leq, >, \geq, =\}$	$\{>, \geq\}$
NDR_6	$\{<, >, \geq, =\}$	$\{<, \leq, =\}$
NDR_7	$\{<, \leq, >, =\}$	$\{>, \geq, =\}$

Table 8 presents the maximal safe NDRs for reals, which are the same for rationals. Reals and rationals are examples of dense domains: between every two different numbers there always exists a third one. This property is responsible for new safe NDRs. Specifically, either $\leq$ or $\geq$ can be added to O_- of NDR_2 from Table 7 because it does not violate the weak convexity property (e.g. $(\leq, 5) \not\rightarrow_{\mathbb{R}} (=, 5) \vee (\leq, 4)$). For the same reason, O_+ of NDR_5 and NDR_6 from Table 7 can be extended with $<$ and $>$ respectively because the weak convexity property which did not apply for $\mathbb{Z}$ now applies for $\mathbb{R}$ (e.g. $(<, 5) \not\rightarrow_{\mathbb{R}} (=, 4) \vee (\leq, 3)$).

6 Related Work

Datatypes have been extensively studied in the context of DLs [3,7,9]. Extensions of expressive DLs with datatypes have been examined in depth [7] with the main focus on decidability. Baader, Brandt and Lutz [3] formulated tractable extensions of $\mathcal{EL}$ with datatypes using a p-admissibility restriction for datatypes. A datatype $\mathcal{D}$ is p-admissible if (i) satisfiability and implication of conjunctions of datatype restrictions can be decided in polynomial time, and (ii) $\mathcal{D}$ is convex: if a conjunction of datatype restrictions implies a disjunction of datatype restrictions then it also implies one of its disjuncts [3]. In our case instead of condition (i) we require that implication and satisfiability of just datatype restrictions (not conjunctions) is decidable in polynomial time since we do not consider functional features. Condition (ii) is relaxed to the requirement of safety for NDRs since we take into account not only the domain of the datatypes and the types of restrictions but also the polarity of their occurrences. The relaxed restrictions allow for more expressive usage of datatypes in tractable languages, as demonstrated by the example given in the introduction. Furthermore, Baader, Brandt and Lutz did not provide a classification of datatypes that are p-admissible; in our case we provide such a classification for natural numbers, integers, rationals and reals. The EL Profile of OWL 2 [2] is inspired by $\mathcal{EL}^{++}$ and restricts all OWL 2 datatypes to satisfy p-admissibility. In particular, only equality can be used in datatype restrictions. Our result can allow for a significant extension of datatypes in the OWL 2 EL Profile, where in addition inequalities can be used negatively.

Our work is not the only one where the convexity property is relaxed without losing tractability. It has been shown [9] that the convexity requirement is not necessary provided that (i) the ontology contains only concept definitions of the form $A \equiv C$, where A is a concept name, and (ii) every concept name occurs at most once in the left-hand side of the definition. In some applications this requirement can be too restrictive since it disallows the usage of general concept inclusion axioms (GCIs), such as the axiom (2) given in the introduction, which do not cause any problem in our case.

7 Conclusions and Future Work

In this work we made a fine-grained analysis of extensions of $\mathcal{EL}$ with numerical datatypes, focusing not only on the types of relations but also on the polarities of their occurrences in axioms. We made a full classification of cases where these restrictions result in a tractable extension for natural numbers, integers, rationals and reals. One practically relevant case for these datatypes is when positive occurrences of datatype expressions can only use equality and negative occurrences can use any of the numerical relations considered. This case was motivated by an example of a pharmacy-related ontology and can be proposed as a candidate for a future extension of the OWL 2 EL Profile. For the cases where the extension is tractable, we provided a polynomial sound and complete consequence-based reasoning procedure, which can be seen as an extension of the completion-based procedure for $\mathcal{EL}$. We think that the procedure can be straightforwardly extended to accommodate other constructors in $\mathcal{EL}^{++}$ such as (complex) role inclusions, nominals, domain and range restrictions and assertions since these constructors do not interact with datatypes [10]. We hope to investigate these extensions in future works.

In future work we also plan to consider other OWL datatypes, such as strings, binary data or date and time, functional features, and to try to extend the consequence-based procedure for Horn $\mathcal{SHIQ}$ [11] with our rules for datatypes. For example, to extend the procedure with functional features, we probably need a notion of "functional safety" for an NDR that corresponds to the strong convexity property (see Definition 7). In order to achieve even higher expressivity for datatypes we shall study how to combine different restrictions on the datatypes occurring in an ontology so that tractability is preserved. For example, using two safe NDRs in a single ontology may result in intractability, as is the case for

NDR_1 and NDR_2 for integers (see Table 7). One possible solution to this problem is to specify explicitly which features can be used with which NDRs in order to separate their usage in ontologies.

References

1. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F.: The Description Logic Handbook: Theory, Implementation, and Applications, 2nd edn. Cambridge University Press, Cambridge (2007)
2. Grau, B.C., Horrocks, I., Motik, B., Parsia, B., Patel-Schneider, P.F., Sattler, U.: OWL 2: The next step for OWL. J. Web Sem. 6(4), 309–322 (2008)
3. Baader, F., Brandt, S., Lutz, C.: Pushing the $\mathcal{EL}$ Envelope. In: IJCAI, pp. 364–369. Professional Book Center (2005)
4. Cote, R., Rothwell, D., Palotay, J., Beckett, R., Brochu, L.: The systematized nomenclature of human and veterinary medicine. Technical report, Northfield, IL: College of American Pathologists (1993)
5. The Gene Ontology Consortium: Gene Ontology: Tool for the unification of biology. Nature Genetics 25, 25–29 (2000)
6. Magka, D.: Consequence-Based Datatype Reasoning in $\mathcal{EL}$: Identifying the Tractable Fragments. Master's thesis, Oxford University Computing Laboratory (2009)
7. Lutz, C.: Description Logics with Concrete Domains-A Survey. In: Advances in Modal Logic, pp. 265–296. King's College Publications (2002)
8. Magka, D., Kazakov, Y., Horrocks, I.: Tractable Extensions of the Description Logic $\mathcal{EL}$ with Numerical Datatypes. Technical report, Oxford University Computing Laboratory (2010), posted on, `http://web.comlab.ox.ac.uk/isg/people/despoina.magka/publications/reports/NDRTechnicalReport.pdf`
9. Haase, C., Lutz, C.: Complexity of Subsumption in the $\mathcal{EL}$ Family of Description Logics: Acyclic and Cyclic tboxes. In: ECAI, vol. 178, pp. 25–29. IOS Press, Amsterdam (2008)
10. Baader, F., Brandt, S., Lutz, C.: Pushing the $\mathcal{EL}$ envelope further. In: Proceedings of the OWLED 2008 DC Workshop on OWL: Experiences and Directions (2008)
11. Kazakov, Y.: Consequence-driven Reasoning for Horn $\mathcal{SHIQ}$ Ontologies. In: IJCAI, pp. 2040–2045 (2009)

Analytic Tableaux for Higher-Order Logic with Choice

Julian Backes and Chad E. Brown

Saarland University, Saarbrücken, Germany

Abstract. While many higher-order interactive theorem provers include a choice operator, higher-order automated theorem provers currently do not. As a step towards supporting automated reasoning in the presence of a choice operator, we present a cut-free ground tableau calculus for Church's simple type theory with choice. The tableau calculus is designed with automated search in mind. In particular, the rules only operate on the top level structure of formulas. Additionally, we restrict the instantiation terms for quantifiers to a universe that depends on the current branch. At base types the universe of instantiations is finite. We prove completeness of the tableau calculus relative to Henkin models.

1 Introduction

Interactive theorem provers based on classical higher-order logic (e.g., Isabelle-HOL [16], HOL [12] and the successors of the HOL system) build in the axiom of choice by including a form of Hilbert's ε binder and appropriate rules. Church's formulation of the simple theory of types [11] included a choice operator (called ι) and an axiom of choice at each type. Henkin defined a general notion of a model of Church's type theory with choice and proved completeness [13]. A higher-order version of the TPTP has been under development the past few years [18]. In 2009 it was decided that Henkin models with choice would be the default semantics of the higher-order TPTP.

Automated theorem provers for classical higher-order logic (e.g., TPS [3] and LEO-II [7]) do not currently build in the axiom of choice. Completeness of such calculi is judged with respect to a variant of Henkin's models without choice [2,6]. What would be involved in adding support for choice? Assume a new logical constant ε_σ of type $(\sigma \to o) \to \sigma$ at each type σ is added to the syntax. We need new rules corresponding to this constant. The fundamental property ε_σ should satisfy is expressed by the formula

$$\forall p_{\sigma\to o} x_\sigma . px \to p(\varepsilon_\sigma p)$$

Our purpose in this paper is to give a complete analytic tableau calculus for higher-order logic with choice that forms a basis for automated reasoning in the logic. Mints [15] has given a sequent calculus for relational higher-order logic with an ε-operator and proves completeness. Mints' calculus does not include arbitrary function types and the corresponding simply typed λ-terms. We adapt

Mints' rules for a simply typed formulation in the style of Church. We obtain tighter restrictions on when Mints' main choice rule (the ε-rule) needs to be applied. Furthermore, we show we can omit Mints' ε-extensionality rule altogether. These results are important for automated reasoning because these two rules would be highly branching in practice. In addition to including cut-free rules for the ε-operator, we give strong restrictions on the instantiation of universal quantifiers over base types analogous to those reported in [9].

In Section 2 we give a quick presentation of the syntax and semantics of simple type theory with choice. In Section 3 we present the tableau calculus. In Section 4 we define the notion of an evident set and prove that every evident set has a Henkin model. We define a notion of abstract consistency and use it to prove completeness of the tableau calculus in Section 5. We discuss related work and conclude in Sections 6 and 7. For reasons of space several proofs are omitted. Detailed proofs are available in [5].

2 Preliminaries

We start by giving the syntax for simple type theory with a choice operator in the style of Church [11]. Types (σ, τ, μ) are given inductively by the base type o (of truth values), ι (of individuals) and $\sigma \to \tau$ (of functions from σ to τ). For brevity, we will omit the arrow and write $\sigma\tau$ for $\sigma \to \tau$. Omitted parenthesis in types associate to the right: $\sigma\tau\mu$ means $\sigma(\tau\mu)$. The results in the paper generalize to the case where there are arbitrarily many base types of individuals. We use β to range over the base types o and ι.

For each type σ we assume a countably infinite set $\mathcal{V}_\sigma$ of variables of type σ. For each type σ we have logical constants $=_\sigma$ of type $\sigma\sigma o$, $\forall_\sigma$ of type $(\sigma o)o$ and ε_σ (the choice operator) of type $(\sigma o)\sigma$. Furthermore, we have logical constants for disjunction $\vee$ of type ooo, negation $\neg$ of type oo, false $\perp$ of type o and for a default individual $*$ of type ι. (The default individual is included only to act as an instantiation when no other instantiation of type ι is allowed by our calculus.) We use x, y to range over variables and c to range over logical constants. A name is either a variable or a logical constant. We use ν to range over names. Variables x and choice operators ε_σ are called *decomposable* names. We use δ to range over decomposable names.

The family of sets Λ_σ of terms of type σ are inductively defined. If ν is a name of type σ, then $\nu \in \Lambda_\sigma$. If $t \in \Lambda_{\sigma\tau}$ and $s \in \Lambda_\sigma$, then we have an application term $ts \in \Lambda_\tau$. If $x \in \mathcal{V}_\sigma$ and $t \in \Lambda_\tau$, then we have an abstraction term $\lambda x.s \in \Lambda_{\sigma\tau}$. A *formula* is a term $s \in \Lambda_o$.

Application associates to the left, so that stu means $(st)u$, with the exception that $\neg tu$ always means $\neg(tu)$. We use infix notation and write $s =_\sigma t$ (or $s = t$) for $=_\sigma st$ and write $s \vee t$ for $\vee st$. We write $s \neq_\sigma t$ (or $s \neq t$) for $\neg(s =_\sigma t)$. We also use binder notation to write $\forall x.s$ for $\forall_\sigma \lambda x.s$ and write $\varepsilon x.s$ for $\varepsilon_\sigma \lambda x.s$.

The set $\mathcal{V}t$ of *free variables of* t is defined as usual. For a set of variables X we write Λ_σ^X for the set of all terms $t \in \Lambda_\sigma$ such that $\mathcal{V}t \subseteq X$.

An *elimination context* ($\mathcal{E}$) is a term with a hole $[]_\sigma$ defined inductively as follows. $[]_\sigma$ is an elimination context of type σ. If $\mathcal{E}$ is an elimination context of type $\tau\mu$ and $s \in \Lambda_\tau$ then $\mathcal{E}s$ is an elimination context of type μ.

Let $\mathcal{E}$ be an elimination context of type σ which has a hole of type τ. We can apply $\mathcal{E}$ to a term $t \in \Lambda_\tau$ to get a term of type σ: $[\,][t] = t$ and $(\mathcal{E}\ s)[t] = \mathcal{E}[t]\ s$.

An *accessibility context* ($\mathcal{C}$) is a term with a hole $[]_\sigma$ of the form $\mathcal{E}$, $\neg\mathcal{E}$, $\mathcal{E} \neq_\iota s$ or $s \neq_\iota \mathcal{E}$ where $\mathcal{E}$ is an elimination context. We can apply an accessibility context $\mathcal{C}$ with a hole of type σ to a term $t \in \Lambda_\sigma$ to get a term of type o in the obvious way. A term s is *accessible* in a set A of formulas iff there is an accessibility context $\mathcal{C}$ such that $\mathcal{C}[s] \in A$.

Let A be a set of formulas. A term s is *discriminating* in A iff there is a term t such that $s \neq_\iota t \in A$ or $t \neq_\iota s \in A$. A *discriminant* Δ of A is a maximal set of discriminating terms such that there is no $s, t \in \Delta$ with $s \neq_\iota t \in A$. (Discriminants first appeared in [10].)

We now turn to a brief description of the semantics. Our notion of an interpretation is essentially that given by Henkin [13]. A *frame* $\mathcal{D}$ is a typed family of nonempty sets such that $\mathcal{D}_o = \{0, 1\}$ and $\mathcal{D}_{\sigma\tau}$ is a set of total functions from $\mathcal{D}_\sigma$ to $\mathcal{D}_\tau$. $\mathcal{D}_o$ is the set of Booleans 0 (*false*) and 1 (*true*). An *assignment* into a frame $\mathcal{D}$ is a function $\mathcal{I}$ that maps every name ν of type σ to an element of $\mathcal{D}_\sigma$. We denote $\mathcal{I}^x_a$ to be the assignment that is like $\mathcal{I}$ but maps the variable x to a.

Table 1. Properties of values of logical constants

prop.	where	holds		for all
$\mathfrak{L}_*(a)$	$a \in \mathcal{D}_\iota$	always		
$\mathfrak{L}_\bot(a)$	$a \in \mathcal{D}_o$	when $a = 0$		
$\mathfrak{L}_\neg(n)$	$n \in \mathcal{D}_{oo}$	when $na = 1$	iff $a = 0$	$a \in \mathcal{D}_o$
$\mathfrak{L}_\vee(d)$	$d \in \mathcal{D}_{ooo}$	when $dab = 1$	iff $a = 1$ or $b = 1$	$a, b \in \mathcal{D}_o$
$\mathfrak{L}_{\forall_\sigma}(p)$	$p \in \mathcal{D}_{(\sigma o)o}$	when $pf = 1$	iff $\forall a \in \mathcal{D}_\sigma\ fa = 1$	$f \in \mathcal{D}_{\sigma o}$
$\mathfrak{L}_{=_\sigma}(q)$	$q \in \mathcal{D}_{\sigma\sigma o}$	when $qab = 1$	iff $a = b$	$a, b \in \mathcal{D}_\sigma$
$\mathfrak{L}_{\varepsilon_\sigma}(\Phi)$	$\Phi \in \mathcal{D}_{(\sigma o)\sigma}$	when $f(\Phi f) = 1$	iff $\exists a \in \mathcal{D}_\sigma\ fa = 1$	$f \in \mathcal{D}_{\sigma o}$

For each logical constant c of type σ we define a corresponding property $\mathfrak{L}_c(a)$ of elements $a \in \mathcal{D}_\sigma$ in Table 1. Essentially $\mathfrak{L}_c(a)$ holds iff a is an appropriate interpretation of c. An assignment $\mathcal{I}$ into $\mathcal{D}$ is *logical* if $\mathfrak{L}_c(\mathcal{I}c)$ holds for each logical constant c. A logical assignment $\mathcal{I}$ must map $\bot$ to 0, $\neg$ to the negation function, and so on. There is no restriction on the value of $\mathcal{I}*$ in $\mathcal{D}_\iota$. The most interesting case to consider is the choice function ε_σ. For an assignment to be logical, $\mathcal{I}\varepsilon_\sigma$ must be a function in $\mathcal{D}_{(\sigma o)\sigma}$ such that $f((\mathcal{I}\varepsilon_\sigma)f) = 1$ for every $f \in \mathcal{D}_{\sigma o}$ except when f is the constant 0 function. There may be many different elements in $\mathcal{D}_{(\sigma o)\sigma}$ satisfying this condition. (Of course, there may also be no element satisfying the condition.)

We now turn to the interpretation of all typed terms. To do this we use induction on terms to lift each assignment $\mathcal{I}$ to a partial function $\hat{\mathcal{I}}$ on terms:

$$\begin{array}{lll} \hat{\mathcal{I}}(\nu) := \mathcal{I}(\nu) & & \\ \hat{\mathcal{I}}(st) := fa & \text{if } \hat{\mathcal{I}}s = f \text{ and } \hat{\mathcal{I}}t = a & \\ \hat{\mathcal{I}}(\lambda x.s) := f & \text{if } \lambda x.s \in \Lambda_{\sigma\tau},\ f \in \mathcal{D}_{\sigma\tau} \text{ and } \forall a \in \mathcal{D}_\sigma\colon \widehat{\mathcal{I}^x_a}s = fa & \end{array}$$

If $\hat{\mathcal{I}}$ is a total function, then we say $\mathcal{I}$ is an *interpretation*.

A *model* $(\mathcal{D},\mathcal{I})$ is a frame $\mathcal{D}$ and a logical interpretation $\mathcal{I}$ into $\mathcal{D}$. We say that a model $(\mathcal{D},\mathcal{I})$ satisfies a formula s iff $\hat{\mathcal{I}}(s) = 1$. A formula is *satisfiable* iff there is a model $(\mathcal{D},\mathcal{I})$ such that $\hat{\mathcal{I}}(s) = 1$. We say $(\mathcal{D},\mathcal{I})$ is *a model of* a set of formulas A if $\hat{\mathcal{I}}(s) = 1$ for every $s \in A$. A set A of formulas is *satisfiable* if there is a model of A.

We assume a type preserving and total *normalization operator* $[\cdot]$ from terms to terms. A term is *normal* iff $[s] = s$. A set of terms is normal if every element of this set is normal. Instead of committing to a specific operator such as β-normalization or $\beta\eta$-normalization, we require the following properties:

N1 $[[s]] = [s]$
N2 $[[s]t] = [st]$
N3 $[\nu s_1 \dots s_n] = \nu[s_1] \dots [s_n]$ if $\nu s_1 \dots s_n \in \Lambda_\beta$ for some base type β and $n \geq 0$
N4 $\hat{\mathcal{I}}[s] = \hat{\mathcal{I}}s$ for every model $(\mathcal{D},\mathcal{I})$.
N5 $\mathcal{V}[s] \subseteq \mathcal{V}s$

Note that by N5 we know $[s] \in \Lambda^X_\sigma$ whenever $s \in \Lambda^X_\sigma$.

A *substitution* is a type preserving partial function from variables to terms. If θ is a substitution, x is a variable, and s is a term that has the same type as x, we write θ^x_s for the substitution that agrees everywhere with θ except possibly on x where it yields s. For each substitution θ we assume there is a type preserving total function $\hat{\theta}$ from terms to terms such that the following conditions hold:

S1 $\hat{\theta}x = \theta x$ for every $x \in \mathrm{Dom}\,\theta$
S2 $\hat{\theta}(st) = (\hat{\theta}s)(\hat{\theta}t)$
S3 $[(\hat{\theta}(\lambda x.s))t] = [\widehat{\theta^x_t}s]$
S4 $[\hat{\theta}s] = [s]$ if $\theta x = x$ for every $x \in \mathrm{Dom}\,\theta \cap \mathcal{V}s$
S5 $[\hat{\theta}[s]] = [\hat{\theta}s]$

The following proposition demonstrates that we can recover a form of β-reduction relative to abstract normalization and substitution. The empty set $\emptyset$ is the substitution that is undefined on every variable.

Proposition 1. $[[\lambda x.s]t] = [\widehat{\emptyset^x_t}s]$

Proof. $[[\lambda x.s]t] \overset{\text{S4}}{=} [[\hat{\emptyset}(\lambda x.s)]t] \overset{\text{N2}}{=} [(\hat{\emptyset}(\lambda x.s))t] \overset{\text{S3}}{=} [\widehat{\emptyset^x_t}s]$

For each set A of formulas and each type σ we define a nonempty universe $\mathcal{U}^A_\sigma \subseteq \Lambda_\sigma$ as follows.

- Let $\mathcal{U}^A_o = \{\bot, \neg\bot\}$.
- Let $\mathcal{U}^A_\iota$ be the set of discriminating terms in A if there is some discriminating term in A.
- Let $\mathcal{U}^A_\iota = \{*\}$ if there are no discriminating terms in A.
- Let $\mathcal{U}^A_{\sigma\tau} = \{[s] | s \in \Lambda_{\sigma\tau}, \mathcal{V}s \subseteq \mathcal{V}A\}$.

When trying to refute a set A of formulas, all our instantiations of type σ will come from the set $\mathcal{U}^A_\sigma$. When the set A is clear in context, we write $\mathcal{U}_\sigma$.

3 Tableau Calculus

A *branch* is a finite set of normal formulas. A *step* is an n+1-tuple $\langle A, A_1, \ldots, A_n\rangle$ of branches where $n \geq 1$, $\bot \notin A$ and $A \subset A_i$ for each $i \in \{1, \ldots, n\}$. The branch A is the *head* of the step $\langle A, A_1, \ldots, A_n\rangle$ and each A_i is an *alternative.* A *rule* is a set of steps, and is usually indicated by a schema. For example, the schema

$$\mathcal{T}_{\text{BE}} \quad \frac{s \neq_o t}{s\,,\,\neg t \;\mid\; \neg s\,,\,t}$$

indicates the set of steps $\langle A, A_1, A_2\rangle$ where $s \neq_o t$ is in A, $\bot \notin A$, $\{s, \neg t\} \not\subseteq A_1$, $\{\neg s, t\} \not\subseteq A_1$, $A_1 = A \cup \{s, \neg t\}$ and $A_2 = A \cup \{\neg s, t\}$. We say a rule *applies to* a branch A if some step in the rule has A as its head. A tableau calculus is also a set of steps. Our tableau calculus $\mathcal{T}$ is given as the union of the rules in Figure 1.

$$\mathcal{T}_{\neg} \quad \frac{s,\neg s}{\bot} \qquad \mathcal{T}_{\neq} \quad \frac{s \neq_\iota s}{\bot} \qquad \mathcal{T}_{\neg\neg} \quad \frac{\neg\neg s}{s} \qquad \mathcal{T}_{\vee} \quad \frac{s \vee t}{s \mid t} \qquad \mathcal{T}_{\neg\vee} \quad \frac{\neg(s \vee t)}{\neg s, \neg t}$$

$$\mathcal{T}_{\forall} \quad \frac{\forall_\sigma s}{[st]} \; t \in \mathcal{U}_\sigma \qquad \mathcal{T}_{\neg\forall} \quad \frac{\neg\forall_\sigma s}{\neg[sx]} \; x \in \mathcal{V}_\sigma \text{ fresh}$$

$$\mathcal{T}_{\text{MAT}} \quad \frac{\delta s_1 \ldots s_n\,,\,\neg\delta t_1 \ldots t_n}{s_1 \neq t_1 \mid \cdots \mid s_n \neq t_n} \; n \geq 1 \qquad \mathcal{T}_{\text{DEC}} \quad \frac{\delta s_1 \ldots s_n \neq_\iota \delta t_1 \ldots t_n}{s_1 \neq t_1 \mid \cdots \mid s_n \neq t_n} \; n \geq 1$$

$$\mathcal{T}_{\text{CON}} \quad \frac{s =_\iota t\,,\,u \neq_\iota v}{s \neq u\,,\,t \neq u \mid s \neq v\,,\,t \neq v} \qquad \mathcal{T}_{\text{BE}} \quad \frac{s \neq_o t}{s\,,\,\neg t \;\mid\; \neg s\,,\,t} \qquad \mathcal{T}_{\text{BQ}} \quad \frac{s =_o t}{s\,,\,t \;\mid\; \neg s\,,\,\neg t}$$

$$\mathcal{T}_{\text{FE}} \quad \frac{s \neq_{\sigma\tau} t}{\neg[\forall x.sx = tx]} \; x \notin \mathcal{V}s \cup \mathcal{V}t \qquad \mathcal{T}_{\text{FQ}} \quad \frac{s =_{\sigma\tau} t}{[\forall x.sx = tx]} \; x \notin \mathcal{V}s \cup \mathcal{V}t$$

$$\mathcal{T}_{\varepsilon} \quad \frac{}{[\forall x.\neg(sx)] \;\mid\; [s(\varepsilon s)]} \; \varepsilon s \text{ accessible, } x \notin \mathcal{V}s$$

Fig. 1. Tableau rules

In the rules $\mathcal{T}_{\text{MAT}}$ (the mating rule) and $\mathcal{T}_{\text{DEC}}$ (the decomposition rule) δ ranges over decomposable names (variables and choice operators). In the rule $\mathcal{T}_{\forall}$ the instantiation term t must belong to the set $\mathcal{U}_\sigma^A$ where A is the head of the step. In the rule $\mathcal{T}_{\neg\forall}$ the variable x must be *fresh* in the sense that it is not in $\mathcal{V}A$ where A is the head of the step. We restrict the $\mathcal{T}_{\neg\forall}$ to apply only in the case where there is no variable $y \in \mathcal{V}_\sigma$ such that $\neg[sy]$ is in the head A. In the context of an automated prover, this restriction implies there is no need to apply the $\mathcal{T}_{\neg\forall}$ rule to a formula $\neg\forall s$ more than once.

We explain the choice rule $\mathcal{T}_\varepsilon$. Whenever we must consider εs, either s corresponds to the empty set and hence $\forall x.\neg(sx)$ holds, or s represents a set containing at least one element and $s(\varepsilon s)$ holds. Note that we obtain a complete calculus even though we only apply the choice rule when εs occurs on the branch in the form $\mathcal{C}[\varepsilon s]$ for some accessibility context $\mathcal{C}$. That is, the choice rule only applies using εs when the branch contains a formula of the form $\varepsilon s t_1 \cdots t_n$, $\neg(\varepsilon s t_1 \cdots t_n)$, $(\varepsilon s t_1 \cdots t_n) \neq_\iota u$ or $u \neq_\iota (\varepsilon s t_1 \cdots t_n)$. This is a tighter restriction than the one given for the choice rule in [15].

The set of *refutable* branches is defined inductively as follows. If $\perp \in A$, then A is refutable. If $\langle A, A_1, \ldots, A_n\rangle$ is a step in $\mathcal{T}$ and every alternative A_i is refutable, then A is refutable.

Proposition 2 (Soundness). *If A is refutable, then A is unsatisfiable.*

Proof. It is enough to check for each step $\langle A, A_1, \cdots, A_n\rangle$ in $\mathcal{T}$ that if A is satisfiable, then A_i is satisfiable for some $i \in \{1, \ldots, n\}$. Each case is easy. For the steps involving the normalization operator, property N4 is used.

Example 1. Let $p \in \mathcal{V}_{\iota o}$. For this example, assume p and $\lambda x.\neg px$ are normal. We refute the set $\{p(\varepsilon x.\neg px), \neg p(\varepsilon p)\}$ using the rules $\mathcal{T}_{\text{MAT}}$, $\mathcal{T}_\varepsilon$, $\mathcal{T}_\forall$ and $\mathcal{T}_\neg$.

$$\begin{array}{c|c}
\multicolumn{2}{c}{p(\varepsilon x.\neg px)} \\
\multicolumn{2}{c}{\neg p(\varepsilon p)} \\
\multicolumn{2}{c}{(\varepsilon x.\neg px) \neq \varepsilon p} \\
\hline
\forall x.\neg px & p(\varepsilon p) \\
\neg p(\varepsilon x.\neg px) & \perp \\
\perp &
\end{array}$$

4 Evident Sets and Model Existence

Let E be a set of normal formulas. We say E is *evident* if it satisfies the conditions in Figure 2. The conditions $\mathcal{E}_{\text{FE}}$, $\mathcal{E}_{\text{FQ}}$ and $\mathcal{E}_\varepsilon$ are formulated in a slightly different way than the corresponding tableau rules $\mathcal{T}_{\text{FE}}$, $\mathcal{T}_{\text{FQ}}$ and $\mathcal{T}_\varepsilon$. The tableau rules are formulated in a way that makes proof search more practical while the evidence conditions are formulated in a way that will ease the model construction. The next proposition demonstrates that these three evidence conditions could also be formulated differently. Later we will use the proposition to help prove certain sets are evident. We omit the proof.

Proposition 3. *Let E be a set of normal formulas satisfying $\mathcal{E}_\forall$ and $\mathcal{E}_{\neg\forall}$.*

1. *For $s, t \in \Lambda_{\sigma\tau}$ and $x \in \mathcal{V}_\sigma \setminus (\mathcal{V}s \cup \mathcal{V}t)$, if $\neg[\forall x.sx =_\tau tx]$ is in E, then $[sy] \neq [ty]$ is in E for some $y \in \mathcal{V}_\sigma$.*
2. *For $s, t \in \Lambda_{\sigma\tau}$ and $x \in \mathcal{V}_\sigma \setminus (\mathcal{V}s \cup \mathcal{V}t)$, if $[\forall x.sx =_\tau tx]$ is in E, then $[su] = [tu]$ is in E for every $u \in \mathcal{U}_\sigma^E$.*
3. *For $s \in \Lambda_{\sigma\tau}$ and $x \in \mathcal{V}_\sigma \setminus \mathcal{V}s$, if $[\forall x.\neg sx]$ is in E, then $\neg[st]$ is in E for every $t \in \mathcal{U}_\sigma^E$.*

Let E be an evident set. In the rest of this section we will construct a model of E. The construction is similar to the ones in [8,9] except for some complications arising from the instantiation restrictions.

$\mathcal{E}_\bot$	$\bot$ is not in E.
$\mathcal{E}_\neg$	If $\neg s$ is in E, then s is not in E.
$\mathcal{E}_{\neq}$	$s \neq_\iota s$ is not in E.
$\mathcal{E}_{\neg\neg}$	If $\neg\neg s$ is in E, then s is in E.
$\mathcal{E}_\vee$	If $s \vee t$ is in E, then either s or t is in E.
$\mathcal{E}_{\neg\vee}$	If $\neg(s \vee t)$ is in E, then $\neg s$ and $\neg t$ are in E.
$\mathcal{E}_\forall$	If $\forall_\sigma s$ is in E, then $[st]$ is in E for every $t \in \mathcal{U}^E_\sigma$.
$\mathcal{E}_{\neg\forall}$	If $\neg\forall_\sigma s$ is in E, then $\neg[sx]$ is in E for some $x \in \mathcal{V}_\sigma$.
$\mathcal{E}_{\text{MAT}}$	If $\delta s_1 \dots s_n$ and $\neg\delta t_1 \dots t_n$ are in E where $n \geq 1$, then $s_i \neq t_i$ is in E for some $i \in \{1, \dots, n\}$.
$\mathcal{E}_{\text{DEC}}$	If $\delta s_1 \dots s_n \neq_\iota \delta t_1 \dots t_n$ is in E where $n \geq 1$, then $s_i \neq t_i$ is in E for some $i \in \{1, \dots, n\}$.
$\mathcal{E}_{\text{CON}}$	If $s =_\iota t$ and $u \neq_\iota v$ are in E, then either $s \neq u$ and $t \neq u$ are in E or $s \neq v$ and $t \neq v$ are in E.
$\mathcal{E}_{\text{BE}}$	If $s \neq_o t$ is in E, then either s and $\neg t$ are in E or $\neg s$ and t are in E.
$\mathcal{E}_{\text{BQ}}$	If $s =_o t$ is in E, then either s and t are in E or $\neg s$ and $\neg t$ are in E.
$\mathcal{E}_{\text{FE}}$	If $s \neq_{\sigma\tau} t$ is in E, then $[sx] \neq [tx]$ is in E for some $x \in \mathcal{V}_\sigma$.
$\mathcal{E}_{\text{FQ}}$	If $s =_{\sigma\tau} t$ is in E, then $[su] = [tu]$ is in E for every $u \in \mathcal{U}^E_\sigma$.
$\mathcal{E}_\varepsilon$	If $\varepsilon_\sigma s$ is accessible in E, then either $[s(\varepsilon s)]$ is in E or $\neg[st]$ is in E for every $t \in \mathcal{U}^E_\sigma$.

Fig. 2. Evidence conditions

Let X be the set $\mathcal{V}E$ of free variables in E. We begin by defining a binary relation $\rhd_\sigma$ by induction on types. For each σ, let $\mathcal{D}_\sigma$ be the range of $\rhd_\sigma$, i.e., set of all a such that there is some $s \in \Lambda^X_\sigma$ such that $s \rhd_\sigma a$.

- $s \rhd_o 0$ if $s \in \Lambda^X_o$ and $[s] \notin E$.
- $s \rhd_o 1$ if $s \in \Lambda^X_o$ and $\neg[s] \notin E$.
- $s \rhd_\iota \Delta$ if $s \in \Lambda^X_\iota$, Δ is a discriminant (of E), and either $[s]$ is not a discriminating term or $[s] \in \Delta$.
- $s \rhd_{\sigma\tau} f$ if $s \in \Lambda^X_{\sigma\tau}$, $f : \mathcal{D}_\sigma \to \mathcal{D}_\tau$ and $st \rhd_\tau fa$ whenever $t \rhd_\sigma a$.

Clearly we have $\rhd_\sigma \subseteq \Lambda^X_\sigma \times \mathcal{D}_\sigma$. Also, by definition of $\mathcal{D}$ we have that for every $a \in \mathcal{D}_\sigma$ there is some $s \in \Lambda^X_\sigma$ such that $s \rhd_\sigma a$. For any set $T \subseteq \Lambda^X_\sigma$ we write $T \rhd a$ if $s \rhd a$ for every $s \in T$.

Lemma 1. *For all types σ, terms $s \in \Lambda^X_\sigma$ and values $a \in \mathcal{D}_\sigma$, $s \rhd a$ iff $[s] \rhd a$.*

Proof. This follows by an easy induction on types σ using N1, N2 and N5. The proof is essentially the same as that of Lemma 3.3 in [8].

The next proposition records a number of useful facts about $\rhd$ and $\mathcal{D}$. In particular, $\mathcal{D}$ is a frame and for every value $a \in \mathcal{D}_\sigma$ there is some $t \in \mathcal{U}^E_\sigma$ such that $t \rhd a$. We omit the proof.

Proposition 4

1. *$\bot \rhd 0$ and $\neg\bot \rhd 1$. In particular, $\mathcal{D}_o = \{0, 1\}$.*
2. *For every discriminant Δ, there is a term $t \in \mathcal{U}^E_\iota$ such that $t \rhd \Delta$. In particular, $\mathcal{D}_\iota$ is the set of all discriminants.*
3. *For all types σ and $a \in \mathcal{D}_\sigma$ there is a term $t \in \mathcal{U}^E_\sigma$ such that $t \rhd a$.*

4. *If $t \rhd_\mu b$ and $x \in \mathcal{V}_\tau \setminus \mathcal{V}t$, then $\lambda x.t \rhd K_b$ where $K_b : \mathcal{D}_\tau \to \mathcal{D}_\mu$ is the constant b function.*
5. *For all types σ, $\mathcal{D}_\sigma$ is nonempty.*
6. *$\mathcal{D}$ is a frame.*

We now turn to a notion of compatibility of terms. For $s, t \in \Lambda_\sigma^X$ we say $s \sharp t$ holds if either $s \neq t$ or $t \neq s$ is in E.

Definition 1. *For each type σ we define when two terms $s, t \in \Lambda_\sigma^X$ are* compatible (*written $s \parallel t$*) *by induction on types.*

$\sigma = o$**:** *$s \parallel t$ if $\{[s], \neg[t]\} \not\subseteq E$ and $\{\neg[s], [t]\} \not\subseteq E$.*
$\sigma = \iota$**:** *$s \parallel t$ if $[s]\sharp[t]$ does not hold.*
$\sigma = \tau\mu$**:** *$s \parallel t$ if for all $u, v \in \Lambda_\tau^X$ $u \parallel v$ implies $su \parallel tv$.*

We say a set $T \subseteq \Lambda_\sigma^X$ is compatible *if $s \parallel t$ for all $s, t \in T$.*

The next lemma provides relationships between compatibility of terms and the presence of disequations in E. Note that part (2) of the lemma implies $\varepsilon_\sigma \parallel \varepsilon_\sigma$ for every type σ and $x \parallel x$ for every variable $x \in X$.

Lemma 2. *For all types σ we have the following:*

1. *For all $s, t \in \Lambda_\sigma^X$, if $s \parallel t$, then $[s]\sharp[t]$ does not hold.*
2. *For all $\delta s_1 \cdots s_n, \delta t_1, \cdots t_n \in \Lambda_\sigma^X$ where $n \geq 0$ and δ is a decomposable name, either $\delta s_1 \cdots s_n \parallel \delta t_1, \cdots t_n$ or there is some $i \in \{1, \ldots, n\}$ such that $[s_i]\sharp[t_i]$.*

Proof. By mutual induction on σ. The only complicated case is proving (1) when σ is $\tau\mu$. Assume $s \parallel t$ holds and $[s]\sharp[t]$ holds. By $\mathcal{E}_{\text{FE}}$ $[[s]x]\sharp[[t]x]$ for some variable x. If $x \in X$, then $x \parallel_\tau x$ by inductive hypothesis (2) and so $sx \parallel_\mu tx$, contradicting inductive hypothesis (1) and N2. Assume $x \notin X$. In particular, $x \notin \mathcal{V}s \cup \mathcal{V}t \cup \mathcal{V}[sx] \cup \mathcal{V}[tx]$. In this case we can prove $[sx]$ is $[s(\varepsilon x.\bot)]$ and $[tx]$ is $[t(\varepsilon x.\bot)]$. Using the inductive hypothesis (2) we can prove $\varepsilon x.\bot \parallel \varepsilon x.\bot$ and derive a contradiction.

The next lemma relates compatibility to $\rhd$ and can be proven by an easy induction on types.

Lemma 3. *For all sets $T \subseteq \Lambda_\sigma^X$, T is compatible iff there exists a value $a \in \mathcal{D}_\sigma$ such that $T \rhd a$.*

We now turn to the interpretation of the choice operators. We use a construction similar to that of Mints [15] adapted to our setting.

Let $f \in \mathcal{D}_{\sigma o}$ be a function and εs be a term in Λ_σ^X. We write $f \propto \varepsilon s$ (read f *chooses* εs) iff $s \rhd f$ and $\varepsilon[s]$ is accessible in E. Let $f^0 := \{\varepsilon s \in \Lambda_\sigma^X | f \propto \varepsilon s\}$.

Lemma 4. *Let E be an evident set and let $f \in \mathcal{D}_{\sigma o}$ be a function. Then, there is some $a \in \mathcal{D}_\sigma$ such that $f^0 \rhd a$.*

Proof. We show that f^0 is compatible. Lemma 3 gives us the claim. Let $\varepsilon s, \varepsilon t \in f^0$. By definition of $\propto$, $s, t \rhd f$ and hence, by Lemma 3, $s \parallel t$. By Lemma 2(2) $\varepsilon \parallel \varepsilon$. Thus $\varepsilon s \parallel \varepsilon t$.

For any type σ, we define $\varPhi_\sigma \in \mathcal{D}_{\sigma o} \to \mathcal{D}_\sigma$ as follows:

$$\varPhi_\sigma f = \begin{cases} \text{some } b & \text{such that } fb = 1 \text{ if } f^0 \text{ is empty and such a } b \text{ exists.} \\ \text{some } a & \text{such that } f^0 \triangleright a. \end{cases}$$

The existence of an a in the second case follows from Lemma 4. Note that the second case includes the case in which f is the constant 0 function. In particular, if f is the constant 0 function and f^0 is empty, then $\varPhi_\sigma f$ can be any $a \in \mathcal{D}_\sigma$.

Lemma 5. *Let E be an evident branch, ε be a choice operator, $\varepsilon t_1 \dots t_n \in \varLambda_\sigma^X$ and $a \in \mathcal{D}_\sigma$. If $\varepsilon t_1 \dots t_n \not\triangleright a$, then $\varepsilon[t_1]\dots[t_n]$ is accessible in E.*

Proof. The proof is by an easy induction on σ.

Lemma 6. *For any type σ we have $\varepsilon_\sigma \triangleright \varPhi_\sigma$.*

Proof. Assume $\varepsilon \not\triangleright \varPhi$. Then, there are s, f such that $s \triangleright f$ but $\varepsilon s \not\triangleright \varPhi f$. By Lemma 5 $\varepsilon[s]$ is accessible in E. Hence $\varepsilon s \in f^0$. There is some a such that $\varPhi f = a$ and $f^0 \triangleright a$. Thus $\varepsilon s \triangleright a$, a contradiction.

Lemma 7. *$\mathfrak{L}_{\varepsilon_\sigma}(\varPhi_\sigma)$ holds. That is, $\varPhi$ as defined above is a choice function.*

Proof. Let $f \in \mathcal{D}_{\sigma o}$ be a function and $b \in \mathcal{D}_\sigma$ be such that $fb = 1$. Suppose $f(\varPhi f) = 0$. Then f^0 must be nonempty (by the definition of $\varPhi f$). Choose some $\varepsilon s \in f^0$. We will show a contradiction. By $\mathcal{E}_\varepsilon$ there are two possibilities:

1. $[s(\varepsilon s)] \in E$: In this case $s(\varepsilon s) \not\triangleright 0$. On the other hand, $s \triangleright f$ and $\varepsilon \triangleright \varPhi$ (by Lemma 6) and so $s(\varepsilon s) \triangleright f(\varPhi f)$. This contradicts our assumption that $f(\varPhi f) = 0$.
2. $\neg[st] \in E$ for every $t \in \mathcal{U}_\sigma^E$. By Proposition 4(3) there is some term $t \in \mathcal{U}_\sigma^E$ such that $t \triangleright b$. Hence $\neg[st] \in E$. By definition of $\triangleright_o$, $st \not\triangleright 1$. On the other hand, we know $st \triangleright fb$ since $s \triangleright f$ and $t \triangleright b$, contradicting the assumption that $fb = 1$.

Lemma 8. *If $s \triangleright_\sigma a$, $t \triangleright_\sigma b$ and $s = t$ is in E, then $a = b$.*

Lemma 9. *For each $c \in \varLambda_\sigma$ there is some $a \in \mathcal{D}_\sigma$ such that $\mathfrak{L}_c(a)$ and $c \triangleright a$.*

Proof. If c is a choice operator ε_σ, then we know $\varepsilon_\sigma \triangleright \varPhi_\sigma$ and $\mathfrak{L}_{\varepsilon_\sigma}(\varPhi_\sigma)$ by Lemmas 6 and 7. Checking for the other logical constants is tedious, but not difficult. Lemma 8 is used in the case where c is $=_\sigma$.

We say an assignment $\mathcal{I}$ into $\mathcal{D}$ is *admissible* if $c \triangleright \mathcal{I}c$ for all logical constants c. The following lemma can be proven by induction on terms.

Lemma 10. *Let s be a term, θ be a substitution and $\mathcal{I}$ be an admissible assignment into $\mathcal{D}$. Suppose for every $x \in \mathcal{V}s$, $x \in \operatorname{Dom}\theta$ and $\theta x \triangleright \mathcal{I}x$. Then $s \in \operatorname{Dom}\hat{\mathcal{I}}$ and $\hat{\theta}s \triangleright \hat{\mathcal{I}}s$.*

Now we can prove the model existence theorem for evident sets.

Theorem 1 (Model Existence). *Every evident set is satisfiable.*

Proof. Let E be an evident set. Take $\triangleright$ and $\mathcal{D}$ as defined in this section. We define an assignment $\mathcal{I}$ as follows. For each logical constant c we can choose $\mathcal{I}c$ such that $c \triangleright \mathcal{I}c$ and $\mathfrak{L}_c(\mathcal{I}c)$ by Lemma 9. This ensures we will have a logical, admissible assignment. For each variable $x \in X$ we know $x \parallel x$ by Lemma 2(2) and we can choose $\mathcal{I}x$ such that $x \triangleright \mathcal{I}x$ by Lemma 3. For each variable $x \in \mathcal{V}_\sigma$ not in X we take $\mathcal{I}x = \Phi_\sigma K_0$ where K_0 is the constant 0 function. Let θ be the substitution mapping each $x \in X$ to x and each variable $x \in \mathcal{V}_\sigma$ not in X to $\varepsilon_\sigma y.\bot \in \Lambda_\sigma^X$. By Lemma 6 and Proposition 4(4) we know that $\varepsilon_\sigma y.\bot \triangleright \Phi K_0$ and hence $\theta x \triangleright \mathcal{I}x$ for every variable x. By Lemma 10 we know every $s \in \operatorname{Dom} \hat{\mathcal{I}}$ and $\hat{\theta} s \triangleright \hat{\mathcal{I}} s$ for every term s. In particular, $\mathcal{I}$ is an interpretation. It remains to prove $\hat{\mathcal{I}} s = 1$ for all $s \in E$. Let $s \in E$ be given. By S4 we know $[\hat{\theta} s] = [s]$. Using this and Lemma 1 we know $s \triangleright \hat{\mathcal{I}} s$. Since $s \not\triangleright 0$ and $s \triangleright \hat{\mathcal{I}} s$, we must have $\hat{\mathcal{I}} s = 1$ as desired.

We can now prove that if the tableau calculus $\mathcal{T}$ cannot make progress on a branch, then this branch is satisfiable and in fact has a model with finitely many individuals.

Corollary 1. *Let A be a branch. Suppose $\bot \notin A$ and A is not the head of any step in the calculus $\mathcal{T}$. Then A is evident and there is a model $(\mathcal{D}, \mathcal{I})$ of A where $\mathcal{D}_\iota$ is finite.*

Proof. Once we know A is evident, we know we have a model $(\mathcal{D}, \mathcal{I})$ of A where $\mathcal{D}_\iota$ is the set of discriminants of A. Since A is finite, there are only finitely many discriminating terms of A and hence only finitely many discriminants.

The evidence condition $\mathcal{E}_\bot$ follows from the assumption that $\bot \notin A$. The conditions $\mathcal{E}_\neg$ and $\mathcal{E}_{\neq}$ follows from $\bot \notin A$ and the assumption that the rules $\mathcal{T}_\neg$ and $\mathcal{T}_{\neq}$ do not apply to A. Except for $\mathcal{E}_{\text{FE}}$, $\mathcal{E}_{\text{FQ}}$ and $\mathcal{E}_\varepsilon$, the remaining evidence conditions follow immediately from the assumption that the corresponding rule does not apply. After we know $\mathcal{E}_\forall$ and $\mathcal{E}_{\neg\forall}$ hold for A, we can conclude that $\mathcal{E}_{\text{FE}}$, $\mathcal{E}_{\text{FQ}}$ and $\mathcal{E}_\varepsilon$ hold for A using Proposition 3 and the assumption that the corresponding rule does not apply.

Example 2. Let $p \in \mathcal{V}_{\iota o}$ and $q \in \mathcal{V}_o$. For this example assume $[s] = s$ for all $\beta\eta$-normal forms s. We prove $\forall_o q.\varepsilon_{\iota o} p \neq \varepsilon_{\iota o} x.q$ is satisfiable. Applying tableau rules we can construct a branch with the following formulas:

$$\begin{array}{cccccc}
\forall_o q.\varepsilon p \neq \varepsilon x.q, & \varepsilon p \neq \varepsilon x.\bot, & \varepsilon p \neq \varepsilon x.\neg\bot, & p \neq \lambda x.\bot, & p \neq \lambda x.\neg\bot, & \neg\forall x.px = \bot, \\
\neg\forall x.px = \neg\bot, & px \neq \bot, & py \neq \neg\bot, & px, & \neg\bot, & \neg py, \\
x \neq y, & p(\varepsilon p), & \varepsilon p \neq y, & \forall x.\neg\bot & &
\end{array}$$

5 Abstract Consistency and Completeness

We now lift the model existence theorem for evident sets to a model existence theorem for abstractly consistent sets. This will allow us to prove completeness

$\mathcal{C}_\bot$	$\bot$ is not in A.
$\mathcal{C}_\neg$	If $\neg s$ is in A, then s is not in A.
$\mathcal{C}_{\neq}$	$s \neq_\iota s$ is not in A.
$\mathcal{C}_{\neg\neg}$	If $\neg\neg s$ is in A, then $A \cup \{s\}$ is in Γ.
$\mathcal{C}_\vee$	If $s \vee t$ is in A, then $A \cup \{s\}$ or $A \cup \{t\}$ is in Γ.
$\mathcal{C}_{\neg\vee}$	If $\neg(s \vee t)$ is in A, then $A \cup \{\neg s, \neg t\}$ is in Γ.
$\mathcal{C}_\forall$	If $\forall_\sigma s$ is in A, then $A \cup \{[st]\}$ is in Γ for every $t \in \mathcal{U}_\sigma^A$.
$\mathcal{C}_{\neg\forall}$	If $\neg\forall_\sigma s$ is in A, then $A \cup \{\neg[sx]\}$ is in Γ for some variable x.
$\mathcal{C}_{\text{MAT}}$	If $xs_1 \ldots s_n$ is in A and $\neg xt_1 \ldots t_n$ is in A, then $n \geq 1$ and $A \cup \{s_i \neq t_i\}$ is in Γ for some $i \in \{1, \ldots, n\}$.
$\mathcal{C}_{\text{DEC}}$	If $xs_1 \ldots s_n \neq_\iota xt_1 \ldots t_n$ is in A, then $n \geq 1$ and $A \cup \{s_i \neq t_i\}$ is in Γ for some $i \in \{1, \ldots, n\}$.
$\mathcal{C}_{\text{CON}}$	If $s =_\iota t$ and $u \neq_\iota v$ are in A, then either $A \cup \{s \neq u, t \neq u\}$ or $A \cup \{s \neq v, t \neq v\}$ is in Γ.
$\mathcal{C}_{\text{BE}}$	If $s \neq_o t$ is in A, then either $A \cup \{s, \neg t\}$ or $A \cup \{\neg s, t\}$ is in Γ.
$\mathcal{C}_{\text{BQ}}$	If $s =_o t$ is in A, then either $A \cup \{s, t\}$ or $A \cup \{\neg s, \neg t\}$ is in Γ.
$\mathcal{C}_{\text{FE}}$	If $s \neq_{\sigma\tau} t$ is in A, then $A \cup \{\neg[\forall x.sx =_\tau tx]\}$ is in Γ for some $x \in \mathcal{V}_\sigma \setminus (\mathcal{V}s \cup \mathcal{V}t)$.
$\mathcal{C}_{\text{FQ}}$	If $s =_{\sigma\tau} t$ is in A, then $A \cup \{[\forall x.sx =_\tau tx]\}$ is in Γ for some $x \in \mathcal{V}_\sigma \setminus (\mathcal{V}s \cup \mathcal{V}t)$.
$\mathcal{C}_\varepsilon$	If $\varepsilon_\sigma s$ is accessible in A, then either $A \cup \{[s(\varepsilon s)]\}$ is in Γ or there is some $x \in \mathcal{V}_\sigma \setminus \mathcal{V}s$ such that $A \cup \{[\forall x.\neg(sx)]\}$ is in Γ.

Fig. 3. Abstract consistency conditions (must hold for every $A \in \Gamma$)

of the tableau calculus $\mathcal{T}$. The use of abstract consistency to prove completeness was first used by Smullyan [17] and later used by several authors in various higher-order settings [1,14,6,8].

A set Γ of branches is an *abstract consistency class* if it satisfies the conditions in Figure 3. In Lemma 12 we will prove that every member of an abstract consistency class can be extended to an evident set. In order to verify the $\mathcal{E}_\forall$ condition we will need the following lemma relating universes for different sets of formulas.

Lemma 11. *Let $\mathcal{A}$ be a nonempty subset of Γ and let E be $\bigcup \mathcal{A}$. Suppose for every branch $B \subseteq E$ there is a branch $A \in \mathcal{A}$ such that $B \subseteq A$. Then for every $t \in \mathcal{U}_\sigma^E$ there is some $A \in \mathcal{A}$ such that $t \in \mathcal{U}_\sigma^A$.*

We can now prove the desired extension lemma.

Lemma 12 (Extension Lemma). *Let Γ be an abstract consistency class. For every $A \in \Gamma$ there is an evident set E such that $A \subseteq E$.*

Proof. Let $u^0, u^1, \ldots$ be an enumeration of all normal formulas. We will construct a sequence $A_0 \subseteq A_1 \subseteq A_2 \subseteq \cdots$ of branches such that every $A_n \in \Gamma$. Let $A_0 := A$. We define A_{n+1} by cases. If there is no $B \in \Gamma$ such that $A_n \cup \{u_n\} \subseteq B$, then let $A_{n+1} := A_n$. Otherwise, choose some $B \in \Gamma$ such that $A_n \cup \{u_n\} \subseteq B$. We consider five subcases.

1. If u_n is of the form $\neg\forall_\sigma s$, then choose A_{n+1} to be $B \cup \{\neg[sx]\} \in \Gamma$ for some $x \in \mathcal{V}_\sigma$. This is possible since Γ satisfies $\mathcal{C}_{\neg\forall}$.
2. If u_n is of the form $s \neq_{\sigma\tau} t$, then choose A_{n+1} to be $B \cup \{\neg[\forall x.sx =_\tau tx]\} \in \Gamma$ for some $x \in \mathcal{V}_\sigma \setminus ([s] \cup [t])$. This is possible by $\mathcal{C}_{\text{FE}}$.

3. If u_n is of the form $s =_{\sigma\tau} t$, then choose A_{n+1} to be $B \cup \{[\forall x.sx =_\tau tx]\} \in \Gamma$ for some $x \in \mathcal{V}_\sigma \setminus ([s] \cup [t])$. This is possible by $\mathcal{C}_{\text{FQ}}$.
4. Suppose u_n is of the form $C[\varepsilon_\sigma s]$ where C is an accessibility context. (Note that if u_n is of this form, then it cannot be of one of the previous forms by the definition of an accessibility context.) By $\mathcal{C}_\varepsilon$ there either $B \cup \{[s(\varepsilon s)]\}$ is in Γ or there is some $x \in \mathcal{V}_\sigma \setminus \mathcal{V}s$ such that $B \cup \{[\forall x.\neg(sx)]\}$ is in Γ. If $B \cup \{[s(\varepsilon s)]\}$ is in Γ, then let A_{n+1} be $B \cup \{[s(\varepsilon s)]\}$. Otherwise, choose A_{n+1} to be $B \cup [\forall x.\neg(sx)] \in \Gamma$ for some $x \in \mathcal{V}_\sigma \setminus \mathcal{V}s$.
5. If no previous case applies, then let A_{n+1} be B.

Let $E := \bigcup_{n \in \mathbb{N}} A_n$. We must prove E satisfies the evidence conditions. We check only $\mathcal{E}_\forall$ and $\mathcal{E}_\varepsilon$ in detail, leaving the others to the reader. Proposition 3 is helpful for verifying $\mathcal{E}_{\text{FE}}$ and $\mathcal{E}_{\text{FQ}}$ just as it is helpful verifying $\mathcal{E}_\varepsilon$ below.

$\mathcal{E}_\forall$ Assume $\forall_\sigma s$ is in E. Let $t \in \mathcal{U}_\sigma^E$ be a normal term. Let n be such that $u_n = [st]$. By Lemma 11 (taking $\mathcal{A}$ to be $\{A_r | r \geq n \text{ and } \forall_\sigma s \in A_r\}$) there is some $r \geq n$ such that $t \in \mathcal{U}_\sigma^{A_r}$ and $\forall_\sigma s$ is in A_r. By $\mathcal{C}_\forall$ $A_r \cup \{[st]\}$ is in Γ. Since $A_n \cup \{u_n\} \subseteq A_r \cup \{[st]\}$, we have $[st] = u_n \in A_{n+1} \subseteq E$.

$\mathcal{E}_\varepsilon$ Assume $\varepsilon_\sigma s$ is accessible in E. Then there is some accessibility context C such that $C[\varepsilon_\sigma s]$ is in E. Let n be such that u_n is $C[\varepsilon_\sigma s]$. Let $r \geq n$ be such that u_n is in A_r. By the definition of A_{n+1} either $[s(\varepsilon s)]$ is in A_{n+1} or $[\forall x.\neg(sx)]$ is in A_{n+1} for some $x \in \mathcal{V}_\sigma \setminus \mathcal{V}s$. In the first case we are done. In the second case let $x \in \mathcal{V}_\sigma \setminus \mathcal{V}s$ be such that $[\forall x.\neg(sx)]$ is in E. Let $t \in \mathcal{U}_\sigma^E$ be given. By Proposition 3(3) we know $\neg[st]$ is in E.

Using the extension lemma we can lift the model existence theorem for evident sets to a model existence theorem for abstract consistency classes.

Theorem 2 (Model Existence). *Let Γ be an abstract consistency class. Every $A \in \Gamma$ is satisfiable.*

Proof. Let $A \in \Gamma$ be given. By Lemma 12 there is an evident set E such that $A \subseteq E$. By Theorem 1 E is satisfiable.

We can now prove completeness of the tableau calculus $\mathcal{T}$. Let $\Gamma_\mathcal{T}$ be the set of all branches A which are not refutable. We will first prove $\Gamma_\mathcal{T}$ is an abstract consistency class and then use Model Existence to prove completeness.

Lemma 13. *$\Gamma_\mathcal{T}$ is an abstract consistency class.*

Proof. It is easy to check each condition in Figure 3 using the corresponding tableau rule in $\mathcal{T}$. For example, we check $\mathcal{C}_\varepsilon$. Suppose $\varepsilon_\sigma s$ is accessible in A, $A \cup \{[s(\varepsilon s)]\}$ is not in $\Gamma_\mathcal{T}$ and $A \cup \{[\forall x.\neg(sx)]\}$ is not in $\Gamma_\mathcal{T}$ for every $x \in \mathcal{V}_\sigma \setminus \mathcal{V}s$. Choose some $x \in \mathcal{V}_\sigma \setminus \mathcal{V}s$. We know $A \cup \{[s(\varepsilon s)]\}$ and $A \cup \{[\forall x.\neg(sx)]\}$ are refutable. Hence A is refutable using $\mathcal{C}_\varepsilon$, contradicting $A \in \Gamma_\mathcal{T}$.

Completeness now follows directly from Lemma 13 and Theorem 2.

Theorem 3 (Completeness). *Let A be a branch. If A is not refutable, then A is satisfiable.*

6 Related Work

This work is an extension of two lines of research. First, we have extended the tableau calculus of Brown and Smolka [8] to support a choice operator at every type. We have done this by modifying sequent rules given by Mints [15] to be tableau rules and adapting the relevant parts of his cut-elimination proof. Second, we have obtained tighter restrictions on the instantiations of quantifiers than were available before.

In [9] Brown and Smolka give a complete tableau calculus for a first-order subsystem (EFO) of higher-order logic. Quantifiers are only allowed at type ι there and the instantiations are restricted to discriminating terms. We have maintained this restriction on instantiations for quantifiers at type ι. In addition we have proven that it is enough to instantiate quantifiers at type o with the two terms $\bot$ and $\neg\bot$. As for quantifiers at function types, we have proven that these instantiations need not consider variables that do not already occur free on the branch.

The choice rule given in this paper is similar to a ε-rule for a sequent calculus given by Mints [15]. We briefly sketch a comparison between our rules and the rules of Mints.

Translating into our language, Mints' rule could be represented as

$$(\textsc{Mints' } \varepsilon)\ \frac{}{[\neg(st)] \mid [s(\varepsilon s)]}\ \varepsilon s \text{ occurs on the branch}$$

By εs occurs on the branch we simply mean that εs appears as any subterm where none of the free variables of s are captured by a λ-binder. Note that this rule could apply more often than our $\mathcal{T}_\varepsilon$ rule. Our $\mathcal{T}_\varepsilon$ rule cannot be applied until εs appears on the branch in one of the forms $\varepsilon s t_1 \cdots t_n$, $\neg(\varepsilon s t_1 \cdots t_n)$, $(\varepsilon s t_1 \cdots t_n) \neq_\iota u$ or $u \neq_\iota (\varepsilon s t_1 \cdots t_n)$. Furthermore, in Mints' system the ε-rule would need to be applied for each new instantiation term t. In practice this could lead to the need to refute branches with $[s(\varepsilon s)]$ multiple times. We have avoided this by using the quantified formula $[\forall x.\neg(sx)]$ on the left branch.

Mints also includes an ε-extensionality rule in [15]. In our context, his rule could be realized as

$$(\textsc{Mints' ext } \varepsilon)\ \frac{}{s \neq t \mid (\varepsilon s) = (\varepsilon t)}\ \varepsilon_\sigma s \text{ and } \varepsilon_\sigma t \text{ occur on the branch}$$

In words, whenever $\varepsilon_\sigma s$ and $\varepsilon_\sigma t$ both occur on the branch, we must consider the case where s and t are different, and the case where εs and εt are the same. This rule could be highly branching in practice. When n different terms of the form εs occur on the branch, then the rule must be applied $\frac{n^2-n}{2}$ times. Furthermore, it has the disadvantage that it adds a positive equation to the branch. If σ is a function type, this will lead to the need to perform instantiations. We were able to omit such a rule entirely from our system and still prove completeness. It seems that Mints needed such a rule because the extensionality in [15] is not liberal enough. Translated into our context, the extensionality rule in [15] includes the rule

$$\text{(Special Case of Mints' extensionality)} \quad \frac{\varepsilon s s_1 \dots s_n\,,\ \neg \varepsilon s t_1 \dots t_n}{s_1 \neq t_1 \mid \cdots \mid s_n \neq t_n} \quad n \geq 1$$

This corresponds to our mating rule, except that we have liberalized the rule to include the case when the corresponding first arguments of ε are different.

$$\text{(Special Case of } \mathcal{T}_{\text{MAT}}) \quad \frac{\varepsilon s_1 \dots s_n\,,\ \neg \varepsilon t_1 \dots t_n}{s_1 \neq t_1 \mid \cdots \mid s_n \neq t_n} \quad n \geq 1$$

7 Conclusion

We have presented a cut-free tableau calculus for Church's simple type theory with a choice operator. The calculus is designed with automated proof search in mind. In particular, only accessible terms on the branch need to be considered in order to apply a rule. Furthermore, instantiation terms are restricted according to the type and the formulas on the branch. At type o only instantiations corresponding to true and false are considered. At the base type ι only discriminating terms on the branch need to be considered (except when there are no discriminating terms in which case a default element can be used). Note that this means only finitely many instantiations at type ι need to be considered at each stage of the search. At function types, the set of instantiations is infinite, but we have at least proven that we do not need to consider instantiations with free variables that do not occur on the current branch.

The first author has extended this work by also considering description operators and if-then-else operators in addition to choice operators. This work has been reported in his Master's thesis [4]. The same style of rules (restricted to accessible terms) and model construction (using discriminants and possible values) can be used to incorporate description and if-then-else. Interpreting if-then-else is straightforward. Interpreting description is analogous to the interpretation of choice given here.

The second author has implemented a new higher-order automated theorem prover, Satallax, based on the ground calculus in this paper. Early results show the implementation to be competitive with the automated theorem provers TPS [3] and LEO-II [7] as well as the automated features of Isabelle [16].

References

1. Andrews, P.B.: Resolution in type theory. J. Symb. Log. 36, 414–432 (1971)
2. Andrews, P.B.: General models and extensionality. J. Symb. Log. 37, 395–397 (1972)
3. Andrews, P.B., Brown, C.E.: TPS: A hybrid automatic-interactive system for developing proofs. Journal of Applied Logic 4(4), 367–395 (2006)
4. Backes, J.: Tableaux for higher-order logic with if-then-else, description and choice. Master's thesis, Universität des Saarlandes (2010)
5. Backes, J., Brown, C.E.: Analytic tableaux for higher-order logic with choice. Technical report, Programming Systems Lab, Saarland University, Saarbrücken, Germany (January 2010)

6. Benzmüller, C., Brown, C.E., Kohlhase, M.: Higher-order semantics and extensionality. J. Symb. Log. 69, 1027–1088 (2004)
7. Benzmüller, C., Theiss, F., Paulson, L., Fietzke, A.: LEO-II — A cooperative automatic theorem prover for higher-order logic. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 162–170. Springer, Heidelberg (2008)
8. Brown, C.E., Smolka, G.: Analytic tableaux for simple type theory and its first-order fragment. Technical report, Programming Systems Lab, Saarland University, Saarbrücken, Germany (December 2009) (accepted for publication by Logical Methods in Computer Science)
9. Brown, C.E., Smolka, G.: Extended first-order logic. In: Urban, C. (ed.) TPHOLs 2009. LNCS, vol. 5674, pp. 164–179. Springer, Heidelberg (2009)
10. Brown, C.E., Smolka, G.: Terminating tableaux for the basic fragment of simple type theory. In: Giese, M., Waaler, A. (eds.) TABLEAUX 2009. LNCS (LNAI), vol. 5607, pp. 138–151. Springer, Heidelberg (2009)
11. Church, A.: A formulation of the simple theory of types. J. Symb. Log. 5, 56–68 (1940)
12. Gordon, M.J., Melham, T.F.: Introduction to HOL: A Theorem-Proving Environment for Higher-Order Logic. Cambridge University Press, Cambridge (1993)
13. Henkin, L.: Completeness in the theory of types. J. Symb. Log. 15, 81–91 (1950)
14. Huet, G.P.: Constrained Resolution: A Complete Method for Higher Order Logic. PhD thesis, Case Western Reserve University (1972)
15. Mints, G.: Cut-elimination for simple type theory with an axiom of choice. J. Symb. Log. 64(2), 479–485 (1999)
16. Nipkow, T., Paulson, L.C., Wenzel, M.T. (eds.): Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)
17. Smullyan, R.M.: First-Order Logic. Springer, Heidelberg (1968)
18. Sutcliffe, G., Benzmüller, C., Brown, C.E., Theiss, F.: Progress in the development of automated theorem proving for higher-order logic. In: Schmidt, R.A. (ed.) CADE 2009. LNCS, vol. 5663, pp. 116–130. Springer, Heidelberg (2009)

Monotonicity Inference for Higher-Order Formulas

Jasmin Christian Blanchette[⋆] and Alexander Krauss

Institut für Informatik, Technische Universität München, Germany
{blanchette,krauss}@in.tum.de

Abstract. Formulas are often monotonic in the sense that if the formula is satisfiable for given domains of discourse, it is also satisfiable for all larger domains. Monotonicity is undecidable in general, but we devised two calculi that infer it in many cases for higher-order logic. The stronger calculus has been implemented in Isabelle's model finder Nitpick, where it is used to prune the search space, leading to dramatic speed improvements for formulas involving many atomic types.

1 Introduction

Formulas occurring in logical specifications often exhibit monotonicity in the sense that if the formula is satisfiable when the types are interpreted with sets of given (positive) cardinalities, it is still satisfiable when these sets become larger. Consider the following formulas, in which superscripts indicate types:

1. $\exists x^{\alpha}\, y.\ x \neq y$
2. $f\, x^{\alpha} = x \wedge f\, y \neq y$
3. $(\forall x^{\alpha}.\ f\, x = x) \wedge f\, y \neq y$
4. $\{y^{\alpha}\} = \{z\}$
5. $\exists x^{\alpha}\, y.\ x \neq y \wedge \forall z.\ z = x \vee z = y$
6. $\forall x^{\alpha}\, y.\ x = y$

Formulas 1 and 2 are satisfiable iff $|\alpha| > 1$, formula 3 is unsatisfiable, formula 4 is satisfiable for any cardinality of α, formula 5 is satisfiable iff $|\alpha| = 2$, and formula 6 is satisfiable iff $|\alpha| = 1$. Formulas 1 to 4 are monotonic, whereas 5 and 6 are not.

Monotonicity can be exploited in model finders to prune the search space. Model finders are automatic tools that generate finite set-theoretic models of formulas. They are useful for exploring a specification (e.g., to check if a set of axioms is satisfiable) and for producing counterexamples. Notable model finders include Paradox [5], MACE [11], and SEM [19] for first-order logic (FOL), Alloy [9] and Kodkod [17] for first-order relational logic, and Nitpick [3] and Refute [18] for higher-order logic (HOL).

Model finders for many-sorted or typed logics typically work by systematically enumerating the domain cardinalities for the atomic types (type variables and other uninterpreted types) occurring in the formula. To exhaust all models up to a given cardinality bound k for a formula involving n atomic types, a model finder iterates through k^n combinations of cardinalities and must consider all models for each of these combinations. In general, this exponential behavior is necessary for completeness, since the formula may require a model with specific cardinalities. However, if the formula is monotonic, it is sufficient to consider only the models in which all types have cardinality k.

[⋆] Research supported by the DFG grant Ni 491/11-1.

Monotonicity occurs surprisingly often in practice. As a real-world example, consider the specification of a hotel key card system with recordable locks [8, pp. 299–306; 13]. Such a specification involves rooms, guests, and keys, modeled as distinct atomic types. A desirable property of the system is that only the occupant of a room may unlock it. Clearly, a counterexample requiring one room, two guests, and four keys will still be a counterexample if more rooms, guests, or keys are available.

In this paper, we present two calculi for detecting monotonicity of HOL formulas. The first calculus (Sect. 5) is based on the idea of tracking the use of equality and quantifiers. Although useful on its own, it mainly serves as a stepping stone for a second, refined calculus (Sect. 6), which uses a type system to detect the ubiquitous "sets as predicates" idiom and treats it specially. The soundness proof of the refined calculus explicitly relates models of different sizes. Both calculi are readily adapted to handle inductive datatypes (Sect. 7), which are pervasive in HOL formalizations. Our evaluation is done in the context of Nitpick (Sect. 8), a counterexample generator for Isabelle/HOL [14]. Although the focus is on HOL, the approach could be adapted to any logic that provides unbounded quantification, such as many-sorted FOL with equality.

2 Related Work

In plain first-order logic without equality, every formula is monotonic, since it is impossible to express an upper bound on the cardinality of the models and hence any model can be extended to a model of arbitrarily larger cardinality. This monotonicity property is essentially a weak form of the upward Löwenheim–Skolem theorem.

When equality is added, nonmonotonic formulas follow suit. For example, the formula $\forall x\, y.\; x = y$ is satisfied only by singleton models. Nonetheless, $\forall x\, y.\; x = y \longrightarrow P(x,y)$ is monotonic, because equality occurs only negatively. Distinguishing between positive and negative occurrences of equality is a natural syntactic criterion for detecting monotonicity, and our approach is based on this idea.

Moving to higher-order logic introduces new complications. Since HOL is typed, we are interested in monotonicity with respect to a given type variable or some other uninterpreted type α. Moreover, our calculi must cope with occurrences of α in nested function types such as $(\alpha \to \beta) \to \beta$ and in datatypes such as α *list*. We are not aware of any previous work on detecting monotonicity for HOL.

In the first-order world, Alloy constitutes an interesting case in point. Although Alloy's logic is unsorted, models must give a semantics to "primitive types," which are sets of uninterpreted atoms. Early versions of the Alloy language ensured monotonicity with respect to the primitive types by providing only bounded quantification and disallowing explicit references to the sets that denote the types [9]. Monotonicity has been lost in more recent versions of the language, which allow such references [8, p. 165]. Nonetheless, many Alloy formulas are monotonic, notably the existential–bounded-universal class of formulas studied by Kuncak and Jackson [10].

For some logics, small model theorems give an upper bound on the cardinality of a sort [4], primitive type [12], or variable's domain [15]. If no model exists below that bound, no larger models exist. Paradox and Alloy exploit such theorems to speed up the search. Our approach is complementary and could be called a "large model" theorem.

3 Higher-Order Logic

Our presentation of HOL is very similar to that of Andrews [1], but instead of a single type ι of individuals, we use type variables α, β, γ to denote uninterpreted types.

Definition 3.1 (Syntax). *The* types *and* terms *of HOL are that of the simply-typed λ-calculus, augmented with constants and a special type o of Booleans:*

Types:		*Terms:*	
$\sigma ::= o$	(*Boolean type*)	$t ::= x^{\sigma}$	(*variable*)
$\quad\mid \alpha$	(*type variable*)	$\quad\mid c^{\sigma}$	(*constant*)
$\quad\mid \sigma \to \sigma$	(*function type*)	$\quad\mid t\ t$	(*application*)
		$\quad\mid \lambda x^{\sigma}.\ t$	(*abstraction*)

The function arrow associates to the right, reflecting the left-associativity of application. We assume throughout that terms are well-typed using the standard typing rules and often omit the type superscripts. A formula is a term of type o.

Unlike in Gordon's version of HOL [6], on which several popular proof assistants are based [7, 14, 16], we treat polymorphism in the metalanguage: Polymorphic constants such as equality are expressed as collections of constants, one for each type.

Types and terms are interpreted in the standard set-theoretic way, relative to a type environment that fixes the interpretation of type variables.

Definition 3.2 (Scope). *A* scope S *is a function from type variables to nonempty sets (domains). We write $S \leq_{\alpha} S'$ to mean that $S(\alpha) \subseteq S'(\alpha)$ and $S(\beta) = S'(\beta)$ for all $\beta \neq \alpha$.*

The set $S(\alpha)$ can be finite or infinite, although for model finding we usually have finite domains in mind. In contexts where S is clear, the cardinality of $S(\alpha)$ is written $|\alpha|$ and the elements of $S(\alpha)$ are denoted by 0, 1, 2, etc. Often, scopes are also called "type environments"; our terminology here is consistent with Jackson [9].

Definition 3.3 (Interpretation of Types). *The* interpretation $[\![\sigma]\!]_S$ *of type σ in scope S is defined recursively by the equations*

$$[\![o]\!]_S = \{\bot, \top\} \qquad [\![\alpha]\!]_S = S(\alpha) \qquad [\![\sigma \to \tau]\!]_S = [\![\sigma]\!]_S \to [\![\tau]\!]_S$$

where $A \to B$ denotes the set of (total) functions from A to B.

Definition 3.4 (Models). *A* constant model *is a scope-indexed family of functions M_S that map each constant c^{σ} to a value $M_S(c) \in [\![\sigma]\!]_S$. A* variable assignment *A for scope S is a function that maps each variable x^{σ} to a value $A(x) \in [\![\sigma]\!]_S$. A* model *for S is a triple $\mathcal{M} = (S, A, M)$, where A is a variable assignment for S and M is a constant model.*

Definition 3.5 (Interpretation of Terms). *Let $\mathcal{M} = (S, A, M)$ be a model. The* interpretation $[\![t]\!]_{\mathcal{M}}$ *of a term t is defined recursively by the equations*

$$[\![x]\!]_{(S,A,M)} = A(x) \qquad [\![t\ u]\!]_{(S,A,M)} = [\![t]\!]_{(S,A,M)}\big([\![u]\!]_{(S,A,M)}\big)$$
$$[\![c]\!]_{(S,A,M)} = M_S(c) \qquad [\![\lambda x^{\sigma}.\ t]\!]_{(S,A,M)} = a \in [\![\sigma]\!]_S \mapsto [\![t]\!]_{(S,A[x \mapsto a],M)}.$$

If t is a formula and $[\![t]\!]_{\mathcal{M}} = \top$, we say that $\mathcal{M}$ is a model *of t, written $\mathcal{M} \vDash t$. A formula is* satisfiable *for scope S if it has a model for S.*

We use constants to express the logical primitives, whose interpretation is fixed a priori for each scope. Our definition of HOL is fairly minimalistic, with equality ($=^{\sigma\to\sigma\to o}$ for any σ) and implication ($\longrightarrow^{o\to o\to o}$) as primitive constants. In the sequel, we always use the standard constant model $\widehat{M}$, which interprets implication and equality in the standard way, and we omit the last component of $(S, A, \widehat{M})$.

The remaining connectives and quantifiers are defined as abbreviations in terms of implication and equality. Abbreviations also cater for set-theoretic notations.

Notation 3.1 (Logical Abbreviations)

$$\begin{array}{rlrl}
\mathit{True} \equiv & (\lambda x^o.\, x) = (\lambda x.\, x) & p \wedge q \equiv & \neg\,(p \longrightarrow \neg\,q) \\
\mathit{False} \equiv & (\lambda x^o.\, x) = (\lambda x.\, \mathit{True}) & p \vee q \equiv & \neg\,p \longrightarrow q \\
\neg\,p \equiv & p \longrightarrow \mathit{False} & \forall x^\sigma.\, p \equiv & (\lambda x.\, p) = (\lambda x.\, \mathit{True}) \\
p \neq q \equiv & \neg\,p = q & \exists x^\sigma.\, p \equiv & \neg\forall x.\, \neg\,p.
\end{array}$$

Notation 3.2 (Set Abbreviations)

$$\begin{array}{lrr}
\emptyset \equiv \lambda x.\, \mathit{False} & s \cap t \equiv \lambda x.\, s\ x \wedge t\ x & x \in s \equiv s\ x \\
\mathcal{U} \equiv \lambda x.\, \mathit{True} & s \cup t \equiv \lambda x.\, s\ x \vee t\ x & \mathit{insert}\ x\ s \equiv (\lambda y.\, y = x) \cup s. \\
 & s - t \equiv \lambda x.\, s\ x \wedge \neg\,t\ x &
\end{array}$$

The constants $\emptyset$ and *insert* can be seen as constructors for finite sets. Following tradition, we write $\{x_1, \ldots, x_n\}$ rather than $\mathit{insert}\ x_1\ (\ldots(\mathit{insert}\ x_n\ \emptyset)\ldots)$.

4 Monotonicity

The introduction gave an informal definition of monotonicity. A more rigorous definition follows.

Definition 4.1 (Monotonicity). *A formula t is* monotonic *w.r.t. a type variable α if for all scopes S, S' such that $S \leq_\alpha S'$, if t is satisfiable for S, it is also satisfiable for S'. It is* antimonotonic *w.r.t. α if its negation is monotonic w.r.t. α.*

Example 4.1. If you have five Swedish friends and all five are blond, the existential statement "at least one of your Swedish friends is dark-haired" is monotonic—it will either stay false or become true as you expand your circle of Nordic friends. Inversely, the universal statement "all your Swedish friends are blond" is antimonotonic—it will either stay true or become false as you make new friends. ■

Theorem 4.1 (Undecidability). *Monotonicity w.r.t. α is undecidable.*

Proof (reduction). For any closed HOL formula t, let $t^\star \equiv t \vee \forall x^\alpha\, y.\, x = y$, where α does not occur in t. Clearly, $t^\star$ must be monotonic if t is valid, since the second disjunct becomes irrelevant in this case. If t is not valid, then $t^\star$ cannot be monotonic, since it is true for $|\alpha| = 1$ due to the second disjunct but false for some larger scopes. Thus, validity in HOL (which is undecidable) can be reduced to monotonicity. □

The best we can do is approximate monotonicity.

Convention. In the rest of this paper, we denote by $\tilde{\alpha}$ the type variable w.r.t. which we consider monotonicity.

5 A Simple Calculus

This section presents the simple calculus $\mathfrak{M}_{\mathsf{F}}$ for inferring monotonicity. This simple calculus serves as a stepping stone toward the more general calculus $\mathfrak{M}_{\mathsf{FS}}$ of Sect. 6. (The 'F' in $\mathfrak{M}_{\mathsf{F}}$ and $\mathfrak{M}_{\mathsf{FS}}$ stands for "function," whereas 'S' stands for "set.") Since the results in this section are subsumed by those of the next section, we omit the proofs.

5.1 Extension Relation and Constancy

We first introduce a concept that is similar to monotonicity but that applies to terms of any type—the notion of *constancy*. Informally, a term is constant if it denotes essentially the same value before and after we enlarge the scope. What it means to denote "essentially the same value" can be formalized using an extension relation $\sqsubseteq$, which relates elements of the smaller scope to elements of the larger scope.

For types such as o and $\tilde{\alpha}$, this is easy: Any element of the smaller scope is also present in the larger scope and can serve as an extension. In the case of functions, we expect that the extended function coincides with the original one where applicable; elements not present in the smaller scope may be mapped to any value. For example, when going from $|\tilde{\alpha}| = 1$ to $|\tilde{\alpha}| = 2$, the function $f^{\tilde{\alpha}\to o} = [0 \mapsto \top]$ can be extended to either $g = [0 \mapsto \top, 1 \mapsto \bot]$ or $g' = [0 \mapsto \top, 1 \mapsto \top]$. For now, we take the liberal view that both g and g' are "essentially the same value" as f, which we write $f \sqsubseteq^{\tilde{\alpha}\to o} g$ and $f \sqsubseteq^{\tilde{\alpha}\to o} g'$. We will reconsider this decision in Sect. 6.

Definition 5.1 (Extension Relation). *Let σ be a type, and let S, S' be scopes such that $S \leq_{\tilde{\alpha}} S'$. The* extension relation *$\sqsubseteq^\sigma \subseteq [\![\sigma]\!]_S \times [\![\sigma]\!]_{S'}$ for S and S' is defined by the following equivalences:*

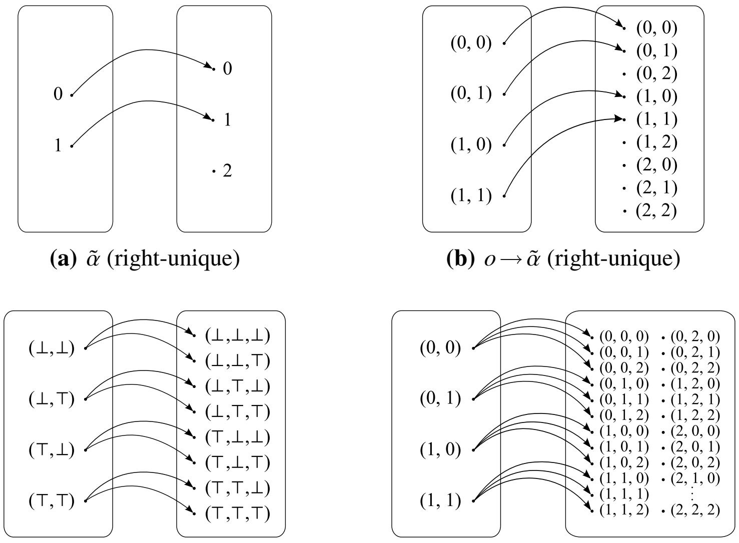
$$\begin{array}{rll} a \sqsubseteq^\sigma b & \textit{iff} \quad a = b & \textit{if } \sigma \textit{ is } o \textit{ or a type variable} \\ f \sqsubseteq^{\sigma\to\tau} g & \textit{iff} \quad \forall a\, b.\; a \sqsubseteq^\sigma b \longrightarrow f(a) \sqsubseteq^\tau g(b). & \end{array}$$

Definition 5.2 (Model Extension). *Let $\mathcal{M} = (S, A)$ and $\mathcal{M}' = (S', A')$ be models. The model $\mathcal{M}'$* extends *$\mathcal{M}$, written $\mathcal{M} \sqsubseteq \mathcal{M}'$, if $S \leq_{\tilde{\alpha}} S'$ and $A(x) \sqsubseteq^\sigma A'(x)$ for all x^σ.*

The symbol $\sqsubseteq^\sigma$ is read "is extended by." Fig. 1 illustrates $\sqsubseteq^\sigma$ for various types. We represent a function from σ to τ by a $|\sigma|$-tuple such that the nth element for σ (according to the lexicographic order, with $\bot < \top$ and $n < n + 1$) is mapped to the nth tuple component. Observe that $\sqsubseteq^\sigma$ is always left-total ($\forall a.\, \exists b.\; a \sqsubseteq^\sigma b$) and left-unique (i.e., injective: $\forall a\, a'\, b.\; a \sqsubseteq^\sigma b \wedge a' \sqsubseteq^\sigma b \longrightarrow a = a'$). It is also right-total (i.e., surjective: $\forall b.\, \exists a.\;\; a \sqsubseteq^\sigma b$) if $\tilde{\alpha}$ does not occur positively in σ (e.g., $\sigma = \tilde{\alpha} \to o$), and right-unique (i.e., functional: $\forall a\, b\, b'.\; a \sqsubseteq^\sigma b \wedge a \sqsubseteq^\sigma b' \longrightarrow b = b'$) if $\tilde{\alpha}$ does not occur negatively (e.g., $\sigma = o \to \tilde{\alpha}$). These properties are crucial to the correctness of our calculus.

Definition 5.3 (Constancy). *A term t^σ is* constant *if $[\![t]\!]_{\mathcal{M}} \sqsubseteq^\sigma [\![t]\!]_{\mathcal{M}'}$ for all models $\mathcal{M}$, $\mathcal{M}'$ such that $\mathcal{M} \sqsubseteq \mathcal{M}'$.*

Example 5.1. $f^{\tilde{\alpha}\to\tilde{\alpha}}\, x$ is constant. Proof: Let $A(x) = a_1$ and $A(f)(a_1) = a_2$. For any $\mathcal{M}' = (S', A')$ that extends $\mathcal{M} = (S, A)$, we have $A(x) \sqsubseteq^{\tilde{\alpha}} A'(x)$ and $A(f) \sqsubseteq^{\tilde{\alpha}\to\tilde{\alpha}} A'(f)$. By definition of $\sqsubseteq^\sigma$, $A'(x) = a_1$ and $A'(f)(a_1) = a_2$. Thus, $[\![f\, x]\!]_{\mathcal{M}} = [\![f\, x]\!]_{\mathcal{M}'} = a_2$. ∎



(a) $\tilde{\alpha}$ (right-unique) **(b)** $o \to \tilde{\alpha}$ (right-unique)

(c) $\tilde{\alpha} \to o$ (right-total) **(d)** $\tilde{\alpha} \to \tilde{\alpha}$ (neither)

Fig. 1. $\sqsubseteq^{\sigma}$ for various types σ, with $|S(\tilde{\alpha})| = 2$ and $|S'(\tilde{\alpha})| = 3$

Example 5.2. $x^{o\to\tilde{\alpha}} = y$ is constant. Proof: For any $\mathcal{M}' = (S', A')$ that extends $\mathcal{M} = (S, A)$, we have $A(x) \sqsubseteq^{o\to\tilde{\alpha}} A'(x)$ and $A(y) \sqsubseteq^{o\to\tilde{\alpha}} A'(y)$. By definition of $\sqsubseteq^{\sigma}$, $A'(x) = A(x)$ and $A'(y) = A(y)$. Hence, $[\![x = y]\!]_{\mathcal{M}} = [\![x = y]\!]_{\mathcal{M}'}$. ∎

Example 5.3. $f^{\tilde{\alpha}\to o} = g$ is not constant. Counterexample: $|S(\tilde{\alpha})| = 1$, $A(f) = A(g) = (\top)$, $|S'(\tilde{\alpha})| = 2$, $A'(f) = (\top, \bot)$, $A'(g) = (\top, \top)$. Then $[\![f = g]\!]_{(S,A)} = \top$ but $[\![f = g]\!]_{(S',A')} = \bot$. ∎

More generally, we note that variables are always constant, and constancy is preserved by application and λ-abstraction. On the other hand, the equality symbol $=^{\sigma\to\sigma\to o}$ is constant only if $\tilde{\alpha}$ does not occur negatively in σ. Moreover, since $\sqsubseteq^{o}$ is the identity relation, constant formulas are both monotonic and antimonotonic.

5.2 Syntactic Criteria

We syntactically approximate constancy, monotonicity, and antimonotonicity with the predicates $\mathbf{K}(t)$, $\mathbf{M}^{+}(t)$, and $\mathbf{M}^{-}(t)$, respectively. The goal is to derive $\mathbf{M}^{+}(t)$ for the formula t we wish to prove monotonic. The predicates depend on $\mathbf{TV}^{+}(\sigma)$ and $\mathbf{TV}^{-}(\sigma)$, which collect the positive and negative type variables of σ.

Definition 5.4 (Positive and Negative Type Variables). *The sets of* positive type variables $\mathbf{TV}^{+}(\sigma)$ *and of* negative type variables $\mathbf{TV}^{-}(\sigma)$ *of a type σ are defined as follows:*

$$\mathbf{TV}^{+}(\alpha) = \{\alpha\} \qquad \mathbf{TV}^{s}(o) = \emptyset$$
$$\mathbf{TV}^{-}(\alpha) = \emptyset \qquad \mathbf{TV}^{s}(\sigma \to \tau) = \mathbf{TV}^{\bar{s}}(\sigma) \cup \mathbf{TV}^{s}(\tau).$$

If $s = +$, then $\bar{s}$ denotes $-$; otherwise, $\bar{s}$ denotes $+$.

Definition 5.5 (Constancy and Monotonicity Rules). *The predicates* $\mathbf{K}(t)$, $\mathbf{M}^{+}(t)$, *and* $\mathbf{M}^{-}(t)$ *are inductively defined by the rules*

$$\frac{}{\mathbf{K}(x)} \qquad \frac{}{\mathbf{K}(\longrightarrow)} \qquad \frac{\tilde{\alpha} \notin \mathbf{TV}^{-}(\sigma)}{\mathbf{K}(=^{\sigma\to\sigma\to o})} \qquad \frac{\mathbf{K}(t^{\sigma\to\tau}) \quad \mathbf{K}(u^{\sigma})}{\mathbf{K}(t\,u)} \qquad \frac{\mathbf{K}(t)}{\mathbf{K}(\lambda x.\, t)}$$

$$\frac{\mathbf{K}(t)}{\mathbf{M}^{s}(t)} \qquad \frac{\mathbf{M}^{\bar{s}}(t) \quad \mathbf{M}^{s}(u)}{\mathbf{M}^{s}(t \longrightarrow u)} \qquad \frac{\mathbf{K}(t^{\sigma}) \quad \mathbf{K}(u^{\sigma})}{\mathbf{M}^{-}(t = u)} \qquad \frac{\mathbf{M}^{-}(t)}{\mathbf{M}^{-}(\forall x.\, t)} \qquad \frac{\mathbf{M}^{+}(t) \quad \tilde{\alpha} \notin \mathbf{TV}^{+}(\sigma)}{\mathbf{M}^{+}(\forall x^{\sigma}.\, t)}.$$

The rules for $\mathbf{K}$ simply traverse the term structure and ensure that equality is not used on types in which $\tilde{\alpha}$ occurs positively. The first two rules for $\mathbf{M}^{+}$ and $\mathbf{M}^{-}$ are easy to justify semantically. The other three are more subtle:

– The $\mathbf{M}^{-}(t = u)$ rule is sound because the extensions of distinct elements are always distinct (since $\sqsubseteq^{\sigma}$ is left-unique).
– The $\mathbf{M}^{-}(\forall x.\, t)$ rule is sound because if enlarging the scope makes x range over new elements, these cannot make $\forall x.\, t$ become true if it was false in the smaller scope.
– The $\mathbf{M}^{+}(\forall x.\, t)$ rule is the most difficult one. If $\tilde{\alpha}$ does not occur at all in σ, then monotonicity is preserved. Otherwise, there is the danger that the formula t is true for all values $a \in [\![\sigma]\!]_S$ but not for some $b \in [\![\sigma]\!]_{S'}$. However, in Sect. 6 we will show that this can only happen for b's that do not extend any a, which can only exist if $\alpha \in \mathbf{TV}^{+}(\sigma)$.

Example 5.4. The following derivation shows that $\forall x^{\alpha\to o}.\, P\,x$ is monotonic w.r.t. α:

$$\frac{\dfrac{\dfrac{\dfrac{}{\mathbf{K}(P)} \quad \dfrac{}{\mathbf{K}(x)}}{\mathbf{K}(P\,x)}}{\mathbf{M}^{+}(P\,x)} \qquad \alpha \notin \mathbf{TV}^{+}(\alpha \to o)}{\mathbf{M}^{+}(\forall x^{\alpha\to o}.\, P\,x)}$$

■

Example 5.5. Formula 4 from Sect. 1 is monotonic, but $\mathbf{M}^{+}$ fails on it:

$$\frac{\dfrac{\dfrac{\dfrac{\alpha \notin \mathbf{TV}^{-}(\alpha \to o)}{\mathbf{K}(=^{(\alpha\to o)\to(\alpha\to o)\to o})} \quad \dfrac{\vdots}{\mathbf{K}(\{y\})}}{\mathbf{K}((=)\ \{y\})} \qquad \dfrac{\vdots}{\mathbf{K}(\{z\})}}{\mathbf{K}(\{y\} = \{z\})}}{\mathbf{M}^{+}(\{y\} = \{z\})}$$

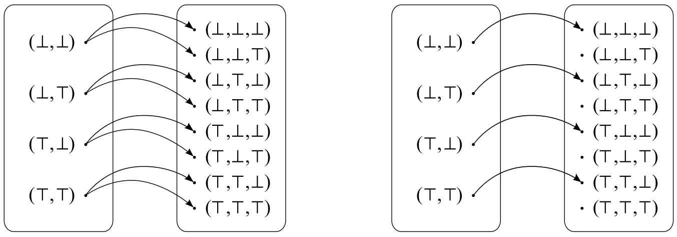
The assumption $\alpha \notin \mathbf{TV}^{-}(\alpha \to o)$ cannot be discharged, since $\mathbf{TV}^{-}(\alpha \to o) = \{\alpha\}$. ■

The last example exhibits a significant weakness of the calculus $\mathfrak{M}_{\mathsf{F}}$. HOL identifies sets with predicates, yet $\mathbf{M}^{+}$ prevents us from comparing terms of type $\tilde{\alpha} \to o$ for equality. This happens because the extension of a function of this type is not unique (cf. Fig. 1(c)), and thus equality is generally not preserved as we enlarge the scope.

This behavior of $\sqsubseteq^{\tilde{\alpha}\to o}$ is imprecise for sets, as it puts distinct sets in relation; for example, $\{0\} \sqsubseteq^{\tilde{\alpha}\to o} \{0,2\}$ if $S(\tilde{\alpha}) = 2$ and $S'(\tilde{\alpha}) = 3$. We would normally prefer each set to admit a unique extension, namely the set itself. This would make set equality constant. The next section introduces a refined calculus that formalizes this idea.

6 A Refined Calculus

To solve the problem sketched above, we introduce an alternative version of $\sqsubseteq^\sigma$ such that the extension of a set is always the set itself. Rephrased in terms of functions, this means that the extended function must return $\bot$ for all elements that are "new" in the larger scope. Fig. 2 compares this more conservative "set" approach to the liberal "functional" approach of Sect. 5; in subfigure (b), it may help to think of $(\bot,\bot)$ and $(\bot,\bot,\bot)$ as $\emptyset$, $(\top,\bot)$ and $(\top,\bot,\bot)$ as $\{0\}$, and so on.



(a) functional view (right-total) **(b)** set view (right-unique)

Fig. 2. $\sqsubseteq^{\tilde{\alpha}\to o}$ with $S(\tilde{\alpha}) = 2$ and $S'(\tilde{\alpha}) = 3$

With this approach, we could easily infer that $\{y\} = \{z\}$ is constant. However, the wholesale application of this principle would have pernicious consequences on constancy: Semantically, the universal set $\mathcal{U}^{\tilde{\alpha}\to o}$, among others, would no longer be constant; syntactically, the introduction rule for $\mathbf{K}(\lambda x.\, t)$ would no longer be sound.

What we propose instead is a hybrid approach that supports both forms of extensions in various combinations. The required bookkeeping is conveniently expressed as a type system, in which each function arrow is annotated with F ("function") or S ("set"):

Definition 6.1 (Annotated Types). *An* annotated type *is a HOL type in which each function arrow carries an* annotation $X \in \{\mathsf{F}, \mathsf{S}\}$.

The annotations have no influence on the interpretation of types as sets of values, which is unchanged. Instead, they specify how $\sqsubseteq$ should extend functional values to larger scopes. While F-functions are extended as in the previous section, the extension of an S-function must map all new values to $\bot$.

6.1 Refined Extension Relation

The extension relation $\sqsubseteq^\sigma$ distinguishes between the two kinds of arrows. The F case coincides with Def. 5.1.

Definition 6.2 (Extension Relation). *Let σ be an annotated type, and let S, S' be scopes such that $S \leq_{\tilde{\alpha}} S'$. The* extension relation $\sqsubseteq^\sigma \subseteq [\![\sigma]\!]_S \times [\![\sigma]\!]_{S'}$ *for S and S' is defined by the following equivalences:*

$$\begin{array}{lll} a \sqsubseteq^{\sigma} b & \textit{iff} & a = b \qquad\qquad\qquad\qquad \textit{if } \sigma \textit{ is } o \textit{ or a type variable} \\ f \sqsubseteq^{\sigma \to_{\mathsf{F}} \tau} g & \textit{iff} & \forall a\, b.\, a \sqsubseteq^{\sigma} b \longrightarrow f(a) \sqsubseteq^{\tau} g(b) \\ f \sqsubseteq^{\sigma \to_{\mathsf{S}} \tau} g & \textit{iff} & \forall a\, b.\, a \sqsubseteq^{\sigma} b \longrightarrow f(a) \sqsubseteq^{\tau} g(b) \textit{ and } \forall b.\, (\exists a.\, a \sqsubseteq^{\sigma} b) \vee g(b) = (\!|\tau|\!)_{S'} \end{array}$$

where $(\!|o|\!)_S = \bot$, $(\!|\sigma \to \tau|\!)_S = a \in [\![\sigma]\!]_S \mapsto (\!|\tau|\!)_S$, *and* $(\!|\alpha|\!)_S$ *is any element of* $S(\alpha)$.

Although the S annotation is tailored to predicates, the annotated type $\sigma \to_{\mathsf{S}} \tau$ is legal for any type τ. The value $(\!|\tau|\!)_S$ then takes the place of $\bot$ as the default extension.

We now prove the crucial properties of $\sqsubseteq^{\sigma}$, to which we alluded in Sect. 5. The unusual definitions of $\mathbf{TV}^{+}$ and $\mathbf{TV}^{-}$ in the S case ensure that Lem. 6.2 holds uniformly.

Lemma 6.1. *The relation* $\sqsubseteq^{\sigma}$ *is left-total and left-unique* (*injective*).

Proof (structural induction on σ). For o and α, both properties are obvious. For $\sigma \to_X \tau$, we assume by induction that $\sqsubseteq^{\sigma}$ and $\sqsubseteq^{\tau}$ are left-unique and left-total. Since $\sqsubseteq^{\sigma \to_{\mathsf{S}} \tau} \subseteq \sqsubseteq^{\sigma \to_{\mathsf{F}} \tau}$, it suffices to show that $\sqsubseteq^{\sigma \to_{\mathsf{F}} \tau}$ is left-unique and $\sqsubseteq^{\sigma \to_{\mathsf{S}} \tau}$ is left-total.

LEFT-UNIQUENESS: We assume $f,\, f' \sqsubseteq^{\sigma \to_{\mathsf{F}} \tau} g$ and show that $f = f'$. For every $a \in [\![\sigma]\!]_S$, left-totality of $\sqsubseteq^{\sigma}$ yields an extension b with $a \sqsubseteq^{\sigma} b$. Then $f(a) \sqsubseteq^{\tau} g(b)$ and $f'(a) \sqsubseteq^{\tau} g(b)$, and since $\sqsubseteq^{\tau}$ is left-unique, $f(a) = f'(a)$.

LEFT-TOTALITY: For $f \in [\![\sigma \to \tau]\!]_S$, we find an extension g as follows: Let $b \in [\![\sigma]\!]_{S'}$. If b extends an a, that a is unique by left-uniqueness of $\sqsubseteq^{\sigma}$. Since $\sqsubseteq^{\tau}$ is left-total, there exists a y such that $f(a) \sqsubseteq^{\tau} y$, and we let $g(b) = y$. If b does not extend any a, then we set $g(b) = (\!|\tau|\!)_{S'}$. By construction, $f \sqsubseteq^{\sigma \to_{\mathsf{S}} \tau} g$. □

Definition 6.3 (Positive and Negative Type Variables). *The sets of* positive type variables $\mathbf{TV}^{+}(\sigma)$ *and of* negative type variables $\mathbf{TV}^{-}(\sigma)$ *of an annotated type* σ *are defined as follows:*

$$\begin{array}{rlrl} \mathbf{TV}^{+}(o) &= \emptyset & \mathbf{TV}^{-}(o) &= \emptyset \\ \mathbf{TV}^{+}(\alpha) &= \{\alpha\} & \mathbf{TV}^{-}(\alpha) &= \emptyset \\ \mathbf{TV}^{+}(\sigma \to_{\mathsf{F}} \tau) &= \mathbf{TV}^{-}(\sigma) \cup \mathbf{TV}^{+}(\tau) & \mathbf{TV}^{-}(\sigma \to_{\mathsf{F}} \tau) &= \mathbf{TV}^{+}(\sigma) \cup \mathbf{TV}^{-}(\tau) \\ \mathbf{TV}^{+}(\sigma \to_{\mathsf{S}} \tau) &= \mathbf{TV}^{+}(\sigma) \cup \mathbf{TV}^{-}(\sigma) \cup \mathbf{TV}^{+}(\tau) & \mathbf{TV}^{-}(\sigma \to_{\mathsf{S}} \tau) &= \mathbf{TV}^{-}(\tau). \end{array}$$

Lemma 6.2. *If* $\tilde{\alpha} \notin \mathbf{TV}^{+}(\sigma)$, *then* $\sqsubseteq^{\sigma}$ *is right-total* (*surjective*). *If* $\tilde{\alpha} \notin \mathbf{TV}^{-}(\sigma)$, *then* $\sqsubseteq^{\sigma}$ *is right-unique* (*functional*).

Proof (structural induction on σ). For o and α, both properties are obvious. For $\sigma \to_X \tau$, we assume by induction that the implications hold for $\sqsubseteq^{\sigma}$ and $\sqsubseteq^{\tau}$.

RIGHT-UNIQUENESS OF $\sqsubseteq^{\sigma \to_{\mathsf{F}} \tau}$: If $\tilde{\alpha} \notin \mathbf{TV}^{-}(\sigma \to_{\mathsf{F}} \tau) = \mathbf{TV}^{+}(\sigma) \cup \mathbf{TV}^{-}(\tau)$, then by induction hypothesis $\sqsubseteq^{\sigma}$ is right-total and $\sqsubseteq^{\tau}$ is right-unique. We consider $g,\, g'$ such that $f \sqsubseteq^{\sigma \to_{\mathsf{F}} \tau} g$ and $f \sqsubseteq^{\sigma \to_{\mathsf{F}} \tau} g'$, and show that $g = g'$. For every $b \in [\![\sigma]\!]_{S'}$, right-totality of $\sqsubseteq^{\sigma}$ yields a restriction $a \sqsubseteq^{\sigma} b$. Then $f(a) \sqsubseteq^{\tau} g(b)$ and $f(a) \sqsubseteq^{\tau} g'(b)$, and since $\sqsubseteq^{\tau}$ is right-unique, $g(b) = g'(b)$.

RIGHT-UNIQUENESS OF $\sqsubseteq^{\sigma \to_{\mathsf{S}} \tau}$ AND RIGHT-TOTALITY OF $\sqsubseteq^{\sigma \to_X \tau}$: Omitted. □

Notice how the new definition of $\mathbf{TV}^{+}$ and $\mathbf{TV}^{-}$ solves the problem exhibited by the formula $\{y\} = \{z\}$ (Ex. 5.5), since the $\tilde{\alpha}$ in $\tilde{\alpha} \to_{\mathsf{S}} o$ counts as a positive occurrence. However, we must now ensure that types are consistently annotated.

6.2 Type Checking

Checking constancy can be seen as a type checking problem involving annotated types. The basic idea is to derive typing judgments $\Gamma \vdash t : \sigma$, whose intuitive meaning is that the denotations of t in a smaller and a larger scope are related by $\sqsubseteq^\sigma$ (i.e., that t is constant in a sense given by σ). Despite this new interpretation, the typing rules are similar to those of the simply-typed λ-calculus, extended with a particular form of subtyping.

Regrettably, our rules cannot derive the desired types for the basic set operations $\cup$, $\cap$, and $-$ when they are defined as abbreviations (cf. Notat. 3.2). This problem is solved by treating set constants as primitive along with implication and equality.

Definition 6.4 (Context). *A* context *is a pair of mappings* $\Gamma = (\Gamma_\mathrm{c}, \Gamma_\mathrm{v})$, *where* Γ_c *maps constant symbols to sets of annotated types, and* Γ_v *maps variables to annotated types. A constant context* Γ_c *is* compatible *with a constant model* M *if* $\sigma \in \Gamma_\mathrm{c}(c)$ *implies* $M_S(c) \sqsubseteq^\sigma M_{S'}(c)$ *for all scopes* S, S' *with* $S \leq_{\tilde{\alpha}} S'$ *and for all constants* c *and types* σ.

Definition 6.5 (Standard Constant Context). *The* standard constant context $\widehat{\Gamma}_\mathrm{c}$ *is the following mapping:*

$$\begin{array}{rcll}
\longrightarrow & \mapsto & \{o \rightarrow_\mathsf{F} o \rightarrow_\mathsf{F} o\} & \\
= & \mapsto & \{\sigma \rightarrow_\mathsf{F} \sigma \rightarrow_X o & \mid X \in \{\mathsf{F},\mathsf{S}\},\ \tilde{\alpha} \notin \mathbf{TV}^-(\sigma)\} \\
\emptyset & \mapsto & \{\sigma \rightarrow_X o & \mid X \in \{\mathsf{F},\mathsf{S}\}\} \\
\mathcal{U} & \mapsto & \{\sigma \rightarrow_\mathsf{F} o\} & \\
\cup & \mapsto & \{(\sigma \rightarrow_X o) \rightarrow_\mathsf{F} (\sigma \rightarrow_X o) \rightarrow_\mathsf{F} \sigma \rightarrow_X o & \mid X \in \{\mathsf{F},\mathsf{S}\}\} \\
\cap & \mapsto & \{(\sigma \rightarrow_X o) \rightarrow_\mathsf{F} (\sigma \rightarrow_X o) \rightarrow_\mathsf{F} \sigma \rightarrow_X o & \mid X \in \{\mathsf{F},\mathsf{S}\}\} \\
- & \mapsto & \{(\sigma \rightarrow_X o) \rightarrow_\mathsf{F} (\sigma \rightarrow_\mathsf{F} o) \rightarrow_\mathsf{F} \sigma \rightarrow_X o & \mid X \in \{\mathsf{F},\mathsf{S}\}\} \\
\in & \mapsto & \{\sigma \rightarrow_\mathsf{F} (\sigma \rightarrow_X o) \rightarrow_\mathsf{F} o & \mid X \in \{\mathsf{F},\mathsf{S}\}\} \\
\mathit{insert} & \mapsto & \{\sigma \rightarrow_\mathsf{F} (\sigma \rightarrow_X o) \rightarrow_\mathsf{F} \sigma \rightarrow_X o & \mid X \in \{\mathsf{F},\mathsf{S}\},\ \tilde{\alpha} \notin \mathbf{TV}^-(\sigma)\}.
\end{array}$$

Allowing constants to have multiple annotated types gives us a form of polymorphism on the annotations. We treat $\widehat{\Gamma}_\mathrm{c}$ as a global table of annotated types for constants.

Lemma 6.3. *The standard constant context is compatible with the standard constant model.*

Proof. CASE $=$: Since $\tilde{\alpha} \notin \mathbf{TV}^-(\sigma)$, $\sqsubseteq^\sigma$ is right-unique (Lem. 6.2). Unfolding the definition of $\sqsubseteq$, we assume $a \sqsubseteq^\sigma b$ and show that if $a' \sqsubseteq^\sigma b'$, then $(a = a') = (b = b')$, and that if there exists no restriction a' such that $a' \sqsubseteq^\sigma b'$, then $(b = b') = \bot$. The first part follows from the left-uniqueness and right-uniqueness of $\sqsubseteq^\sigma$. For the second part, $b \neq b'$ because a restricts b but b' admits no restriction.

OTHER CASES: Omitted. □

Defs. 5.2 and 5.3 and the **K** part of Def. 5.5 from Sect. 5 are generalized as follows.

Definition 6.6 (Model Extension). *Let* $\mathcal{M} = (S, A)$ *and* $\mathcal{M}' = (S', A')$ *be models. The model* $\mathcal{M}'$ extends $\mathcal{M}$ *in a context* Γ, *written* $\mathcal{M} \sqsubseteq_\Gamma \mathcal{M}'$, *if* $S \leq_{\tilde{\alpha}} S'$ *and* $\Gamma_\mathrm{v}(x) = \sigma$ *implies* $A(x) \sqsubseteq^\sigma A'(x)$ *for all* x.

Definition 6.7 (Constancy). *Let σ be an annotated type. A term t is σ-*constant *in a context Γ if $[\![t]\!]_{\mathcal{M}} \sqsubseteq^{\sigma} [\![t]\!]_{\mathcal{M}'}$ for all models $\mathcal{M}$, $\mathcal{M}'$ such that $\mathcal{M} \sqsubseteq_{\Gamma} \mathcal{M}'$.*

Definition 6.8 (Typing Rules). *The typing relation $\Gamma \vdash t : \sigma$ is given by the rules*

$$\frac{\Gamma_{\mathrm{v}}(x) = \sigma}{\Gamma \vdash x : \sigma}\ \textsc{Var} \qquad \frac{\sigma \in \Gamma_{\mathrm{c}}(c)}{\Gamma \vdash c : \sigma}\ \textsc{Const}$$

$$\frac{\Gamma \vdash t : \sigma' \rightarrow_{\mathsf{X}} \tau \quad \Gamma \vdash u : \sigma \quad \sigma \leq \sigma'}{\Gamma \vdash t\,u : \tau}\ \textsc{App} \qquad \frac{\Gamma[x \mapsto \sigma] \vdash t : \tau}{\Gamma \vdash \lambda x.\, t : \sigma \rightarrow_{\mathsf{F}} \tau}\ \textsc{Lam}$$

where $X \in \{\mathsf{F}, \mathsf{S}\}$ and the subtype relation $\sigma \leq \tau$ is defined by the rules

$$\frac{}{o \leq o} \qquad \frac{}{\alpha \leq \alpha} \qquad \frac{\sigma' \leq \sigma \quad \tau \leq \tau'}{\sigma \rightarrow_{\mathsf{X}} \tau \leq \sigma' \rightarrow_{\mathsf{F}} \tau'} \qquad \frac{\tau \leq \tau'}{\sigma \rightarrow_{\mathsf{S}} \tau \leq \sigma \rightarrow_{\mathsf{S}} \tau'}.$$

In the above definition, $\Gamma[x \mapsto \sigma]$ abbreviates $(\Gamma_{\mathrm{c}}, \Gamma_{\mathrm{v}}[x \mapsto \sigma])$.

Lemma 6.4. *If $\sigma \leq \sigma'$, then $\sqsubseteq^{\sigma} \subseteq \sqsubseteq^{\sigma'}$.*

Proof. By straightforward induction on the derivation of $\sigma \leq \sigma'$. □

Theorem 6.1 (Soundness of Typing). *If $\Gamma \vdash t : \sigma$, then t is σ-constant in Γ.*

Proof (induction on the derivation of $\Gamma \vdash t : \sigma$)

VAR: Obvious, since $A(x) \sqsubseteq^{\sigma} A'(x)$ by assumption for $\sigma = \Gamma_{\mathrm{v}}(x)$.

CONST: Obvious, since $\widehat{M}_S(c) \sqsubseteq^{\sigma} \widehat{M}_{S'}(c)$ by assumption for all $\sigma \in \Gamma_{\mathrm{c}}(c)$.

APP: By induction hypothesis, and since $\sqsubseteq^{\sigma' \rightarrow_{\mathsf{S}} \tau} \subseteq \sqsubseteq^{\sigma' \rightarrow_{\mathsf{F}} \tau}$, we have $[\![t]\!]_{\mathcal{M}} \sqsubseteq^{\sigma' \rightarrow_{\mathsf{F}} \tau} [\![t]\!]_{\mathcal{M}'}$ and $[\![u]\!]_{\mathcal{M}} \sqsubseteq^{\sigma} [\![u]\!]_{\mathcal{M}'}$. Lem. 6.4 and the condition $\sigma \leq \sigma'$ imply that $[\![u]\!]_{\mathcal{M}} \sqsubseteq^{\sigma'} [\![u]\!]_{\mathcal{M}'}$. Then by Def. 6.2, we know that $[\![t\,u]\!]_{\mathcal{M}} = [\![t]\!]_{\mathcal{M}}([\![u]\!]_{\mathcal{M}}) \sqsubseteq^{\tau} [\![t]\!]_{\mathcal{M}'}([\![u]\!]_{\mathcal{M}'}) = [\![t\,u]\!]_{\mathcal{M}'}$, which shows that $t\,u$ is τ-constant in Γ.

LAM: Let $a \in [\![\sigma]\!]_S$ and $b \in [\![\sigma]\!]_{S'}$ such that $a \sqsubseteq^{\sigma} b$. Then we have the extended models $\mathcal{M}_a = (S, A[x \mapsto a])$ and $\mathcal{M}'_b = (S', A'[x \mapsto b])$. Thus, $\mathcal{M}_a \sqsubseteq_{\Gamma[x \mapsto \sigma]} \mathcal{M}'_b$, and by induction hypothesis $[\![\lambda x.\ t]\!]_{\mathcal{M}}(a) = [\![t]\!]_{\mathcal{M}_a} \sqsubseteq^{\tau} [\![t]\!]_{\mathcal{M}'_b} = [\![\lambda x.\ t]\!]_{\mathcal{M}'}(b)$. This implies $[\![\lambda x.\ t]\!]_{\mathcal{M}} \sqsubseteq^{\sigma \rightarrow_{\mathsf{F}} \tau} [\![\lambda x.\ t]\!]_{\mathcal{M}'}$. □

6.3 Monotonicity Checking

The rules for monotonicity and antimonotonicity are almost the same as in the previous section, except that they now extend the context when moving under a quantifier.

Definition 6.9 (Monotonicity Rules). *The predicates $\Gamma \vdash \mathbf{M}^{+}(t)$ and $\Gamma \vdash \mathbf{M}^{-}(t)$ are given by the rules*

$$\frac{\Gamma \vdash t : o}{\Gamma \vdash \mathbf{M}^{s}(t)}\ \textsc{Term} \qquad \frac{\Gamma \vdash \mathbf{M}^{\bar{s}}(t) \quad \Gamma \vdash \mathbf{M}^{s}(u)}{\Gamma \vdash \mathbf{M}^{s}(t \longrightarrow u)}\ \textsc{Imp} \qquad \frac{\Gamma \vdash t : \sigma \quad \Gamma \vdash u : \sigma}{\mathbf{M}^{-}(t = u)}\ \textsc{Eq}^{-}$$

$$\frac{\Gamma[x \mapsto \sigma] \vdash \mathbf{M}^{-}(t)}{\Gamma \vdash \mathbf{M}^{-}(\forall x.\, t)}\ \textsc{All}^{-} \qquad \frac{\Gamma[x \mapsto \sigma] \vdash \mathbf{M}^{+}(t) \quad \tilde{\alpha} \notin \mathbf{TV}^{+}(\sigma)}{\Gamma \vdash \mathbf{M}^{+}(\forall x.\, t)}\ \textsc{All}^{+}.$$

Theorem 6.2 (Soundness of $\mathbf{M}^s$). *Let $\mathcal{M}$ and $\mathcal{M}'$ be models such that $\mathcal{M} \sqsubseteq_\Gamma \mathcal{M}'$. If $\Gamma \vdash \mathbf{M}^+(t)$, then $\mathcal{M} \vDash t$ implies $\mathcal{M}' \vDash t$. If $\Gamma \vdash \mathbf{M}^-(t)$, then $\mathcal{M} \nvDash t$ implies $\mathcal{M}' \nvDash t$.*

Proof (induction on the derivation of $\Gamma \vdash \mathbf{M}(t)$). Let $\mathcal{M} = (S, A)$ and $\mathcal{M}' = (S', A')$.

TERM: Because constancy implies (anti)monotonicity for type o.

IMP: Obvious.

EQ$^-$: Assume that $\Gamma \vdash s : \sigma$, $\Gamma \vdash t : \sigma$, and $\mathcal{M} \nvDash s = t$. Since $\mathcal{M}$ is a standard model, we know that $[\![s]\!]_\mathcal{M} \neq [\![t]\!]_\mathcal{M}$. By Thm. 6.1, we have $[\![s]\!]_\mathcal{M} \sqsubseteq^\sigma [\![s]\!]_{\mathcal{M}'}$ and $[\![t]\!]_\mathcal{M} \sqsubseteq^\sigma [\![t]\!]_{\mathcal{M}'}$. By the left-uniqueness of $\sqsubseteq^\sigma$, the extensions cannot be equal, and thus $\mathcal{M}' \nvDash s = t$.

ALL$^-$: Assume that $\Gamma[x \mapsto \sigma] \vdash \mathbf{M}^-(t)$ and $\mathcal{M} \nvDash \forall x^\sigma.\, t$. Then there exists $a \in [\![\sigma]\!]_S$ such that $(S, A[x \mapsto a]) \nvDash t$. Since $\sqsubseteq^\sigma$ is left-total, there exists an extension $b \in [\![\sigma]\!]_{S'}$ with $a \sqsubseteq^\sigma b$. Since $(S, A[x \mapsto a]) \sqsubseteq_\Gamma (S', A'[x \mapsto b])$, we can conclude $(S', A'[x \mapsto b]) \nvDash t$ by induction hypothesis. Thus $\mathcal{M}' \nvDash \forall x^\sigma.\, t$.

ALL$^+$: Assume that $\Gamma[x \mapsto \sigma] \vdash \mathbf{M}^+(t)$, $\tilde{\alpha} \notin \mathbf{TV}^+(\sigma)$, and $\mathcal{M} \vDash \forall x^\sigma.\, t$. We show that $\mathcal{M}' \vDash \forall x^\sigma.\, t$. Let $b \in [\![\sigma]\!]_{S'}$. Since $\sqsubseteq^\sigma$ is right-total (Lem. 6.2), there exists a restriction $a \in [\![\sigma]\!]_S$ with $a \sqsubseteq^\sigma b$. By assumption, $(S, A[x \mapsto a]) \vDash t$. Since $(S, A[x \mapsto a]) \sqsubseteq_\Gamma (S', A'[x \mapsto b])$, we can conclude $(S', A'[x \mapsto b]) \vDash t$ by induction hypothesis. □

Corollary 6.1 (Soundness of $\mathfrak{M}_{\mathsf{FS}}$). *If $\Gamma \vdash \mathbf{M}^+(t)$ can be derived in some arbitrary context Γ, then t is monotonic. If $\Gamma \vdash \mathbf{M}^-(t)$ can be derived in some arbitrary context Γ, then t is antimonotonic.*

Example 6.1. Let $\{\alpha\}$ stand for $\alpha \to_\mathsf{S} o$, and let $\Gamma_\mathrm{V} = [x \mapsto \{\alpha\}, y \mapsto \{\alpha\}]$. The following derivation shows that $x^{\alpha \to o} = y$ is monotonic w.r.t. α:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma(=) = \{\alpha\} \to_\mathsf{F} \{\alpha\} \to_\mathsf{F} o}}{\Gamma \vdash (=) : \{\alpha\} \to_\mathsf{F} \{\alpha\} \to_\mathsf{F} o} \quad \dfrac{\dfrac{}{\Gamma(x) = \{\alpha\}}}{\Gamma \vdash x : \{\alpha\}} \quad \dfrac{}{\{\alpha\} \leq \{\alpha\}}}{\Gamma \vdash (=)\ x : \{\alpha\} \to_\mathsf{F} o} \quad \dfrac{\dfrac{}{\Gamma(y) = \{\alpha\}}}{\Gamma \vdash y : \{\alpha\}} \quad \dfrac{}{\{\alpha\} \leq \{\alpha\}}}{\dfrac{\Gamma \vdash x = y : o}{\Gamma \vdash \mathbf{M}^+(x = y)}}$$
■

Example 6.2. The following table lists some example formulas, including those from Sect. 1. For each formula, we indicate whether it is monotonic or antimononic w.r.t. α according to the calculi $\mathfrak{M}_\mathsf{F}$ and $\mathfrak{M}_\mathsf{FS}$ and to the semantic definitions.

	MONOTONIC			ANTIMONOTONIC		
FORMULA	$\mathfrak{M}_\mathsf{F}$	$\mathfrak{M}_\mathsf{FS}$	SEM.	$\mathfrak{M}_\mathsf{F}$	$\mathfrak{M}_\mathsf{FS}$	SEM.
$\exists x^\alpha\, y.\; x \neq y$	✓	✓	✓	·	·	·
$f\, x^\alpha = x \wedge f\, y \neq y$	✓	✓	✓	✓	✓	✓
$x^{o \to \alpha} = y$	✓	✓	✓	✓	✓	✓
$s^{\alpha \to o} = t$	·	✓	✓	✓	✓	✓
$\{y^\alpha\} = \{z\}$	·	✓	✓	✓	✓	✓
$(\lambda x^\alpha.\; x = y) = (\lambda x.\; x = z)$	·	·	✓	✓	✓	✓
$(\forall x^\alpha.\; f\, x = x) \wedge f\, y \neq y$	·	·	✓	✓	✓	✓
$\forall x^\alpha\, y.\; x = y$	·	·	·	✓	✓	✓
$\exists x^\alpha\, y.\; x \neq y \wedge \forall z.\; z = x \vee z = y$	·	·	·	·	·	·

■

6.4 Type Inference

Expecting all types to be fully annotated with F and S is unrealistic, so we now face the problem of computing annotations such that a given term is typable—a type inference problem. We follow a standard approach to type inference: We start by annotating all types with *annotation variables* ranging over $\{\mathsf{F},\mathsf{S}\}$. Then we construct a typing derivation, collecting a set of constraints over the annotations. Finally, we look for an instantiation for the annotation variables that satisfies all the constraints.

Definition 6.10 (Annotation Constraints). *An* annotation constraint *over a set of annotation variables V is an expression of the form $\sigma \leq \tau$, $\tilde{\alpha} \notin \mathbf{TV}^{+}(\sigma)$, or $\tilde{\alpha} \notin \mathbf{TV}^{-}(\sigma)$, where the types σ and τ may contain annotation variables in V. Given a valuation $\rho : V \to \{\mathsf{F},\mathsf{S}\}$, the meaning of a constraint is defined as in Defs. 6.3 and 6.8.*

A straightforward way of solving such constraints is to encode them in propositional logic, following Defs. 6.3 and 6.8, and give them to a SAT solver. Annotation variables, which may take two values, are mapped directly to propositional variables. This approach proved very efficient on the problems that we encountered in our experiments.

So far, we have been unable to prove that the satisfiability problem for this constraint language is NP-complete. We suspect that it is not, but we have not found a polynomial-time algorithm. Thus, it is unclear if our use of a SAT solver is fully appropriate from a theoretical point of view, even though it works perfectly well in practice.

7 Inductive Datatypes

To make monotonicity checking useful in practice, we must support user-defined types, which we have ignored so far. The most important way of introducing new types in Isabelle/HOL is to declare an inductive datatype using the command

$$\textbf{datatype}\ \bar{\alpha}\,\kappa\ =\ C_1\,\sigma_{11}\ldots\sigma_{1k_1}\ \mid\ \cdots\ \mid\ C_n\,\sigma_{n1}\ldots\sigma_{nk_n}$$

Inductive datatypes are a derived concept in HOL [2]. However, our analysis benefits from treating them specially as opposed to unfolding the underlying construction.

The datatype declaration introduces the type constructor κ, together with the term constructors C_i of type $\sigma_{i1} \to_{\mathsf{F}} \cdots \to_{\mathsf{F}} \sigma_{ik_i} \to_{\mathsf{F}} \bar{\alpha}\,\kappa$. The type $\bar{\alpha}\,\kappa$ may occur recursively in the σ_{ij}'s, but only in positive positions. For simplicity, we assume that any arrows in the σ_{ij}'s are annotated with F or S. (In the implementation, annotation variables are used to infer the annotations.) The interpretation $[\![\bar{\alpha}\,\kappa]\!]_S$ of the new type is given by the corresponding free term algebra.

We must now extend the basic definitions of $\sqsubseteq$, $\leq$, and $\mathbf{TV}^s$ to this new construct. For Def. 6.2, we add the following case:

$$C_i(a_1,\ldots,a_{k_i}) \sqsubseteq^{\bar{\tau}\,\kappa} C_i(b_1,\ldots,b_{k_i}) \quad \textit{iff}\ \forall j \in \{1,\ldots,k_i\}.\ a_j \sqsubseteq^{\sigma_{ij}[\bar{\alpha}\mapsto\bar{\tau}]} b_j.$$

Similarly, Def. 6.3 is extended with

$$\mathbf{TV}^s(\bar{\tau}\,\kappa) = \bigcup_{\substack{1\leq i\leq n\\ 1\leq j\leq k_i}} \mathbf{TV}^s(\sigma_{ij}[\bar{\alpha}\mapsto\bar{\tau}])$$

and Def. 6.8 with

$$\frac{\sigma_{ij}[\bar{\alpha} \mapsto \bar{\tau}] \leq \sigma_{ij}[\bar{\alpha} \mapsto \bar{\tau}'] \quad \textit{for all } 1 \leq i \leq n,\, 1 \leq j \leq k_i}{\bar{\tau}\;\kappa \leq \bar{\tau}'\;\kappa}\,.$$

To extend our soundness result, we must show that Lems. 6.1 to 6.4 still hold. The proofs are straightforward and omitted from this paper. Constancy of the datatype constructors also follows directly from the above definitions.

8 Evaluation

What proportion of monotonic formulas are detected as such by our calculi? We applied Nitpick's implementations of $\mathfrak{M}_{\mathsf{F}}$ and (a large fragment of) $\mathfrak{M}_{\mathsf{FS}}$ on the user-supplied theorems from six highly polymorphic Isabelle theories. In the spirit of counterexample generation, we conjoined the negated theorems with the relevant axioms. The results are given below.

Theory	Formulas		Success rate	
	$\mathfrak{M}_{\mathsf{F}}$	$\mathfrak{M}_{\mathsf{FS}}$	$\mathfrak{M}_{\mathsf{F}}$	$\mathfrak{M}_{\mathsf{FS}}$
AVL2	$18_{/24}$	$22_{/24}$	75.0%	91.7%
Fun	$49_{/87}$	$71_{/87}$	56.3%	81.6%
Huffman	$41_{/94}$	$86_{/94}$	43.6%	91.5%
List	$266_{/524}$	$402_{/524}$	50.8%	76.7%
Map	$94_{/97}$	$97_{/97}$	96.9%	100.0%
Relation	$59_{/144}$	$94_{/144}$	41.0%	65.3%

The table indicates how many formulas were found to involve at least one monotonic type variable using $\mathfrak{M}_{\mathsf{F}}$ and $\mathfrak{M}_{\mathsf{FS}}$, respectively, over the total number of formulas involving type variables in the six theories. Since the formulas are all negated theorems, they are all semantically monotonic (no models exist for any scope).

An ideal way to assess the calculi would have been to try them on a representative database including non-theorems, but we lack such a database. Nonetheless, our experience suggests that the calculi perform as well on non-theorems as on theorems, because realistic non-theorems tend to use equality and quantifiers in essentially the same way as theorems. Interestingly, non-theorems that are derived from theorems by omitting an assumption or mistyping a variable name are even more likely to pass the monotonicity check than the corresponding theorems.

Although the study of monotonicity is interesting in its own right and leads to an elegant theory, our main motivation—speeding up model finders—is resolutely pragmatic. For Nitpick, which uses a default upper bound of 8 on the cardinality of the atomic types, we observed a speed increase factor of about 5 per inferred monotonic type. Since each monotonic type reduces the number of scopes to consider by a factor of 8, we could perhaps expect an 8-fold speed increase; however, the scopes that can be omitted by exploiting monotonicity are smaller and faster to check than those that are actually checked. The time spent performing the monotonicity analysis (i.e., generating the annotation constraints and solving the resulting SAT problem) is negligible.

9 Discussion

Fully covariant arrow. In Def. 6.3, both the positive and the negative type variables in σ count as positive occurrences in $\sigma \rightarrow_{\mathsf{S}} \tau$. This raises the question of whether a fully covariant behavior, with $\mathbf{TV}^{s}(\sigma \rightarrow_{\mathsf{S}} \tau) = \mathbf{TV}^{s}(\sigma) \cup \mathbf{TV}^{s}(\tau)$, can also be achieved, possibly with a different definition of $\sqsubseteq^{\sigma \rightarrow_{\mathsf{S}} \tau}$. Although such a behavior looks more regular, it would make the calculus unsound, as the following counterexample shows.

Consider the formula $t \equiv \forall F^{(\alpha \to o) \to o}\, f^{\alpha \to o}\, g\, h.\; f \in F \wedge g \in F \wedge f\, a \neq g\, a \longrightarrow h \in F$. The formula is not monotonic w.r.t. α: Regardless of the value of the free variable a, it is true for $|\alpha| = 1$, since the assumptions imply that $f \neq g$, and as there are only two functions of type $\alpha \to o$, h can only be one of them, so it must be in F. This argument breaks down for larger scopes, so the formula is not monotonic. However, with a fully covariant S-arrow, we could type F as $F^{(\alpha \rightarrow_{\mathsf{F}} o) \rightarrow_{\mathsf{S}} o}$ and the rule $\textsc{All}^{+}$ would apply, since there is no positive occurrence of α in the types of F, f, g, and h.

Principal types. Similar to the simply-typed λ-calculus, our type system admits principal types if we promote annotation variables to first-class citizens. When performing type inference, we would then keep the constraints as part of the type, instead of computing an arbitrary solution for the collected constraints. More precisely, a *type schema* has the form $\forall \overline{X}.\, \forall \overline{\alpha}.\; \sigma\; \langle C \rangle$, where σ is an annotated type containing annotation variables $\overline{X}$ and type variables $\overline{\alpha}$, and C is a list of constraints of the form given in Def. 6.10. As an example, equality has the principal type schema $\forall \alpha.\; \alpha \rightarrow_{\mathsf{F}} \alpha \rightarrow_{X} o\; \langle \tilde{\alpha} \notin \mathbf{TV}^{-}(\alpha) \rangle$. This approach nicely extends ML-style polymorphism.

Set comprehensions. An obvious weakness of our type system is that the rule $\textsc{Lam}$ always types λ-abstractions with F-arrows. The only way to construct terms whose type contains S annotations is by building them from a set of primitives whose types are justified semantically. This solution is far from optimal. To take just one example, consider the term $\lambda x\, z.\; \exists y.\; R\, x\, y \wedge S\, y\, z$, which composes two binary relations R and S. Semantically, composition is constant for type $(\alpha \rightarrow_{\mathsf{S}} \beta \rightarrow_{\mathsf{S}} o) \rightarrow_{\mathsf{F}} (\beta \rightarrow_{\mathsf{S}} \gamma \rightarrow_{\mathsf{S}} o) \rightarrow_{\mathsf{F}} \alpha \rightarrow_{\mathsf{S}} \gamma \rightarrow_{\mathsf{S}} o$, but our analysis cannot infer this. As a consequence, our analysis cannot infer the monotonicity of any of the four type variables occurring in the associativity law for composition, unless composition is added to the constant context Γ_{c}.

10 Conclusion

In model finders that work by enumerating scopes (domain cardinalities specifications), the choice of the scopes and their order is critical to obtain good performance. Yet, little work has been done on this problem beyond the discovery of small model theorems.

We presented a solution for HOL that prunes the search space by inferring monotonicity with respect to atomic types. Monotonicity is in general undecidable, so we approximate it with syntactic criteria. The main difficulty occurs in conjunction with common set idioms, which we detect using a suitable type system. Our approach also handles datatypes defined in terms of the atomic types. A direction for future research would be to extend the type system to handle more syntactic idioms (e.g., set comprehensions and “almost full” sets such as $\mathcal{U} - \{x\}$), thereby strengthening the analysis.

Our measurements show that monotonic formulas are pervasive in HOL formalizations and that syntactic criteria can detect them more often than not. Our more powerful calculus $\mathfrak{M}_{\mathsf{FS}}$ has been implemented as part of Isabelle's SAT-based counterexample generator Nitpick, with dramatic speed gains. It will be interesting to see whether this success can be repeated in the context of other model finders.

Acknowledgment. We would like to thank Lukas Bulwahn, Tobias Nipkow, Mark Summerfield, and the anonymous reviewers for suggesting several textual improvements.

References

1. Andrews, P.B.: An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof, 2nd edn. Applied Logic, vol. 27. Springer, Heidelberg (2002)
2. Berghofer, S., Wenzel, M.: Inductive datatypes in HOL—lessons learned in formal-logic engineering. In: Bertot, Y., Dowek, G., Hirschowitz, A., Paulin, C., Théry, L. (eds.) TPHOLs 1999. LNCS, vol. 1690, pp. 19–36. Springer, Heidelberg (1999)
3. Blanchette, J.C., Nipkow, T.: Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In: Kaufmann, M., Paulson, L. (eds.) ITP-10. LNCS. Springer, Heidelberg (to appear, 2010)
4. Claessen, K.: Private communication (2009)
5. Claessen, K., Sörensson, N.: New techniques that improve MACE-style model finding. In: MODEL (2003)
6. Gordon, M.J.C., Melham, T.F. (eds.): Introduction to HOL: A Theorem Proving Environment for Higher Order Logic. Cambridge University Press, Cambridge (1993)
7. Harrison, J.: HOL Light: A tutorial introduction. In: Srivas, M., Camilleri, A. (eds.) FMCAD 1996. LNCS, vol. 1166, pp. 265–269. Springer, Heidelberg (1996)
8. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press, Cambridge (2006)
9. Jackson, D., Shlyakhter, I., Sridharan, M.: A micromodularity mechanism. In: FSE/ESEC 2001, pp. 62–73 (2001)
10. Kuncak, V., Jackson, D.: Relational analysis of algebraic datatypes. In: Gall, H.C. (ed.) ESEC/FSE 2005 (2005)
11. McCune, W.: A Davis–Putnam program and its application to finite first-order model search: Quasigroup existence problems. Technical report, ANL (1994)
12. Momtahan, L.: Towards a small model theorem for data independent systems in Alloy. ENTCS 128(6), 37–52 (2005)
13. Nipkow, T.: Verifying a hotel key card system. In: Barkaoui, K., Cavalcanti, A., Cerone, A. (eds.) ICTAC 2006. LNCS, vol. 4281, pp. 1–14. Springer, Heidelberg (2006)
14. Nipkow, T., Paulson, L.C., Wenzel, M. (eds.): Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)
15. Pnueli, A., Rodeh, Y., Strichman, O., Siegel, M.: The small model property: How small can it be? Inf. Comput. 178(1), 279–293 (2002)
16. 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)
17. Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 632–647. Springer, Heidelberg (2007)
18. Weber, T.: SAT-Based Finite Model Generation for Higher-Order Logic. Ph.D. thesis, Dept. of Informatics, T.U. München (2008)
19. Zhang, J., Zhang, H.: SEM: A system for enumerating models. In: Kaufmann, M. (ed.) IJCAI 95, vol. 1, pp. 298–303 (1995)

Sledgehammer: Judgement Day

Sascha Böhme[*] and Tobias Nipkow

Institut für Informatik, Technische Universität München

Abstract. Sledgehammer, a component of the interactive theorem prover Isabelle, finds proofs in higher-order logic by calling the automated provers for first-order logic E, SPASS and Vampire. This paper is the largest and most detailed empirical evaluation of such a link to date. Our test data consists of 1240 proof goals arising in 7 diverse Isabelle theories, thus representing typical Isabelle proof obligations. We measure the effectiveness of Sledgehammer and many other parameters such as run time and complexity of proofs. A facility for minimizing the number of facts needed to prove a goal is presented and analyzed.

1 Introduction

Sledgehammer (**SH**) [4,5,7], developed by Paulson *et al.* at Cambridge, is a linkup of the interactive theorem prover (**ITP**) Isabelle/HOL [6] (for higher-order logic) and automated first-order provers (**ATP**s). The purpose of this paper is to evaluate SH in various dimensions, but in particular w.r.t. effectiveness: How much benefit can the average Isabelle user expect from SH? *What's in it for Joe Proof?*

In addition to evaluating effectiveness, this paper also presents data for determining weaknesses of SH, improving SH and developing it further. Some of our data have already influenced the SH setup.

Based on a large sample of typical Isabelle proofs, the following aspects of SH are analyzed: success rates; complications when turning ATP proofs into Isabelle proofs; run times; size and difficulty of proofs. We also describe and analyze a new addition to SH for reducing the number of facts used in proofs.

We believe that our analysis is of interest to both the ATP and ITP community. Many of the issues faced by SH will be faced by any linkup with ATPs where a) there is a non-trivial translation from some logic H into the language of the ATPs and b) proofs delivered by the ATPs are translated back and checked as proofs in H. Invariably, such back-and-forth translations introduce problems and lead to lost proofs, one of the issues we investigate in detail.

For a management summary of our findings, please fast-forward to §8.

See `http://www.in.tum.de/~nipkow/pubs/ijcar10.html` for our test data and log files. Many of these problems have been included into the TPTP library (see `http://www.tptp.org`).

[*] Research supported by BMBF project *Verisoft XT*.

1.1 Related Work

We do not give a comprehensive overview of combinations of interactive and automatic theorem provers. Relevant related work is already discussed in publications by Paulson *et al.* The main focus of our paper is empirical evaluation. Meng and Paulson present statistics in their reports on the development of SH, too, but those were based on hand-picked problems primarily meant for tuning the setup. This paper is an independent evaluation with a large set of test data coming from a wider collection of theories (= modules for definitions and proofs) in the Isabelle distribution, with no hand selection: all goals arising in the Isabelle proofs are passed to the ATPs.

Another large scale empirical evaluation of automating ITP proofs with ATPs is reported for the Mizar system by Urban [13].

On the empirical side, the CASC [11] has been setting standards for many years now and we refer to it throughout the text. Apart from differences in what data is analyzed, the main difference to CASC is attitude or perspective:

> Although we look at success rates and other data of individual ATPs, SH is not a competition among ATPs but between ATPs and Isabelle: how much can ATPs do automatically where Isabelle requires nontrivial human interaction?

2 Sledgehammer

When SH is activated by an Isabelle user to prove some goal G, it performs the following steps:

1. Based on the syntactic form of G, a set S of relevant **facts** (axioms and lemmas) is extracted from the background theory, that is, all the knowledge available to the user at this point. S typically contains hundreds of facts.
2. $S \cup \{\neg G\}$ is translated from HOL (with a Haskell-like type system) into untyped first-order logic (FOL).
3. One or more ATPs are called.
4. If one of the ATPs returns a proof of G, i.e. a refutation of $\neg G$, SH extracts the facts $R \subseteq S$ used in the proof.

Strictly speaking, at this point SH is over, but we regard the next step as part of SH: The user is given the option to call the internal ATP *Metis* (**M**) that will try to prove G with the help of R. M was designed and implemented by Hurd [3] and adapted for Isabelle by Paulson and Susanto [7]. M is written in SML and produces actual Isabelle/HOL proofs, the whole point of the setup. This double-checking is necessary because the translation from HOL to FOL is potentially unsound (see §4.1). M is slow compared to the leading ATPs, but its performance at CASC is respectable. Typically R contains only a few facts, in rare cases more than 20. As a result, the final step in the SH process, calling M with R, succeeds surprisingly often — more below. Successful M calls become part of the Isabelle proof text. We call the M step **proof reconstruction** because M has to search for a proof once more, but w.r.t. a very focussed set of facts. Essentially, SH uses the ATPs as very powerful relevance filters for M.

A recent enhancement of SH (implemented by Fabian Immler at TUM) is the ability to call ATPs remotely via Sutcliffe's *SystemOnTPTP* [12].

3 The Setup

Our test data are 7 theories from the Isabelle distribution and the Archive of Formal Proofs (afp.sf.net) representative for a range of typical applications. All theories were developed without the use of SH.

Arrow	Arrow's impossibility theorem	8%	LS
NS	Needham-Schroeder shared-key protocol	8%	I
Hoare	Completeness of Hoare logic with procedures	16%	IL
Jinja	Type soundness of a subset of Java	13%	IL
SN	SN of typed λ-calculus with de Bruijn indices	9%	AI
FTA	Fundamental Theorem of Algebra	34%	A
FFT	Fast Fourier Transform	12%	A L

The leftmost column shows the names used below, the % column what percentage of the overall 1240 problems/goals come from each theory, and the rightmost column the logical nature of each theory: A means arithmetic; I means that the theory is dominated by recursive functions defined by equations and inductive relations defined by Horn clauses; L means that λs occur; S means sets.

Our data differs from that of Meng and Paulson in two respects: they consider 153 [4] and 285 [5] goals as opposed to our 1240, and their goals are hand-selected for the purpose of tuning SH, where extremely easy or extremely hard problems are unhelpful. In contrast, all goals arising in our 7 theories are part of our test data, including those proved by induction, of which there are 72. Thus we claim our data is representative for the average goals actually faced by Isabelle users.

We have run all experiments with the 3 ATPs SH currently supports, with their default SH options: E [9] (version 1.0 in auto mode), SPASS [14] (version 3.5 with SOS enabled), and Vampire [8] (version 9.0 in CASC mode). SOS, *set of support*, is a complete resolution strategy [16] but incomplete in the SPASS context. We confirmed that SPASS with SOS worked best for us.

Below we abbreviate the provers as **E**, **S** and **V**.

We conducted our tests on Dual Core Intel Xeon processors running at 3.06 GHz. The ATPs were run with different timeouts, and "timeout" refers to ATPs by default. In contrast, M's timeout was fixed at 30s (for the tests — normally M can run as long as the user likes). The reason for the 30s: During interactive proof development, proof text does not evolve linearly one step after the next, but whole regions are continuously modified (by the user) and rechecked (by the machine). In our experience, 30s is at the limit of what users are prepared to tolerate for rechecking (as opposed to finding) of a proof step.

4 Success Rates

Below we present the success rates (in percent) both for the ATPs and for M proof reconstruction runs — we refer to them as ATP-success and M-success. Remember that M-success is what counts for the Isabelle user, because only those proofs can be imported into Isabelle. We have run E, S and V on all 7 sample theories, with ATP timeouts of 5, 10, 30, 60 and 120 seconds. In the following table we show the data obtained for 5s and 120s. The rightmost column, labeled with ∅, gives the average. The table contains for each prover, timeout and theory combination two values, the ATP and M success rates in percent. Both success rates are relative to the total number of problems (in each theory). Hence M-success is never above ATP-success.

	Arrow	NS	Hoare	Jinja	SN	FTA	FFT	∅
E 5	22 19	46 32	46 42	24 23	57 56	50 49	12 12	40 37
E 120	26 19	58 40	51 46	26 24	59 58	58 56	19 17	45 41
S 5	29 26	38 38	50 42	22 20	50 46	53 51	15 12	40 37
S 120	30 27	41 41	51 42	22 20	50 47	55 52	15 12	42 38
V 5	18 16	22 22	35 34	26 24	49 47	49 48	10 10	35 33
V 120	35 29	52 46	57 47	29 26	61 58	62 58	18 14	49 44
ESV 5	33 28	56 44	53 48	28 26	61 58	61 58	17 15	48 44
ESV 120	42 34	65 56	61 53	31 27	63 61	67 63	22 18	54 48

Percentages of problems solved

For example, E with a timeout of 120s can solve 26% of all goals in theory Arrow, but after running M on the solved goals (with the facts identified by the ATP), only 19% of all goals could actually be proved (because M failed to reconstruct $(26 - 19)/26 \approx 27\%$ of the E proofs). The two bottom rows of the table represent the ATP **ESV**, which runs E, S and V in parallel, and each of them is run until it finds a proof or reaches timeout.

For a subset of theories, the data is shown graphically for all timeouts in Figures 1 to 4, where the two success rates are labeled by P and PM, where $P \in \{\mathsf{E}, \mathsf{S}, \mathsf{V}\}$. The M-success rate of ESV is shown as the grey area. Individual ATP-success rates may lie above the grey area. Note that all graphs in this paper use a logarithmic timeline.

The splendid news is that on average

> Running each of the 3 provers for 5s yields the same M-success rate (44%) as running the most effective one for 120s.

This result is crucial for interactive proofs, where every second counts. The gain of 3 ATPs over 1 is impressive: the success rates rise between 4 and 13 percentage points, depending on the timeout (5s or 120s) and the ATP we compare ESV with. Having hard empirical evidence for the effectiveness of ESV has influenced Isabelle's SH setup: ESV is now the default setting. More precisely V is invoked *remotely* (see §2) because a) V is not readily available (for example not on MacOS) and b) it requires only a dual core machine and internet access to

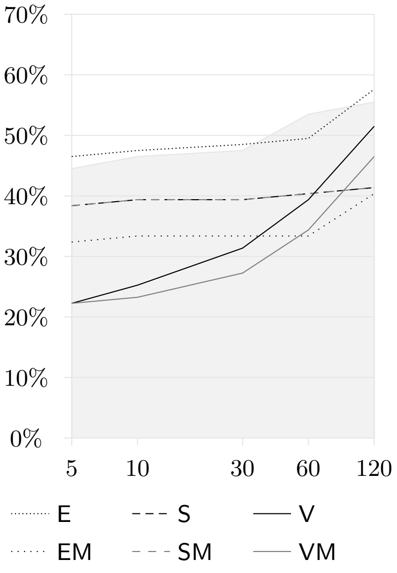


Fig. 1. Theory NS

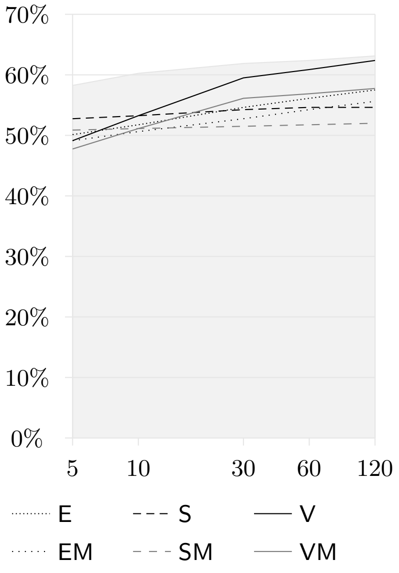


Fig. 2. Theory FTA

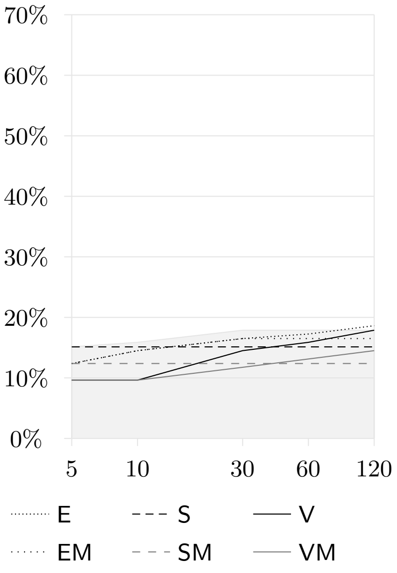


Fig. 3. Theory FFT

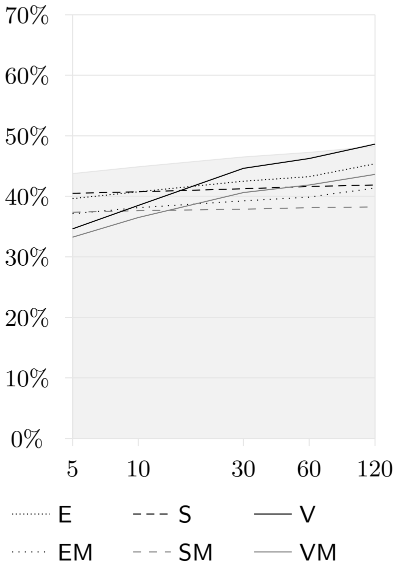
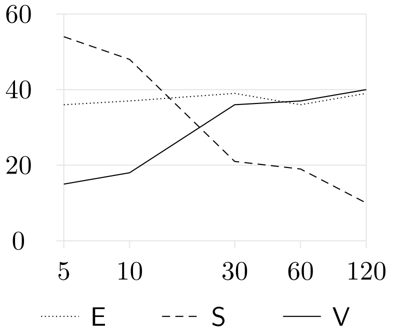


Fig. 4. Average

benefit from this setup without the provers stealing each others' cycles — thanks to Sutcliffe's SystemOnTPTP.

The Figures 1 to 4 bring out the difference between the provers' performance profiles very well. S starts out (at 5s) above E and V but does not improve much, whereas E and V keep on growing and overtake S at some point. The reason for V's behaviour is *strategy scheduling*: V runs multiple strategies in sequence until one finds a proof. Thus V is able to utilize long timeouts more effectively. Neither E nor S employ strategy scheduling. It is to E's credit that its success rate keeps growing almost as well as V's.

Success rates are one thing, indispensability is another. Indispensability of a prover can be judged by the number of goals only it can prove, its *uniqueness number* (an intuitive variant of Sutcliffe's more refined *SOTAC* [11]). In the diagram to the right we show the uniqueness numbers (cumulative over all theories) of our ATPs with varying timeout. Clearly, S is indispensable for users with no patience, V is indispensable for users with a lot of time, and nobody should be without E.



Further important observations concerning success rates are:

– The difference between the ATP and M success rates increases for E and V over time. This is not surprising because M's timeout is fixed. The problem is discussed in detail in the next subsection.
– In theory NS (Fig. 1), E tops all other provers, but EM is 10% below E. This is an exception and turns out to be the result of a typing problem (see §4.1).
– The theory with the lowest success rate is FFT. We conjecture (this is difficult to ascertain) that the culprit is λ: there are many goals that contain the summation operator $\sum$ which gives rise to a λ (internally). We looked at the goals that were proved and only one contained a $\sum$.
– Our average success rates are lower than Meng and Paulson's [4]. We believe that this is due to the fact that they hand-selected their goals whereas we picked entire theories with very variable success rates.

Readers disappointed by the actual ATP success rates should keep in mind that although all Isabelle goals are provable, not all ATP problems are: 72 problems require induction, and an unknown and hard to determine number is unprovable because the extraction of relevant facts from the Isabelle theory (step 1 of SH) does not guarantee to preserve provability.

We have also run the ATPs for 240s and observed the success rates: E/S/V prove an additional 9/0/10 goals, M proves an additional 5/0/5 goals — the success rates increase by a mere 0.4/0/0.4 percent. Hence we stopped at 120s.

We will now examine the problem that M may fail to reconstruct ATP proofs.

4.1 M May Fail

On average, the difference between ATP and M success rates is at most 5% (at 120s). This means that at 120s ATP timeout, M fails to reconstruct around 10% of all ATP proofs. The situation is extreme in theory NS, where M fails on 30% of the proofs found by E. There are three reasons why M may fail to reconstruct a proof:

1. The default translation of HOL problems to ATP input is unsound [4]. The reason is that HOL has a Haskell-like type system, which needs to be encoded into unsorted FOL, and by default SH economizes on the amount of type information that is passed to the ATPs for the sake of performance. In some cases this allows the ATPs to find proofs that are no longer sound when translated back to HOL. We call those **type-unsound** below.
2. M is internally a two stage process: the first stage is an ATP that produces proof objects, the second stage translates these proof objects into Isabelle proofs. The first stage is called with the same reduced type information passed to the external ATPs and may also find type-unsound proofs. In such cases the translation to Isabelle proofs throws a type exception.
3. M may time out.

It is hard to distinguish these 3 reasons. For example, if M throws a type exception, the proof found by the external ATP may be type-unsound, and M merely followed suit, or the external ATP may have found a perfectly good proof, but M's first stage found a type-unsound one.

For a timeout of 120s we examined the failed M calls individually and classified them according to the above three-fold distinction. The classification cannot be automatic, as we just explained. We used methods introduced below (e.g. adding full type information) but in some cases it remains approximate. Hence the figures below should be taken with a grain of salt. On average, 66% of failed M calls are genuine M timeouts, 21% are caused by type-unsound ATP proofs and 13% are type-unsound M proofs. But the individual provers differ notably in their profile of failed M calls:

120	Total # of M failures	M timeout	ATP type-unsound	M type-unsound
E	47	34%	51%	15%
S	42	79%	9%	12%
V	56	84%	5%	11%

This confirms V's ability to find difficult proofs, given enough time. And E appears to be the expert at exploiting (type-)unsound axiomatizations.

Of course it is not surprising that M cannot compete with highly optimized ATPs, the CASC already told us that. Hence the SH architecture is often viewed with suspicion by ATP researchers. In fact, we consider M in the SH context surprisingly effective. Of course, we would like to reconstruct all sound ATP proofs. There are two ways to improve the situation:

– *Give M more time.* Experimentally we doubled M timeout from 30s (see §3) to 60s. This reduced the number of M timeouts by 20%, but 60s is the limit of what is acceptable in an interactive system.

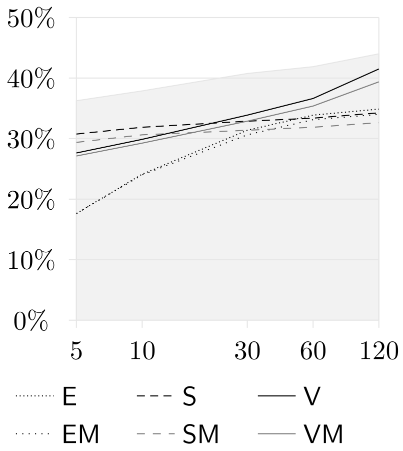
– *Replay the proof found by the ATP rather than reconstruct it.* Paulson [7] presents an extension of SH with a translation scheme from resolution proofs found by an ATP (and output in TSTP format [10]) into Isabelle proof scripts that replay the resolution proof step by step. At the time where that paper was written only E produced TSTP proofs, and the resulting Isabelle proof scripts were a bit brittle, i.e. they would sometimes fail because of technical problems. Last but not least, it is very unattractive to have long resolution proofs in the middle of nicely structured and readable proofs that are customary in Isabelle [15]. For these reasons the default setup for SH is to call M, and translation of TSTP proofs is currently not used at all, although it is clearly the way to go — but see the Conclusions.

When running ESV, M failures are greatly reduced: there are 121 problems where M fails to reconstruct at least one of the ATP proofs, but for 46 of them M succeeds to reconstruct a proof found by a different ATP.

We will now consider how type-unsound ATP and M proofs can be avoided.

4.2 The Fully-Typed Translation

To avoid type-unsound ATP proofs, SH also offers a fully-typed translation from HOL to FOL (**FT** below). The figure to the right shows the resulting average success rates. Compared with Fig. 4, we observe a decrease of the success rate of around 10% by going to FT. Meng and Paulson [4] measured 10–20%, but even 10% is not acceptable. Hence FT remains only an option. It can be useful in case an ATP appears to have found a type-unsound proof. Switching to FT can sometimes allow the ATP to find a valid proof. In most cases, however, the ATP will time out instead, which leaves one none the wiser. The purpose of FT, to avoid M failures, is largely fulfilled: the M-success rates are barely below those for the ATPs. They are not identical, because M may still genuinely time out (or find type-unsound proofs, see §4.3). With increasing ATP timeout, the gap between V and VM is widening because V starts to outperform M again (whose timeout is fixed, see §3).



4.3 Metis with Full Types

M is run by default with the same reduced type information as the ATPs. Thus M also finds type-unsound proofs, which are rejected by Isabelle. Therefore Paulson recently added a version of M with full type information, $\mathbf{M_{FT}}$ below. M_{FT} cannot replace M just like the fully-typed ATP translation cannot replace the default one, because of performance reasons. Instead we have evaluated how often $\mathsf{M_{FT}}$ would reconstruct a proof where M failed. That is, how much benefit we can expect from $\mathsf{M_{FT}}$ in addition to M. All figures below were obtained for 120s ATP timeout and refer to all failures over all theories and provers. The figures for M are given in §4.1 above.

Of the 18 valid ATP proofs where M runs into typing problems, only 6 can be reconstructed by $\mathsf{M_{FT}}$. In most of the other cases $\mathsf{M_{FT}}$ times out (because the type information overwhelms it), but in a few cases $\mathsf{M_{FT}}$ itself runs into typing problems (a somewhat unfortunate situation). Interestingly, $\mathsf{M_{FT}}$ also manages to reconstruct 5 (of 96) proofs where M had timed out. Taken together, $\mathsf{M_{FT}}$ succeeds on 9% of the M failures. These figures are cumulative over all 3 ATPs.

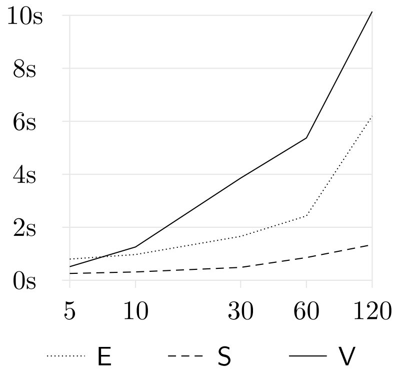
When running ESV, type unsoundness of M almost ceases to be an issue: only two such failed proofs remain. $\mathsf{M_{FT}}$ fails on both of them, too.

5 Time

SH performs the following steps: extracting and translating relevant facts, running the ATPs, and running M. Extraction and translation take 3.2s on average.

Below we show the run times for successful ATP runs in seconds. We exclude the failed runs because they generally end in timeout, thus distorting the statistics. The table on the left is restricted to 30s, 60s and 120s. The average over all theories is shown graphically on the right.

	Arrow	NS	Hoare	Jinja	SN	FTA	FFT
E 30	3.6	2.1	1.0	1.0	0.7	1.9	3.7
E 60	5.0	2.5	1.0	1.0	1.4	3.0	5.0
E 120	5.2	15.3	8.4	1.0	2.4	4.8	10.0
S 30	0.4	0.9	0.4	0.1	0.3	0.6	0.6
S 60	0.4	1.9	0.9	0.1	0.3	1.0	0.6
S 120	0.4	3.9	1.9	2.5	0.3	0.9	0.6
V 30	6.9	5.8	5.7	1.9	2.4	2.9	7.0
V 60	7.8	15.7	7.0	1.9	2.4	3.8	10.0
V 120	18.3	35.0	10.5	2.0	2.5	6.5	20.5



– We can see once again the effect of V's strategy scheduling. S is the most economical with its time.
– E and V agree on what are hard theories and what are easy ones.

For a theoretical analysis of $T(t)$, the total time for successful ATP runs with timeout t, observe in Fig. 4 that the average success rate is roughly linear w.r.t. a logarithmic t. This means that when the timeout increases from t to $2t$, a fixed number k of new goals are proved, on average in time $1.5t$: $T(2t) = T(t) + 1.5kt$. The master theorem for recurrence relations tells us that $T(t)$ is linear; the solution is $T(t) = 1.5kt + c_0$, for some c_0. Since the t axis is logarithmic, the expected graph for $T(t)$ would be an exponential function. Our data supports this merely vaguely.

We have also measured the average time spent by the ATPs on failed proof attempts. Since E and V try so hard, this figure is close to timeout. In some of the theories, S departs from this pattern and its average failure time is 30% below timeout. This could be induced by SOS's incompleteness (see §3).

The average run times for successful runs of M turn out to be moderate, that is, below 1 second, even at 120s ATP timeout:

120	Arrow	NS	Hoare	Jinja	SN	FTA	FFT	∅
E	0.0	0.5	0.3	0.5	0.0	0.0	0.2	0.2
S	0.3	0.1	0.2	0.6	0.0	0.1	0.5	0.2
V	0.3	0.1	0.4	0.3	0.0	0.1	0.1	0.2

This is perfectly acceptable for interactive use.

6 Proof Complexity

How difficult are the proofs found by the ATPs? We will look at it both from the ATP's and the user's point of view. From the ATP's point of view, the time taken to find a proof is one measure already covered. Another one is the number of facts used in the proof (i.e. the cardinality of R, see §2). This is a very crude measure as one can easily have long and difficult to find proofs with only a few facts. But we will observe a strong correlation between fact and time complexity. Below we show the average number of facts for the ATP timeouts of 5s and 120s.

For each prover and theory we show a pair i j where i is the average number of facts returned from ATP proofs and j the average number of facts in (successful!) M proofs. We have $i \geq j$ in most cases because M tends to fail on proofs involving many rather than a few facts. All figures are rounded.

5	Arrow	NS	Hoare	Jinja	SN	FTA	FFT	∅σ
E	3 2	6 5	2 2	3 2	2 2	3 3	5 5	3 3 3
S	2 2	3 3	2 2	2 1	1 1	2 2	6 7	2 2 3
V	2 3	3 3	2 2	2 2	2 2	3 3	5 5	3 3 3

120	Arrow	NS	Hoare	Jinja	SN	FTA	FFT	∅σ
E	9 2	6 5	3 2	3 2	2 2	4 4	6 5	4 3 4
S	2 2	3 3	3 2	2 1	1 1	3 2	6 7	3 2 3
V	3 3	6 5	4 3	3 2	2 2	4 4	7 6	4 3 4

The rightmost column contains triples i j s where i and j are the averages and s is the standard deviation of the j's. We observe the following:

- The average number of facts in M proofs is the same at 5s and 120s, namely 2–3. The standard deviation σ of 3–4, however, indicates that there is quite a bit of variation.
- Raising the timeout from 5s to 120s, the average number of facts in ATP proofs rises from 2–3 to 3–4, just 1 above the fact complexity of M proofs. This confirms the expectation that M failures lose the more difficult proofs, which are of course the ones users would most like to see automated.

– There is a strong correlation between the average fact complexity and run times of ATP proofs: proofs in theories SN and Jinja are particularly short and fast, in theories FFT and NS proofs are particularly long and slow.

Just like success rates, when the timeout is increased from 5s to 120s, fact complexity of proofs increases only slowly. However, the high standard deviation tells another story, which is confirmed when we look at the *maximum* fact complexity. In the table below, pairs $i\ j$ are the maximum fact complexities of all ATP and all M proofs in a given theory, and *max* is the maximum of the maxima.

5	Arrow	NS	Hoare	Jinja	SN	FTA	FFT	*max*
E	7 7	15 13	12 8	10 5	6 6	15 15	12 12	15 15
S	5 5	6 6	11 10	10 6	8 3	25 25	21 21	25 25
V	7 7	8 8	9 9	7 7	8 8	23 23	10 10	23 23
120	Arrow	NS	Hoare	Jinja	SN	FTA	FFT	*max*
E	47 7	15 15	18 14	10 5	7 7	33 33	37 12	47 33
S	5 5	7 7	24 10	10 6	8 5	28 25	21 21	28 25
V	10 7	22 15	52 35	8 7	8 8	44 44	21 18	52 44

From the *max* column we can tell that at 5s, M keeps up with the ATPs, but that at 120s, E and V outperform M significantly. The rest of the table merely illustrates the considerable variation depending on theories and provers.

We now look at proof complexity from a user perspective. For a user, a proof is **trivial** if it consists of the invocation of one of the proof methods `simp`, `auto`, `arith` or `blast`. They are the standard proof tools familiar to all Isabelle users. Their functionality is irrelevant here. The point is that automating such proofs is of little help to Isabelle users, because they are already automatic. Or at least almost so, because the form of the goal will almost always narrow the choice down to one or two of the methods. Note that we no longer consider `simp` and friends trivial if they need to be augmented somehow, e.g. by supplying additional facts, because in that case the user had to figure out what those facts are. It turns out that the percentage of trivial proofs among those found by the ATPs does not vary much with timeout and is around 64% at 30s timeout. In contrast, only 53% of all proofs in the considered theories are trivial. Clearly, ATPs have a predilection for trivial goals, i.e. goals with a trivial proof. *A priori*, this is not at all evident because triviality is in the eye of the beholder! For example, `simp` can generate long chains of conditional rewrites and `arith` complicated proofs in linear arithmetic. But this means that the success rates we measured so far are skewed. What the user really wants to know is this:

How many non-trivial goals can the ATPs prove for me?

Let G be the set of goals, $T \subseteq G$ the trivial ones (for Isabelle) and $A \subseteq G$ the automatically provable ones (by an ATP). Let $t = |T|/|G|$, $t_a = |A \cap T|/|A|$ and $s = |A|/|G|$ (the success rate). Then the *non-trivial success rate* $|A - T|/|G - T|$ turns out to be $s(1 - t_a)/(1 - t)$ by a simple calculation. In our setting $t = 0.53$ and $t_a = 0.64$ on average (see the two tables above). Thus we need to adjust the success rate by 0.77 to obtain the non-trivial success rate. Since s is around 45%

(M-success, see Fig. 4), it means that in fact only around 34% of all non-trivial goals are proved by SH.

Finally, we focus on the textually most complex Isabelle proofs, the compound ones. Those are subproofs consisting of a begin–end (`proof–qed` in Isabelle parlance) block. They are the last resort for most users, if everything else fails. There are 49 compound proofs in our theories (excluding inductions), of which E/S/V solved 8/8/3 with 5s and 9/8/9 with 120s timeout, roughly 17%, which is not bad.

7 Minimization

A set of facts returned by an ATP as a proof of some goal is **redundant** iff some proper subset of the facts proves the goal. Many proofs found by ATPs are redundant, for two very different reasons: certain facts are genuinely useless and can lead to unnecessary detours (if used), but other facts enable shortcuts and their removal forces a detour. Since genuinely useless facts can lead to M timeouts, we have investigated **minimization**, the process of reducing a given set of facts to an irredundant subset that still proves the goal. This is a new extension of SH that we developed at TUM. The implementation is due to Philipp Meyer.

7.1 Minimization Algorithms

The obvious linear algorithm removes one fact after another from the initial set, calls the ATP with the reduced fact set, and puts the fact back if that proof fails now. This requires as many calls of the ATP as there are facts in the initial set.[1] But there is also a clever algorithm based on bisecting the set [1, §4.3]. It can take as little as $\log_2 n$ calls of the ATP (if only 1 fact is needed) and as much as $2n$ (if all are needed). The beauty and lure of the binary algorithm was such that we implemented it right away. It was only after using it for some time that we suspected that it performed worse than the linear one on our data. After measuring the number of iterations of the binary algorithm, it turned out we were right: on average it required 1.15 times as many iterations as the linear one. This agrees with a simulation by Jasmin Blanchette of the binary algorithm on random data with the same redundancy rate as our data (about 1/3, see §7.2). Of course we switched to the linear algorithm as a result. The simulation predicts superiority of the binary algorithm above 40% redundancy.

7.2 Benefits

The following table shows how many additional goals could be proved due to minimization: each entry is the difference between the number of M failures before and after minimization. Negative numbers indicate a loss of proofs.

[1] Since the initial set is quite small (see §6), a default ATP timeout of 5s during minimization suffices most of the time and leads to very acceptable run times, in particular because minimization only needs to be performed once.

120	Arrow	NS	Hoare	Jinja	SN	FTA	FFT	$\sum$
E	3	1	4	1	0	−2	0	7
S	2	0	6	1	0	1	2	12
V	5	2	4	1	0	−3	1	10

Most of the time we gain proofs, but in FTA we lose a few. Losses are of no consequence: minimization is optional, and if M succeeds on the original set of facts but fails on the minimized set, the user would simply stick with the original set and no harm is done. Hence the negative numbers should be disregarded. In a nutshell: 9–13 (out of about 50, see the table in §4.1) M failures can be avoided by minimization. In FTA the net effect is negative because FTA is very algebraic: there is a large redundant set of facts that often allow short proofs; the removal of some such derived shortcut may just push M over the edge.

How redundant are proofs found by the ATPs? How much does minimization reduce the set of facts used? Depending on the theory, 10–50% of the facts can be dropped (30% on average). This does not vary much with ATP timeout, but increases for large proofs. For example, the record 52-fact proof found by V in Hoare (see §6) collapses to 3 facts after minimization.

What is the impact of minimization on M run times? We have measured by how much the run time changes on average, i.e. the average of all a_i/b_i where a_i (b_i) is the time M takes after (before) minimization of proof i, provided both runs succeed. However, run times for the same call of M easily fluctuate by 10ms. In particular, if a_i or b_i is below 10ms, a_i/b_i becomes somewhat random. Hence we have replaced a_i/b_i by 1 if $a_i - b_i \leq$ 10ms. The following table contains the averages of all a_i/b_i in each theory and the weighted average of the averages:

120	Arrow	NS	Hoare	Jinja	SN	FTA	FFT	∅
E	0.9	0.9	1.2	1.0	1.0	0.9	0.7	1.0
S	1.1	1.0	0.9	1.0	1.0	0.9	1.0	1.0
V	0.8	0.7	0.9	1.0	1.0	1.0	0.8	0.9

Minimization can cut both ways but has only a small effect on average.

Finally we look at the impact of minimization on M_{FT} (see §4.3). It turns out that because proofs are often simplified by minimization, M fails less often and M_{FT} is less in demand. If M_{FT} is called where M failed (with minimized sets of facts), a mere additional 4 proofs over all theories and provers is obtained. This is down from 11 additional proofs without minimization (§4.3)

8 Conclusions

With respect to our realistic test data we have established the following:

– Success rates for Sledgehammer are around 45% but vary enormously from theory to theory (from below 20% up to 60%).
– The more meaningful rate of how many non-trivial goals (by Isabelle standards) are solved by the ATPs is around 34%. *ATPs can help ITPs!*

- Running all 3 ATPs together for 5s yields the same success rate (44%) as running the most effective one for 120s. Therefore Sledgehammer now calls all 3 ATPs concurrently.
- SPASS is indispensable for short timeouts, Vampire for long ones, and E in any situation, according to the number of goals proved only by that ATP.
- CASC results for the FOF and CNF divisions (Vampire just ahead of E) correctly predict ATP success on Isabelle theories, although CASC test data is not dominated by Isabelle problems: at CASC-22, none of the FOF problems and 59 of the 200 CNF problems came from Isabelle theories.
- Proof reconstruction in Isabelle using Metis works well most of the time but loses up to 10% of sound ATP proofs, mainly because Metis times out.
- Minimization of ATP proofs reduces the required number of facts by 1/3 (and in an extreme case from 52 down to 3 facts), thus helping 20% of the failed Metis proofs to succeed. Our measurements showed that the naive algorithm was faster than the clever binary one we had implemented first.

For these reasons we plan the following future work items:

- In order to avoid loosing up to 10% of ATP proofs because Metis fails, we intend to reactivate proof replay [7] while ironing out the implementation and presentation problems. We aim at producing truly readable (natural deduction) proofs along the lines of Huang [2].
- We plan an “auto Sledgehammer” mode for Isabelle were each goal is automatically passed to all 3 ATPs with a low timeout like 5s. The low timeout avoids the current effect of users going into sleep mode and waiting for the default 60s timeout of all ATPs before they reactivate their brain.
- Our test harness will also be used for further tuning of Sledgehammer (the default filter parameters determined by Meng and Paulson [4]) and Metis (whose parameters have never been tuned for Isabelle). It will also help in continuous performance monitoring by gathering key figures like success rates in our daily regression tests.

Acknowledgement. Mike Gordon and Larry Paulson generously hosted a visit by Tobias Nipkow to the Cambridge Computer Lab where much of this research was conducted. Geoff Sutcliffe answered numerous questions. Jasmin Blanchette, Alex Krauss and Geoff Sutcliffe helped to improve the paper considerably.

References

1. Bradley, A., Manna, Z.: Property-directed incremental invariant generation. Formal Asp. Comput. 20, 379–405 (2008)
2. Huang, X.: Reconstructing proofs at the assertion level. In: Bundy, A. (ed.) CADE 1994. LNCS, vol. 814, pp. 738–752. Springer, Heidelberg (1994)
3. Hurd, J.: First-order proof tactics in higher-order logic theorem provers. In: Archer, M., Di Vito, B., Muñoz, C. (eds.) Design and Application of Strategies/Tactics in Higher Order Logics. Number NASA/CP-2003-212448 in NASA Technical Reports, pp. 56–68 (2003)

4. Meng, J., Paulson, L.C.: Translating higher-order clauses to first-order clauses. J. Automated Reasoning 40, 35–60 (2008)
5. Meng, J., Paulson, L.C.: Lightweight relevance filtering for machine-generated resolution problems. J. Applied Logic 7, 41–57 (2009)
6. Nipkow, T., Paulson, L., Wenzel, M.: Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002), `http://www.in.tum.de/~nipkow/LNCS2283/`
7. Paulson, L.C., Susanto, K.W.: Source-level proof reconstruction for interactive theorem proving. In: Schneider, K., Brandt, J. (eds.) TPHOLs 2007. LNCS, vol. 4732, pp. 232–245. Springer, Heidelberg (2007)
8. Riazanov, A., Voronkov, A.: The design and implementation of VAMPIRE. AI Commun. 15, 91–110 (2002)
9. Schulz, S.: E - a brainiac theorem prover. AI Commun. 15(2-3), 111–126 (2002)
10. Sutcliffe, G., Zimmer, J., Schulz, S.: TSTP Data-Exchange Formats for Automated Theorem Proving Tools. In: Sorge, V., Zhang, W. (eds.) Distributed Constraint Problem Solving and Reasoning in Multi-Agent Systems, pp. 201–215. IOS Press, Amsterdam (2004)
11. Sutcliffe, G.: The 4th IJCAR Automated Theorem Proving System Competition — CASC-J4. AI Commun. 22, 59–72 (2009)
12. Sutcliffe, G.: SystemOnTPTP. In: McAllester, D. (ed.) CADE 2000. LNCS, vol. 1831, pp. 406–410. Springer, Heidelberg (2000)
13. Urban, J.: MPTP 0.2: Design, implementation, and initial experiments. J. Automated Reasoning 37(1-2), 21–43 (2006)
14. Weidenbach, C., Dimova, D., Fietzke, A., Kumar, R., Suda, M., Wischnewski, P.: Spass version 3.5. In: Schmidt, R.A. (ed.) CADE-22. LNCS, vol. 5663, pp. 140–145. Springer, Heidelberg (2009)
15. Wenzel, M.: Isar — a generic interpretative approach to readable formal proof documents. In: Bertot, Y., Dowek, G., Hirschowitz, A., Paulin, C., Thery, L. (eds.) TPHOLs 1999. LNCS, vol. 1690, pp. 167–183. Springer, Heidelberg (1999)
16. Wos, L., Robinson, G., Carson, D.: Efficiency and completeness of the set of support strategy in theorem proving. J. ACM 12, 536–541 (1965)

Logic between Expressivity and Complexity

Johan van Benthem

Institute for Logic, Language and Computation
University of Amsterdam
`http://staff.science.uva.nl/~johan`

Abstract. Automated deduction is not just application or implementation of logical systems. The field of computational logic also poses deep challenges to our understanding of logic itself. I will discuss some key issues. This text is just an appetizer that will be elaborated in the lecture.

1 Logic and the Balance of Expressive Power and Computational Complexity

Defining and proving/computing are the main faces of logic. But they require a balance. Historically, first-order logic arose from type theory by giving up expressive power in order to gain axiomatizability (and better semantic transfer properties between models). The same move occurred a bit later in going from first-order logic to modal languages: one gives up yet more expressive power, but now one gains decidability (as well as discovering a new nice structural invariance for the modal language: viz. bisimulation).

2 Upward from Modal to Guarded Fragments

What makes the modal move to weaker languages tick? Did we go too fast? Essentially, standard modal operators are local guarded quantifiers of the special first-order form

$$\exists y(G(x,y) \ \& \ \varphi(x,y)),$$

where G is an atomic guard predicate, and the x, y are finite tuples of variables. Restricting quantifiers to only these forms defines the *Guarded Fragment* (GF).

Theorem 1. *GF is* decidable*, with an effective finite model property.*

Up to logical equivalence, GF is also the set of first-order formulas that are invariant for guarded bisimulation, a structural invariance that lies in between bisimulation and (potential) isomorphism.

But the border with complex behaviour lies still a bit higher up inside first-order logic. Decidability continues to hold for the 'Loosely Guarded Fragment' that allows conjunctions of guard atoms $\&G$:

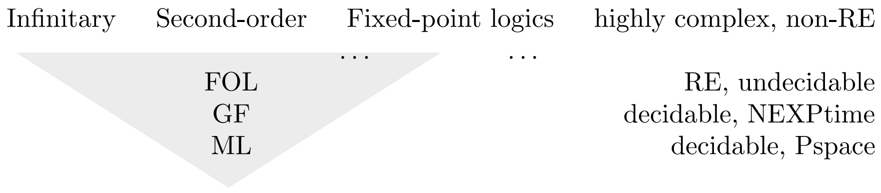
$$\exists y(\&G(x,y) \ \& \ \varphi(x,y)),$$

where any two variables in x, y occur under at least one atom in $\&G$.

Beyond this lie the 'cliffs of complexity': quantifiers expressing well-known confluence (grid) properties are not loosely guarded, think of

$$\forall yz((R(x,y) \;\&\; R(x,z)) \rightarrow \exists u(R(y,u) \;\&\; R(z,u))).$$

These can encode Tiling Problems, and so their logic becomes undecidable.



Aside (restricting a language versus re-interpretation): modal and related moves in logic have two faces. We either restrict to *fragments*, or we interpret all of FOL in some suitable *generalized semantics*, where not all assignments of objects to variables are available, encoding 'dependencies' in the model.

3 Aside: Downward to 'Poor Man's Logics'

Modal logics tend to be *Pspace*-complete. But this is not rock bottom yet. Going down even further to *feasible logics* with *(N)Ptime* satisfiability problems often takes non-Boolean languages.

Open Problem. Find a principled logical analysis for this move.

4 Model Theory in the Small: Lindström Theory

Our style of analysis multiplies logics. So, how can we understand the landscape of possible logical systems in greater generality?

Theorem 2 (Lindström). *FOL is inclusion-maximal with respect to the Compactness and Löwenheim-Skolem properties. For the latter one can also choose: the Karp Property (invariance of all sentences for potential isomorphisms), the RE property (axiomatizability of the valid sentences).*

Traditionally, foundational attention has only been paid to extensions of first-order logic (second-order, infinitary logics). Many characterizations exist, mostly via model-theoretic properties. But what happens if we look down in the landscape *below* FOL, on the idea that 'Small is Beautiful'?

Proof methods in Lindström theory require explicit encoding for back-and-forth properties of partial isomorphisms that capture first-order expressive power. But these are typically non-guarded grid properties. Still, new methods have been developed that work for small languages:

Theorem 3. *ML is maximal with respect to the properties of Compactness and Invariance for Bisimulation.*

Surprisingly, this yields a classical result in the theory of process logics:

Corollary 1. *A first-order formula is definable by a modal formula iff it is invariant for bisimulation.*

Now we can start a general abstract model theory of fragments, with new sorts of result. E.g., FOL is the largest extension of the *3-variable fragment* L_3 with the Compactness and Löwenheim-Skolem properties. (The crux is that 3 variables suffice for the encoding needed in a standard Lindström proof.)

Open Problem. Find a Lindström Theorem for GF.

For a related open problem for modal logic with a universal modality, there is a partial solution by Otto & Piro, but no best result yet [2].

5 Challenge 1: Fixed Point Logics

Logics with fixed-point operators have proved resistant to model-theoretic analysis ever sine the 1970s. (So far, the only things known about LFP(FO) are Karp and strong Löwenheim-Skolem properties.) And yet they are very natural as general logics of induction and recursion.

But modal lightweight logics can carry non-first-order fixed-point structure. Famously, the modal μ-calculus is decidable. And so is the *guarded fixed-point logic* LFP(GF). These logics present a challenge in terms of characterization, since they even improve on standard logics in having uniform interpolation, effective proofs of preservation theorems, etc.

Open Problem. Find a Lindström-type analysis for fixed-point logics (whether first-order or modal).

Perhaps, we have hit a boundary here of the usual model-theoretic stance in logic. We might essentially need further computational properties such as the Effective Finite Model Property, or even more explicitly procedural properties of logics, say, from Automata Theory.

This fits with a general issue in understanding logical systems today. Should we think of them as consisting essentially of *definition plus procedure*? This fits with the current trend to *logic games* that cast basic logical notions (truth, consistency, proof) in terms of interactive procedures. But we have no abstract game theory yet to back this up.

6 Challenge 2: Logic Combinations

As we go down in the landscape, logics get simpler. But in applications to agent theory (and cognitive science) we need to put the simple pieces back together again. Then *logic combinations* become ubiquitous. Can we do this while keeping trees and avoiding grids? Warning example: the *mode of combination* may crucially affect the outcome:

Theorem 4. *Putting together simple modal logics of time/action and knowledge for agents with* Perfect Memory *gives commuting diagrams, and hence the Recurrent Tiling Problem can be encoded, making the logic Π^1_1-complete.*

Similar surprises may be in store in the upcoming logical study of games:

Open Problem. What is the complexity for modal logics of action and preference in games for players that satisfy the usual assumption of *Rationality*?

Issue for reflection: What do these results mean? How bad is high complexity for a logic describing agents? Better focus on *complexity of agent tasks*?

7 Aside: Let the Structure Help

There is another traditional source of low complexity in logics: through wealth rather than poverty. Rich structures can also create decidability, even for dangerous languages with grid patterns:

Theorem 5. *Tarski's elementary geometry is decidable.*

The reason is that Euclidean space allows for 'elimination of quantifiers' in arbitrary first-order sentences. It is not so clear, however, whether this line is very helpful to us in computational logic.

8 Discussion: Practical Perspectives on Expressiveness and Complexity

Finally, if time permits, we will discuss a few outrageously general perspectives on the above issues. *Philosophy*: low complexity may not be needed, since a key aspect of rationality is a talent for exercising 'judgment' in using a potentially dangerous tool. *Cognitive science*: we operate in a complex world by learning where our best talents lie, and then selecting the right inputs. In line with these ideas, there are now new richer views of a reasoning system as merging two crucial abilities: logical inference plus memory search for pattern recognition, combining logic with probabilistic features.

Open Problem. Develop a general model theory for combined logics with probabilistic components.

Interestingly, in this area, the probabilistic Zero-One laws for first-order logic were discovered around the same time as Lindström's Theorem, but no similar theory has emerged yet.

But in this setting, we may also want to rethink our traditional view of 'logic and cognition'. Computational logic designs new forms of behaviour that get *inserted* into existing cognitive practice (just as mathematics has done in history). The new challenge may be understanding this insertion and its 'hybrids' of natural and designed behaviour.

Finally, even without resolving all this in depth, there is the practical world of *teaching*: introducing automated logic and a sense of the above considerations into general logic teaching seems rewarding (see the Opencourse project `http://staff.science.uva.nl/~jaspars/OpenCourse/`)

References

1. Andréka, H., Németi, I., van Benthem, J.: Modal logics and bounded fragments of predicate logic. Journal of Philosophical Logic 27(3), 217–274 (1998)
2. Otto, M., Piro, R.: A Lindström characterisation of the guarded fragment and of modal logic with a global modality. In: Areces, C., Goldblatt, R. (eds.) Advances in Modal Logic 7, papers from the seventh conference on 'Advances in Modal Logic,' Nancy, France, pp. 273–287. College Publications (2008)
3. ten Cate, B., van Benthem, J., Väänänen, J.A.: Lindström theorems for fragments of first-order logic. In: LICS, pp. 280–292 (2007)
4. van Benthem, J.: Exploring Logical Dynamics. Center for the Study of Language and Information, Stanford, CA, USA (1997)
5. van Benthem, J.: Guards, bounds, and generalized semantics. Journal of Logic, Language and Information 14(3), 263–279 (2005)
6. van Benthem, J.: Minimal predicates, fixed-points, and definability. Journal of Symbolic Logic 70(3), 696–712 (2005)
7. van Benthem, J.: A new modal Lindström theorem. Logica Universalis 1(1), 125–138 (2007)
8. van Benthem, J.: Logical Dynamics of Information and Interaction. Cambridge University Press, Cambridge (2010)
9. van Benthem, J.: Modal Logics for Open Minds. Center for the Study of Language and Information, Stanford, CA, USA (2010)

Multi-Prover Verification of Floating-Point Programs[⋆]

Ali Ayad[1,3] and Claude Marché[2,3]

[1] CEA LIST, Software Safety Laboratory, F-91191 Gif-sur-Yvette
[2] INRIA Saclay - Île-de-France, F-91893 Orsay
[3] LRI, Univ. Paris-Sud, CNRS, F-91405 Orsay

Abstract. In the context of deductive program verification, supporting floating-point computations is tricky. We propose an expressive language to formally specify behavioral properties of such programs. We give a first-order axiomatization of floating-point operations which allows to reduce verification to checking the validity of logic formulas, in a suitable form for a large class of provers including SMT solvers and interactive proof assistants. Experiments using the Frama-C platform for static analysis of C code are presented.

1 Introduction

Floating-point (FP for short) computations appear frequently in critical applications where a high level of confidence is sought: aeronautics, space flight, energy (nuclear plants), automotive, etc. There are numerous approaches for checking that a program runs as expected: testing, assertion checking at runtime, model checking, abstract interpretation, etc. *Deductive* verification techniques, originating from the landmark approach of Floyd-Hoare logic, amounts to generating automatically logic formulas called *verification conditions* (VCs for short), using techniques such as Dijkstra's weakest precondition calculus, so that validity of VCs entails soundness of the code with respect to its specification. The generated VCs are checked valid by theorem provers, hopefully automatic ones. Complex behavioral properties of programs can be verified by deductive verification techniques, since these techniques usually come with expressive specification languages to specify the requirements. Nowadays, several implementations of deductive verification approaches exist for standard programming languages, e.g., ESC-Java2 [12] and KeY [6] for Java, Spec# [3] for C#, VCC [29] and Frama-C [18] for C. In each of them, *contracts* (made of preconditions, postconditions, and several other kinds of annotations) are inserted into the program source text with specific syntax, usually in a special form of comments that are ignored by compilers. The resulting annotation languages are called *Behavioral Interface Specification Languages* (BISL), e.g., JML [10] for Java, ACSL [5] for C.

To analyse accurary of FP computations, abstract interpretation-based techniques have shown quite successful on critical software. However, there are very few attempts

⋆ This work was supported by the French national projects: *CerPan* (*Certification of numerical programs*, ANR-05-BLAN-0281-04), *Hisseo* (*Static and dynamic analysis of floating-point programs*, Digiteo 09/2008-08/2011), and *U3CAT* (*Unification of Critical C Code Analysis Techniques*, ANR-09-ARPEGE).

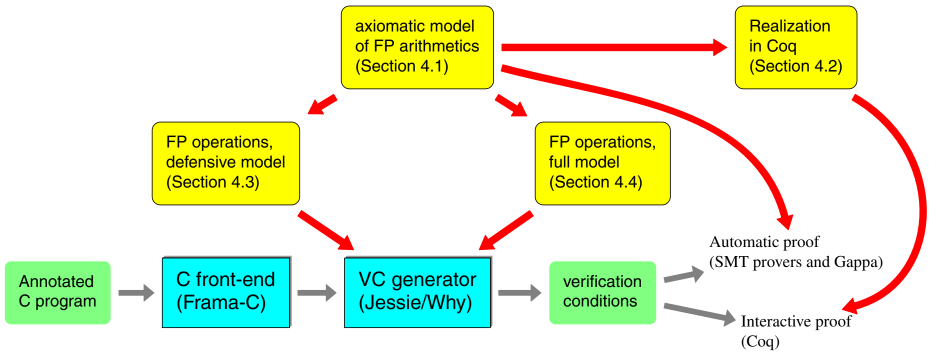

axiomatic model of FP arithmetics (Section 4.1)
Realization in Coq (Section 4.2)
FP operations, defensive model (Section 4.3)
FP operations, full model (Section 4.4)
Automatic proof (SMT provers and Gappa)
Annotated C program
C front-end (Frama-C)
VC generator (Jessie/Why)
verification conditions
Interactive proof (Coq)

Fig. 1. Architecture of our FP modeling

to provide ways to specify and to prove behavioral properties of FP programs in deductive verification systems like those above mentioned. This is difficult because FP computations are described operationally and have tricky behaviors as shown by Monniaux [25]. Consequently, it is hard to describe denotationally in a logic setting. A first proposal has been made in 2007 by Boldo and Filliâtre [7] for C code, using the Coq proof assistant [30] for discharging VCs. The approach presented in this paper is a follow-up of the Boldo-Filliâtre approach, which we extend in two main directions: first a full support of IEEE-754 standard for FP computations, including special floating-point values $\pm\infty$ and NaN (Not-a-Number); and second the use of *automatic* theorem provers. Our contributions are the following:

- Additional constructs to specification languages for specifying behavioral properties of FP computations. This is explained in Section 3.
- Modeling of FP computations by a first-order axiomatization, suitable for a large set of different theorem provers, and interpretation of annotated programs in this modeling (Section 4). There are two possible interpretations of FP operations in programs: a *defensive* version which forbids overflows and consequently apparition of special values (Section 4.3); and a *full* version which allows special values to occur (Section 4.4).
- Combination of several provers to discharge VCs (Section 5).

Our approach is implemented in the Frama-C [18] platform for static analysis of C code, and experiments performed with this platform are presented along this paper (see `http://hisseo.saclay.inria.fr/gallery.html` for other examples). The lower part of Fig. 1 represents the current state of Frama-C, and the upper part presents the additions we make for dealing with FP computations.

2 The IEEE-754 Standard for Floating-Point Arithmetic

The IEEE-754 standard [1] defines an expected behavior of FP computations. It describes binary and decimal formats to represent FP numbers, and specifies the elementary operations and the comparison operators on FP numbers. It explains when FP exceptions occur, and introduces special values to represent signed infinities and NaNs. We summarize here the essential parts we need, see [19] for more details. In this paper we focus on the 32-bits (type `float` in C, Java) and 64-bits (type `double`) binary formats; adaptation to other formats is straightforward. Generally speaking, in any of these formats, an interpretation of the bit sequence under the form of a sign, a mantissa and an exponent is given, so that the set of FP numbers denote a finite subset of real numbers, called the set of *representable* numbers in that format.

For each of the basic operations (add, sub, mul, div, and also sqrt, fused-multiply-add, etc.) the standard requires that it acts as if it first computes a true real number, and then *rounds* it to a number representable in the chosen format, according to some *rounding mode*. The standard defines five rounding modes: if a real number x lies between two consecutive representable FP numbers x_1 and x_2, then the rounding of x is as follows. With mode *Up* (resp. *Down*), it is x_2 (resp. x_1). With *ToZero*, it is x_1 if $x > 0$ and x_2 if $x < 0$. With *NearestAway* and *NearestEven*, it is the closest to x among x_1 and x_2, and if x is exactly the middle of $[x_1, x_2]$ then in the first case it is x_2 if $x > 0$ and x_1 if $x < 0$; whereas in the second case the one with even mantissa is chosen.

The standard defines three special values: $-\infty, +\infty$ and NaN. It also distinguishes between positive zero (+0) and negative zero (-0). These numbers should be treated both in the input and the output of the arithmetic operations as usual, e.g. $(+\infty) + (+\infty) = (+\infty)$, $(+\infty) + (-\infty) = \mathsf{NaN}$, $1/(-\infty) = -0$, $\pm 0/\pm 0 = \mathsf{NaN}$, etc.

IEEE-754 characterizes FP formats by describing their bit representation, but for formal reasoning on FP computations, it is better to consider a more abstract view of binary FP numbers: a FP number is a pair of integers (n, e), which denotes the real number $n \times 2^e$, where $n \in \mathbb{Z}$ is the *integer significand*, and $e \in \mathbb{Z}$ is the *exponent*. For example, in the 32-bit format, the real number 0.1 is approximated by[1] `0x1.99999Ap-4`, which can be denoted by the pair of integers $(13421773, -27)$. Notice that this representation is not unique, since, e.g, (n, e) and $(2n, e - 1)$ represent the same number. This set of pairs denote a superset of all FP numbers in any binary format. A suitable characterization of a given FP format f is provided by a triple $(p, e_{\min}, e_{\max})$ where p is a positive integer called the *precision* of f, and $e_{\min}$ and $e_{\max}$ are two integers which define a range of exponents for f. A number $x = n \times 2^e$ is *representable* in the format f (*f-representable* for short) if n and e satisfy $|n| < 2^p \quad \text{and} \quad e_{\min} \leq e \leq e_{\max}$. If x is representable, its *canonical representative* is the pair (n, e) satisfying the property above for $|n|$ maximal.[2] The characterization of the `float` format is $(24, -149, 104)$ and those of `double` is $(53, -1074, 971)$. The largest f-representable number is $(2^p - 1)2^{e_{\max}}$. In order to express whether an operation overflows or not, we introduce a notion of *unbounded representability* and *unbounded rounding*: A FP number $x = n \times 2^e$ is *unbounded f-representable* for format $f = (p, e_{\min,}, e_{\max})$ if $|n| < 2^p \quad \text{and} \quad e_{\min} \leq e$. The *unbounded f, m-rounding operation* for given format f and rounding mode m maps any real number x to the closest (according to m) unbounded f-representable number. We denote that as $\text{round}_{f,m}$.

[1] C99 notation for hexadecimal FP literals: `0x`*hh*`.`*hh*`p`*dd*, where h are hexadecimal digits and *dd* is in decimal, denotes number $hh.hh \times 2^{dd}$, e.g. `0x1.Fp-4` is $(1 + 15/16) \times 2^{-4}$.

[2] This definition allows a uniform treatment of *normalized* and *denormalized* numbers [1].

```
/*@ requires \abs(x) <= 1.0;
  @ ensures \abs(\result - \exp(x)) <= 0x1p-4; */
double my_exp(double x) {
  /*@ assert \abs(0.9890365552 + 1.130258690*x +
    @          0.5540440796*x*x - \exp(x)) <= 0x0.FFFFp-4; */
  return 0.9890365552 + 1.130258690 * x + 0.5540440796 * x * x;
}
```

Fig. 2. Remez approximation of the exponential function

3 Behavioral Specifications of Floating-Point Programs

We propose extensions to specification languages in order to specify properties of FP programs. As a basis we consider classical first-order logic with built-in equality and arithmetic on both integer and real numbers. We assume also built-in symbols for standard functions such as absolute value, exponential, trigonometric functions and such. Those are typically denoted with backslashes: `\abs`, `\exp`, etc. The core of the specification language is made of a classical BISL (ACSL [5] in our examples) which allows function contracts (preconditions, postconditions, frame clauses, etc.), code annotations (code assertions, loop invariants, etc.) and data invariants.

To deal with FP properties, we first make important design choices. First, there is no FP arithmetic in the annotations: operators $+$, $-$, $*$, $/$ denote operations on mathematical real numbers. Thus, there are neither rounding nor overflow that can occur in logic annotations. Second, in annotations any FP program variable, or more generally any C *left-value* of type `float` or `double`, denotes the real number it represents. The following example illustrates the impact of these choices.

Example 1. The C code of Fig. 2 is an implementation of the exponential function for double precision FP numbers in interval $[-1; 1]$, using a so-called *Remez polynomial approximation* of degree 2.

The contract declared above the function contains a precondition (keyword `requires`) which states that this function is to be called only for values of x with $|x| \leq 1$. The postcondition (keyword `ensures`) states that the returned value (`\result`) is close to the real exponential, the difference being not greater than 2^{-4}. The function body contains an `assert` clause, which specifies a property that holds at the corresponding program point. In that particular code, it states that the expression $0.9890365552 + 1.130258690x + 0.5540440796x^2 - \exp(x)$ *evaluated as a real number*, hence without any rounding, is not greater than $(1 - 2^{-16}) \times 2^{-4}$.

The intermediate assertion thus naturally specifies the *method error*, induced by the mathematical difference between the exponential function and the approximating polynomial; whereas the postcondition takes into account both the method error and the *rounding errors* added by FP computations.

So far we did not specify anything about the rounding mode in which programs are executed. In Java, or by default in C, the default rounding mode is NearestEven. In the C99 standard, there is a possibility for dynamically changing it using `fesetround()`. For efficiency issues, is not recommended to change it too often, so usually a program will run in a fixed rounding mode set once for all. To specify what is the expected

```
//@ pragma allowOverflow
//@ pragma roundingMode(Down)

typedef struct { double l, u; } interval;
/*@ type invariant is_interval(interval i) =
  @     (\is_finite(i.l) || \is_minus_infinity(i.l)) &&
  @     (\is_finite(i.u) || \is_plus_infinity(i.u)) ;     */

/*@ predicate double_le_real(double x,real y) =
  @     (\is_finite(x) && x <= y) || \is_minus_infinity(x);
  @ predicate real_le_double(real x,double y) =
  @     (\is_finite(y) && x <= y) || \is_plus_infinity(y);
  @ predicate in_interval(real x,interval i) =
  @     double_le_real(i.l,x) && real_le_double(x,i.u); */

/*@ ensures \forall real a,b;
  @     in_interval(a,x) && in_interval(b,y) ==>
  @          in_interval(a+b,\result);     */
interval add(interval x, interval y) {
  interval z;
  z.l = x.l + y.l; z.u = −(−x.u − y.u);
  return z;
}
```

Fig. 3. Interval structure, its invariant, and addition of intervals

rounding mode we choose to provide a special global declaration in the specification language: `pragma roundingMode(`*value*`)`; where *value* is either one of the 5 IEEE modes, or 'variable', meaning that it can vary during execution. The default is thus `pragma roundingMode(NearestEven)`. In the 'variable' case, a special *ghost* variable is available in annotations, to denote the current mode. Since the first case is the general one, we focus on it in this paper.

Usually, in a program involving FP computations, it is expected that special values for infinities and NaNs should never occur. For this reason we choose that by default, arithmetic overflow should be forbidden so that special values never occur. This first and default situation is called the *defensive* model: it amounts to check that no overflow occur for all FP operations. For programs where special values are indeed expected to appear, we provide another global declaration: `pragma allowOverflow`, to switch to the so-called *full* model. In that case, a set of additional predicates are provided: `\is_finite`, `\is_infinite`, `\is_NaN` are unary predicates to test whether an expression of type float or double is either finite, infinite or NaN. Additional shortcuts are provided, e.g. `\is_plus_infinity`, etc. (See [2] for details.)

Example 2. Interval arithmetic aims at computing lower bounds and upper bounds of real expressions. It is a typical example of a FP program that uses a specific rounding mode and makes use of infinite values.

An interval is a structure with two FP fields representing a lower and an upper bound. It represents the sets of all the real numbers between these bounds. Fig. 3 provides a C implementation of such a structure, equipped with a *data invariant* [5] which states that the lower bound might be $-\infty$ and the upper bound might be $+\infty$. The two pragmas specify that overflows are expected and the Down rounding mode is in use. In the same figure, a behavioral specification for addition is specified via a predicate `in_interval`(x, i) stating that a real x belongs to an interval i. Notice the trick for computing the upper bound in Down mode, using negations.

Notice that since we choose a standard logic with total functions, usual caution must be taken [11]: a formula should mention the real value of some FP expression x only in contexts where `\is_finite`(x) is known to hold, such as in the definition of predicate `double_le_real` of Fig. 3 (similarly as one should mention $1/x$ only when x is known to be non-null).

4 Modeling FP Computations

We model FP programs and their annotations, in order to reduce soundness to proper VCs. We proceed in four steps, the ones schematized on the upper part of Fig. 1.

4.1 Axiomatization of FP Arithmetics

To remain prover-independent, we model FP numbers with *abstract* datatypes Single and Double (the support for more formats would amount to add new types). For each f among single and double, we introduce an observation function: value_f : $f \to \mathbb{R}$, supposed to denote the real number represented by a FP number (when it is finite). The largest f-representable number is introduced in our modelling by constants max_f : $\mathbb{R}$ defined as max_single $= (2^{24} - 1) \times 2^{104}$ and max_double $= (2^{53} - 1) \times 2^{971}$.

The five IEEE rounding modes are naturally modelled by a concrete datatype mode $=$ Up $|$ Down $|$ ToZero $|$ NearestAway $|$ NearestEven. The function round$_{f,m}$ defined in Section 2 is introduced as an underspecified logic function round_f : mode, $\mathbb{R} \to \mathbb{R}$. Then, the following predicate indicates when the rounding *does not overflow*: no_overflow_$f(m$: mode, x : $\mathbb{R}) :=$ $|$round_$f(m, x)| \leq$ max_f. For example, computing $10^{200} \times 10^{200}$ in 64 bits overflows. In our model, it is represented by round_double(NearestEven, $10^{200} \times 10^{200}$) which is supposed to denote something close to 10^{400}. It exceeds max_Double, thus no_overflow_Double(NearestEven, $10^{200} \times 10^{200}$) is false.

The rounding function round_f is not directly defined: we axiomatize it by some, incomplete, set of axioms. Here are two of them, useful in the examples of this paper: $\forall m$: mode; $x, y : \mathbb{R}$,

$$|x| \leq \text{max\_}f \Rightarrow \text{no\_overflow\_}f(m, x) \quad (1)$$

$$x \leq y \Rightarrow \text{round\_}f(m, x) \leq \text{round\_}f(m, y) \quad (2)$$

In order to annotate FP programs that allow overflows and special values, we extend the above logical constructions with new types, predicates and functions. A natural idea would be to introduce new constants to represent NaN, $+\infty$, $-\infty$. We do not do that for two reasons: first, there are several NaNs, and second, we want to keep the Single and Double types as abstract, equipped with observation functions, and not a mixed abstract/concrete representation with constants. Our proposal is thus to add two new observation functions, similar to value_f, to give the *class* of a float, either *finite*, *infinite* or *NaN*; and its *sign*. We introduce two concrete types Float_class $=$ Finite $|$ Infinite $|$ NaN and Float_sign $=$ Negative $|$ Positive and additional functions class_f : $f \rightarrow$ Float_class and sign_f : $f \rightarrow$ Float_sign which indicate respectively the class and the sign of a FP number.

Additional predicates are defined to test if a FP number is finite, infinite, NaN, etc.: is_finite_$f(x : f)$:= class_$f(x)$ = Finite, and similar definitions for is_infinite_f, is_NaN_f, is_plus_infinity_f, is_minus_infinity_f, etc.

Comparison between two FP numbers is given by the predicates le_f, lt_f, eq_f, etc., e.g.

$$\begin{aligned}\mathsf{le\_}f(x:f,y:f) \;:=\; & (\mathsf{is\_finite\_}f(x) \;\wedge\; \mathsf{is\_finite\_}f(y) \;\wedge\; \mathsf{value\_}f(x) \le \mathsf{value\_}f(y))\\ & \vee\; (\mathsf{is\_minus\_infinity\_}f(x) \;\wedge\; \neg\, \mathsf{is\_NaN\_}f(y))\\ & \vee\; (\neg\, \mathsf{is\_NaN\_}f(x) \;\wedge\; \mathsf{is\_plus\_infinity\_}f(y))\end{aligned}$$

We must constrain our model to ensure that the sign function is consistent with the sign of real numbers: whenever x represents a finite number, sign_$f(x)$ should have the sign of value_$f(x)$. This is achieved by the following definitions

$$\begin{aligned}&\mathsf{same\_sign\_}f(x:f,y:f) \;:=\; \mathsf{sign\_}f(x) = \mathsf{sign\_}f(y)\\ &\mathsf{diff\_sign\_}f(x:f,y:f) \;:=\; \mathsf{sign\_}f(x) \neq \mathsf{sign\_}f(y)\\ &\mathsf{same\_sign\_real\_}f(x:\mathbb{R},y:f) \;:=\\ &\qquad (x < 0 \wedge \mathsf{sign\_}f(y) = \mathsf{Negative}) \vee (x > 0 \wedge \mathsf{sign\_}f(y) = \mathsf{Positive})\end{aligned}$$

and an axiom: $\forall x : f$,

$$(\mathsf{is\_finite\_}f(x) \wedge \mathsf{value\_}f(x) \neq 0) \Rightarrow \mathsf{same\_sign\_real\_}f(\mathsf{value\_}f(x), x) \qquad (3)$$

4.2 A Coq Realization of the Axiomatic Model

Our formalization of FP arithmetic is a first-order, axiomatic one. It is clearly under-specified and incomplete.

We realized this axiomatic model in the Coq proof assistant. This realization has two different goals. First, it allows us to prove the lemmas we added as axioms, thus providing an evidence that our axiomatization is consistent. Second, when dealing with a VC in Coq involving FP arithmetic, we can benefit from all the theorems proved in Coq about FP numbers. We build upon the Gappa [22] library which provides: (1) a definition of binary finite FP numbers: type float2 (a pair of integers as in section 2) together with a function float2R mapping (n, e) to the real $n \times 2^e$; (2) a complete definition of the rounding function. Our realization amounts to declare types format, mode, Float_class and Float_sign as inductive types, and defines max_f by cases. The abstract types Single and Double are realized by Coq records whose fields are:

– genf of type float2;
– The value_f field which is equal to (float2R genf);

- The class_f field, of type $\mathsf{Float\_class}$;
- The sign_f field, of type $\mathsf{Float\_sign}$;
- An *invariant* corresponding to axiom (3).

The last field is a noticeable point: it allows us to realize properly the $\mathsf{finite\_sign}$ axiom above. Finally, the round_f operator is realized by the corresponding one in the Gappa library.

4.3 The Defensive Model of FP Computations

To model the effect of the basic FP operations, we now need to make an important assumption: we assume that both the compiler and the processor implement *strict* IEEE-754, that is any single operation acts as if it first computes a true real number, and then rounds the result to the chosen format, according to the rounding mode. For example, addition of FP numbers is $\text{add}_{f,m}(x, y) = \text{round}_{f,m}(x + y)$ for x, y non-special values, where the $+$ on the right is the mathematical addition of real numbers. This means in particular that addition overflows whenever the rounding overflows. We will discuss this assumption in Section 6.

We model FP operations in FP programs by abstract functions, using the Hoare-style notation $f(x_1, \ldots, x_n) : \{P(x_1, .., x_n)\}\tau\{Q(x_1, .., x_n, \mathsf{result})\}$, which specifies that operation f expects arguments $x_1, \ldots, x_n$ satisfying P (this leads to a VC at each call site) and returns a value r (denoted by keyword result) of type τ, such that $Q(x_1, .., x_n, r)$ holds. In other words, in our modeling we do not say exactly how an operation is performed, but only give its specification.

The defensive model must ensure that no overflows and no NaNs should ever occur. This can be done by proper preconditions to operations. For instance, division of FP numbers is modeled by an abstract function

$$\begin{array}{l}\mathsf{div\_}f(m : \mathsf{mode}, x : f, y : f) : \\ \quad \{\ \mathsf{value\_}f(y) \neq 0\ \wedge \mathsf{no\_overflow\_}f(m, \mathsf{value\_}f(x)/\mathsf{value\_}f(y))\ \} \\ \quad f \\ \quad \{\ \mathsf{value\_}f(\mathsf{result}) = \mathsf{round\_}f(m, \mathsf{value\_}f(x)/\mathsf{value\_}f(y))\ \}\end{array}$$

This reads as: the computation of a FP division requires to check that the divisor is not zero, and the result of the division in $\mathbb{R}$ does not overflow, and it returns a FP number in format f whose real value is the rounding of the real result. Other operations such that addition. subtraction, unary negation and multiplication are defined similarly, and also cast operations between float formats. The square root function is defined similarly, requiring that the argument is non-negative.

Notice that, for a given operation in a program, the expected format of the result is known at compile-time, by static typing. But on the contrary, it should be clarified what is the rounding mode to choose: we use whatever is declared by the pragma `roundingMode` in Section 3.

Particular care has to be taken for FP constant literals: they are not necessarily representable and they are rounded (usually at compile-time) to a FP number according to a certain rounding direction (usually NearestEven). This is modeled by the following abstract function:

$$\begin{array}{l}\mathsf{real\_to\_}f(m:\mathsf{mode}, x:\mathbb{R}):\\ \quad \{\ \mathsf{no\_overflow\_}f(m,x)\ \}\ f\ \{\ \mathsf{value\_}f(\mathsf{result}) = \mathsf{round\_}f(m,x)\}\end{array}$$

This reads as: the real value of the literal must be able to be rounded without overflow, and then the result is its rounding.

4.4 The Full Model of FP Computations

The *full* model allows FP computations to overflow, and make use of special values: NaNs, infinities and signed zeros. Unlike for the defensive model, there are no preconditions on operations. We carefully interpret IEEE-754 informal specifications into postconditions taking all cases into account. Below is the complete specification for the multiplication (see [2] for other operations).

```
mul_$f$($m$ : mode, $x$ : $f$, $y$ : $f$) :
    { // no preconditions }
    $f$
    { ((is_NaN_$f(x) \vee$ is_NaN_$f(y)) \Rightarrow$ is_NaN_$f$(result))
            // NaNs arguments propagate to the result
    $\wedge$ ((is_zero_$f(x) \wedge$ is_infinite_$f(y)) \Rightarrow$ is_NaN_$f$(result))
    $\wedge$ ((is_infinite_$f(x) \wedge$ is_zero_$f(y)) \Rightarrow$ is_NaN_$f$(result))
            // zero times $\infty$ gives NaN
    $\wedge$ ((is_finite_$f(x) \wedge$ is_infinite_$f(y) \wedge$ value_$f(x) \neq 0) \Rightarrow$ is_infinite_$f$(result))
    $\wedge$ ((is_infinite_$f(x) \wedge$ is_finite_$f(y) \wedge$ value_$f(y) \neq 0) \Rightarrow$ is_infinite_$f$(result))
            // $\infty$ times non-zero finite gives $\infty$
    $\wedge$ ((is_infinite_$f(x) \wedge$ is_infinite_$f(y)) \Rightarrow$ is_infinite_$f$(result))
            // $\infty$ times $\infty$ gives $\infty$
    $\wedge$ ((is_finite_$f(x) \wedge$ is_finite_$f(y) \Rightarrow$
        $if$ no_overflow_$f(m,$ value_$f(x) \times$ value_$f(y)))$ $then$
        (is_finite_$f$(result)$\wedge$
        value_$f$(result) $=$ round_$f(m,$ value_$f(x) \times$ value_$f(y)))$
            // finite times finite without overflow
        $else$ (overflow_value($m$, result)))
            // finite times finite with overflow
    $\wedge$ product_sign_$f$(result, $x$, $y$)
            // in any case, sign of result is product of signs
    }
```

where

$\mathsf{is\_zero\_}f(x:f) := \mathsf{class\_}f(x) = \mathsf{Finite}\ \wedge \mathsf{value\_}f(x) = 0$

$$\begin{array}{l}\mathsf{product\_sign\_}f(z:f, x:f, y:f) :=\\ \qquad\qquad (\mathsf{same\_sign\_}f(x,y) \Rightarrow \mathsf{sign\_}f(z) = \mathsf{Positive})\wedge\\ \qquad\qquad (\mathsf{diff\_sign\_}f(x,y) \Rightarrow \mathsf{sign\_}f(z) = \mathsf{Negative})\end{array}$$

$$
\begin{array}{l}
\mathsf{overflow\_value}(m : \mathsf{mode}, x : f) := \\
\quad (m = \mathsf{Down} \Rightarrow \\
\qquad (\mathsf{sign\_}f(x) = \mathsf{Negative} \Rightarrow \mathsf{is\_infinite\_}f(x)) \wedge \\
\qquad (\mathsf{sign\_}f(x) = \mathsf{Positive} \Rightarrow \mathsf{is\_finite\_}f(x) \wedge \mathsf{value\_}f(x) = \mathsf{max\_}f)) \\
\ \wedge\ (m = \mathsf{Up} \Rightarrow \\
\qquad (\mathsf{sign\_}f(x) = \mathsf{Positive} \Rightarrow \mathsf{is\_infinite\_}f(x)) \wedge \\
\qquad (\mathsf{sign\_}f(x) = \mathsf{Negative} \Rightarrow \mathsf{is\_finite\_}f(x) \wedge \mathsf{value\_}f(x) = -\mathsf{max\_}f)) \\
\ \wedge (m = \mathsf{ToZero} \Rightarrow \mathsf{is\_finite\_}f(x) \wedge \\
\qquad (\mathsf{sign\_}f(x) = \mathsf{Negative} \Rightarrow \mathsf{value\_}f(x) = -\mathsf{max\_}f(f)) \wedge \\
\qquad (\mathsf{sign\_}f(x) = \mathsf{Positive} \Rightarrow \mathsf{value\_}f(x) = \mathsf{max\_}f(f))) \\
\ \wedge (m = \mathsf{NearestAway}\ \vee\ m = \mathsf{NearestEven} \Rightarrow \mathsf{is\_infinite\_}f(x))
\end{array}
$$

The auxiliary predicate overflow_value specifies the result of FP operations, in case the real result overflows, depending on its sign and the rounding mode. The predicate product_sign_f encodes the usual rule for the sign of a product. Those are reused for other operations.

5 Discharging Proof Obligations

Our aim is to support as many theorem provers as possible. However, we must consider provers that are able to understand first-order logic with integer and real arithmetic. Suitable automatic provers are those of the SMT-family (Satisfiability Modulo Theories) which support first-order quantification, such as Z3 [15], CVC3 [4], Yices [16], Alt-Ergo [13]. Due to the high expressiveness of the logic, these provers are necessarily incomplete. Hence we may also use interactive theorem provers, such as Coq and PVS.

Additionally, recall that our modeling involves an uninterpreted rounding function round_f. The Gappa tool [23] is an automatic prover, which specifically handles formulas made of equalities and inequalities over expressions involving real constants, arithmetic operations, and the round_f operator. But unlike SMT provers, Gappa does not handle quantifiers.

All the provers mentioned above are available as back-ends for the Frama-C environment and its Jessie/Why plugin [17]. Our experiments are conducted with those.

Example 3 (Example 1 continued). The VCs for our Remez approximation of exponential are the following:

– 3 VCs for the representability of constants `0.9890365552`; `1.130258690` and `0.5540440796` in double format. These are proved by Gappa and by SMT solvers. SMT solvers make use of the axiom (1) on round_f.
– 5 VCs for checking that the three multiplications and the two additions do not overflow. These are automatically proved by Gappa. This demonstrates the power of Gappa to check non-overflow of FP computations in practice.
– 1 VC for the validity of the post-condition. This is also proved by Gappa, as a consequence of the assertion. In other words, whenever Gappa is given the method error, it is able to add the rounding error to deduce the total error.

– 1 VC for the validity of the assertion stating the method error. This is not proved by any automatic prover. It corresponds to the VC:

$$\begin{aligned}&\forall x : \mathsf{Double}, |\mathsf{value\_Double}(x)| \leq 1.0 \Rightarrow\\ &\quad |0.9890365552 + 1.130258690 \times \mathsf{value\_Double}(x) + 0.5540440796\times\\ &\qquad \mathsf{value\_Double}(x) * \mathsf{value\_Double}(x) - \exp(\mathsf{value\_Double}(x))|\\ &\qquad\quad \leq (1 - 2^{-16}) \times 2^{-4}\end{aligned}$$

Indeed, $\mathsf{value\_Double}(x)$ is just an arbitrary real number here, and that formula is a pure real arithmetic formula. It is expected that no automatic prover proves it since they do not know anything about the $\exp$ function. However, this VC can be proved valid using the Coq proof assistant, in a very simple way (2 lines of proof script to write) thanks to its `interval` tactic [23], which is able to bound mathematical expressions using interval arithmetic.

Example 4 (Interval example continued). Although the code for interval addition (Fig. 3) is very simple, it was not proved by automatic provers. We started an interactive proof in Coq, and saw that the proof was complex because it involved a large amount of different cases to distinguish, depending on whether interval bounds are finite or infinite, and whether an overflow occurs or not. Nevertheless, no case was difficult, and we found that the important property that automatic provers were missing was that for any format f and real x: $\mathsf{round\_}f(\mathsf{Down}, x) \leq x$, which can be proved correct using our Coq realization. Adding this property in our axiomatization of $\mathsf{round\_}f$ allows to perform the verification with SMT solvers.

Example 5 (Interval multiplication). To go further, we proved also the multiplication of intervals. Its code is given in Fig. 4. Notice the large number of branches. This code calls some auxiliary functions on intervals from Fig. 5. This was difficult to verify. First, we had to find proper contracts for the auxiliary functions: see the preconditions about signs for `mul_up` and `mul_dn`. Second, the number of cases is definitely larger than for addition: we got a total of 140 VCs, where each of them has a complex propositional structure, leading to consider a large number of subcases. By investigating in Coq the VCs which were not proved automatically, we were able to discover that SMT solvers were missing a few lemmas related to multiplication, e.g for all reals x, y, z and t:

$$(0 \leq x \leq z \wedge 0 \leq y \leq t) \Rightarrow x \times y \leq z \times t$$
$$(0 \leq z \leq x \wedge y \leq t \wedge y < 0) \Rightarrow x \times y \leq z \times t$$

and similar others.

The Z3 prover is able to validate all VCs except one (the first post-condition of `mul_up`). This is done in around 45s on a 3GHz CPU (each VCs is solved within 0.5s), whereas the remaining VC cannot be proved with a 2 minutes time limit. Fortunately, the CVC3 prover is able to solve the remaining VC, but misses 7 other VCs. CVC3 needs a similar amount of time. What is important here is the very good efficiency of SMT solvers, for dealing with all the cases coming from the complex propositional structures of VCs.

```
/*@ ensures \forall real a,b;
  @   in_interval(a,x) && in_interval(b,y) ==>
  @        in_interval(a*b,\result);          */
interval mul(interval x, interval y) {
  interval z;
  if (x.l < 0.0)
    if (x.u > 0.0)
      if (y.l < 0.0)
        if (y.u > 0.0) {
          z.l = min(mul_dn(x.l, y.u), mul_dn(x.u, y.l));
          z.u = max(mul_up(x.l, y.l), mul_up(x.u, y.u)); }
        else { z.l = mul_dn(x.u, y.l); z.u = mul_up(x.l, y.l); }
      else
        if (y.u > 0.0)
          { z.l = mul_dn(x.l, y.u); z.u = mul_up(x.u, y.u); }
        else { z.l = 0.0; z.u = 0.0; }
    else
      if (y.l < 0.0)
        if (y.u > 0.0)
          { z.l = mul_dn(x.l, y.u); z.u = mul_up(x.l, y.l); }
        else { z.l = mul_dn(x.u, y.u); z.u = mul_up(x.l, y.l); }
      else
        if (y.u > 0.0)
          { z.l = mul_dn(x.l, y.u); z.u = mul_up(x.u, y.l); }
        else { z.l = 0.0; z.u = 0.0; }
  else
    if (x.u > 0.0)
      if (y.l < 0.0)
        if (y.u > 0.0)
          { z.l = mul_dn(x.u, y.l); z.u = mul_up(x.u, y.u); }
        else { z.l = mul_dn(x.u, y.l); z.u = mul_up(x.l, y.u); }
      else
        if (y.u > 0.0)
          { z.l = mul_dn(x.l, y.l); z.u = mul_up(x.u, y.u); }
        else { z.l = 0.0; z.u = 0.0; }
    else { z.l = 0.0; z.u = 0.0; }
  return z;
}
```

Fig. 4. Multiplication of intervals

6 Related Works and Perspectives

There exist several formalizations of FP arithmetic in various proof environments: two variants in Coq [14,22] and one in PVS [24] exclude special values; one in ACL2 [27] and one in HOL-light [20] also deal with special values. Compared to those, our purely first-order axiomatization has the clear disadvantage of being incomplete, but has the advantage of allowing use of off-the-shelf automatic theorem provers. Our approach allows to incorporate FP reasoning in environments for program verification for general-purpose programming languages like C or Java.

```
/*@ requires !\is_NaN(x) && !\is_NaN(y);
  @ ensures \le_float(\result,x) && \le_float(\result,y);
  @ ensures \eq_float(\result,x) || \eq_float(\result,y);  */
double min(double x, double y) { return x < y ? x : y; }

/*@ requires !\is_NaN(x) && !\is_NaN(y);
  @ ensures \le_float(x,\result) && \le_float(y,\result);
  @ ensures \eq_float(\result,x) || \eq_float(\result,y);  */
double max(double x, double y) { return x > y ? x : y; }

/*@ requires !\is_NaN(x) && !\is_NaN(y);
  @ requires (\is_infinite(x) || \is_infinite(y))
  @                                ==> \sign(x) != \sign(y);
  @ requires (\is_infinite(x) && \is_finite(y)) ==> y != 0.0;
  @ requires (\is_infinite(y) && \is_finite(x)) ==> x != 0.0;
  @ ensures double_le_real(\result,x*y);
  @ ensures (\is_infinite(x) || \is_infinite(y)) ==>
  @            \is_minus_infinity(\result);                  */
double mul_dn(double x, double y) { return x*y; }

/*@ requires !\is_NaN(x) && !\is_NaN(y);
  @ requires (\is_infinite(x) || \is_infinite(y))
  @                                ==> \sign(x) == \sign(y);
  @ requires (\is_infinite(x) && \is_finite(y)) ==> y != 0.0;
  @ requires (\is_infinite(y) && \is_finite(x)) ==> x != 0.0;
  @ ensures  real_le_double(x * y,\result);
  @ ensures (\is_infinite(x) || \is_infinite(y)) ==>
  @                       \is_plus_infinity(\result);        */
double mul_up(double x, double y) { return -(x*(-y)); }
```

Fig. 5. Auxiliary functions on intervals

In 2006, Leavens [21] described some pitfalls when trying to incorporate FP special values and specifically NaN values in a BISL like JML for Java. In its approach, FP numbers, rounding and such also appear in annotations, which cause several issues and traps for specifiers. We argue that our approach, using instead real numbers in annotations, solves these kind of problems.

In 2006, Reeber & Sawada [28] used the ACL2 system together with a automated tool to verify a FP multiplier unit. Although their goal is at a significantly different concern (hardware verification instead of software behavioral properties) it is interesting to remark that they came to a similar conclusion, that using interactive proving alone is not practicable, but incorporating an automatic tool is successful.

In Section 5, we have seen that we needed both SMT solvers, Gappa for reasoning about rounding, and interactive proving to prove all VCs. Improving cooperation of provers is an interesting perspective, e.g. like in the Jahob verification tool for Java [31] which selects the prover to call depending on the class of goal (but does not support FP). Turning the Gappa techniques for FP into some specific built-in theory for SMT solvers should be considered. Integrating SMT solvers into interactive proving systems is also potentially very useful: possibility of calling Z3 and Vampyre from Isabelle/HOL has been experimented recently, and similar integration in Coq is in progress.

Another future work is to deal with programs, where FP computations do not strictly respect the IEEE standard, due to transformations made at compile-time (reorganization of expression order, use of fused multiply-add instructions) ; or at runtime by using extra precision (e.g., 80 bits FP precision in 387 processors) on intermediate calculations [8].

Discovering the proper annotations (e.g. contract for `mul_up` above) is essential for successful deductive verification. Another interesting future work is to automatically infer annotations, for example using abstract interpretation techniques [26] or abstraction refinement [9], to assist this task.

Acknowledgements. We thank G. Melquiond for his help in the use of the Gappa tool, the FP-specific Coq tactics, and more generally for his suggestions about the approach presented here.

References

1. IEEE standard for floating-point arithmetic. Technical report (2008), `http://dx.doi.org/10.1109/IEEESTD.2008.4610935`
2. Ayad, A., Marché, C.: Behavioral properties of floating-point programs. Hisseo publications (2009), `http://hisseo.saclay.inria.fr/ayad09.pdf`
3. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# Programming System: An Overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005)
4. Barrett, C., Tinelli, C.: CVC3. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 298–302. Springer, Heidelberg (2007)
5. Baudin, P., Filliâtre, J.-C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language (2008), `http://frama-c.cea.fr/acsl.html`
6. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software. LNCS (LNAI), vol. 4334. Springer, Heidelberg (2007)
7. Boldo, S., Filliâtre, J.-C.: Formal Verification of Floating-Point Programs. In: 18th IEEE International Symposium on Computer Arithmetic, Montpellier, France, pp. 187–194 (2007)
8. Boldo, S., Nguyen, T.M.T.: Hardware-independent proofs of numerical programs. In: Proceedings of the Second NASA Formal Methods Symposium. NASA Conference Publication, Washington D.C (April 2010)
9. Brillout, A., Kroening, D., Wahl, T.: Mixed abstractions for floating-point arithmetic. In: FMCAD'09, pp. 69–76. IEEE, Los Alamitos (2009)
10. Burdy, L., Cheon, Y., Cok, D., Ernst, M., Kiniry, J., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. International Journal on Software Tools for Technology Transfer (2004)
11. Chalin, P.: Reassessing JML's logical foundation. In: Proceedings of the 7th Workshop on Formal Techniques for Java-like Programs (FTfJP'05), Glasgow, Scotland (July 2005)
12. Cok, D.R., Kiniry, J.R.: ESC/Java2 implementation notes. Technical report (May 2007), `http://secure.ucd.ie/products/opensource/ESCJava2/ESCTools/docs/Escjava2-ImplementationNotes.pdf`
13. Conchon, S., Contejean, E., Kanig, J., Lescuyer, S.: CC(X): Semantical combination of congruence closure with solvable theories. In: Proceedings of the 5th International Workshop SMT'2007. ENTCS, vol. 198-2, pp. 51–69. Elsevier Science Publishers, Amsterdam (2008)

14. Daumas, M., Rideau, L., Théry, L.: A generic library for floating-point numbers and its application to exact computing. In: Boulton, R.J., Jackson, P.B. (eds.) TPHOLs 2001. LNCS, vol. 2152, p. 169+. Springer, Heidelberg (2001)
15. de Moura, L., Bjørner, N.: Z3, an efficient SMT solver, `http://research.microsoft.com/projects/z3/`
16. Dutertre, B., de Moura, L.: The Yices SMT solver (2006), `http://yices.csl.sri.com/tool-paper.pdf`
17. Filliâtre, J.-C., Marché, C.: The Why/Krakatoa/Caduceus platform for deductive program verification. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 173–177. Springer, Heidelberg (2007)
18. The Frama-C platform (2008), `http://www.frama-c.cea.fr/`
19. Goldberg, D.: What every computer scientist should know about floating-point arithmetic. ACM Computing Surveys 23(1), 5–48 (1991)
20. Harrison, J.: Floating point verification in HOL Light: The exponential function. Formal Methods in System Design 16(3), 271–305 (2000)
21. Leavens, G.: Not a number of floating point problems. Journal of Object Technology 5(2), 75–83 (2006)
22. Melquiond, G.: Floating-point arithmetic in the Coq system. In: Proceedings of the 8th Conference on Real Numbers and Computers, pp. 93–102. Santiago de Compostela, Spain (2008), `http://gappa.gforge.inria.fr/`
23. Melquiond, G.: Proving bounds on real-valued functions with computations. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 2–17. Springer, Heidelberg (2008), `http://www.lri.fr/~melquion/soft/coq-interval/`
24. Miner, P.S.: Defining the IEEE-854 floating-point standard in PVS. Technical Memorandum 110167, NASA Langley (1995)
25. Monniaux, D.: The pitfalls of verifying floating-point computations. ACM Transactions on Programming Languages and Systems 30(3), 12 (2008)
26. Monniaux, D.: Automatic modular abstractions for linear constraints. In: 36th ACM Symposium POPL 2009, pp. 140–151 (2009)
27. Moore, J.S., Lynch, T., Kaufmann, M.: A mechanically checked proof of the correctness of the kernel of the AMD5k86 floating-point division algorithm. IEEE Transactions on Computers 47(9), 913–926 (1998)
28. Reeber, E., Sawada, J.: Combining ACL2 and an automated verification tool to verify a multiplier. In: Sixth International Workshop on the ACL2 Theorem Prover and its Applications, pp. 63–70. ACM, New York (2006)
29. Schulte, W., Xia, S., Smans, J., Piessens, F.: A glimpse of a verifying C compiler, `http://www.cs.ru.nl/~tews/cv07/cv07-smans.pdf`
30. The Coq Development Team. The Coq Proof Assistant Reference Manual – Version V8.2 (2008), `http://coq.inria.fr`
31. Zee, K., Kuncak, V., Rinard, M.: Full functional verification of linked data structures. In: PLDI'08, pp. 349–361. ACM, New York (2008)

Verifying Safety Properties with the TLA$^+$ Proof System

Kaustuv Chaudhuri[1], Damien Doligez[2], Leslie Lamport[3], and Stephan Merz[4]

[1] INRIA Saclay, France
kaustuv.chaudhuri@inria.fr
[2] INRIA Rocquencourt, France
damien.doligez@inria.fr
[3] Microsoft Research Silicon Valley, USA
http://lamport.org
[4] INRIA Nancy, France
stephan.merz@inria.fr

1 Overview

TLAPS, the TLA$^+$ proof system, is a platform for the development and mechanical verification of TLA$^+$ proofs. The TLA$^+$ proof language is declarative, and understanding proofs requires little background beyond elementary mathematics. The language supports hierarchical and non-linear proof construction and verification, and it is independent of any verification tool or strategy. Proofs are written in the same language as specifications; engineers do not have to translate their high-level designs into the language of a particular verification tool. A *proof manager* interprets a TLA$^+$ proof as a collection of *proof obligations* to be verified, which it sends to *backend verifiers* that include theorem provers, proof assistants, SMT solvers, and decision procedures.

The first public release of TLAPS is available from [1], distributed with a BSD-like license. It handles almost all the non-temporal part of TLA$^+$ as well as the temporal reasoning needed to prove standard safety properties, in particular invariance and step simulation, but not liveness properties. Intuitively, a safety property asserts what is permitted to happen; a liveness property asserts what must happen; for a more formal overview, see [3,10].

2 Foundations

TLA$^+$ is a formal language based on TLA (the Temporal Logic of Actions) [12]. It was designed for specifying the high-level behavior of concurrent and distributed systems, but it can be used to specify safety and liveness properties of any discrete system or algorithm. A behavior is a sequence of states, where a state is an assignment of values to *state variables*. Safety properties are expressed by describing the allowed steps (state transitions) in terms of *actions*, which are first-order formulas involving two copies v and v' of each state variable, where v denotes the value of the variable at the *current* state and v' its value at the *next* state. These properties are proved by reasoning about actions, using a small and restricted amount of temporal reasoning. Proving liveness properties requires propositional linear-time temporal logic reasoning plus a few TLA proof rules.

It has always been possible to assert correctness properties of systems in TLA$^+$, but not to write their proofs. We have added proof constructs based on a hierarchical style for writing informal proofs [11]. The current version of the language is essentially the same as the version described elsewhere [7]. Here, we describe only the TLAPS proof system. Hierarchical proofs are a stylistic variant of natural deduction with lemmas and have been used in other declarative proof languages [8,14,15]. A hierarchical proof is either a sequence of steps together with their proofs, or a leaf (lowest-level) proof that simply states the known facts (previous steps and theorems) and definitions from which the desired conclusion follows. The human reader or a backend verifier must ensure that the leaf proofs are correct in their interpretation of TLA$^+$ to believe the entire proof.

The TLAPS proof manager, TLAPM, reads a (possibly incomplete) hierarchical proof and invokes the backend verifiers to verify the leaf proofs. One important backend is Isabelle/TLA$^+$, which is an implementation of TLA$^+$ as an Isabelle object logic (see Section 4.1). Isabelle/TLA$^+$ can be used directly with Isabelle's generic proof methods, or other certifying backend verifiers can produce proofs that are checked by Isabelle/TLA$^+$. Currently, the only certifying backend is the Zenon theorem prover [4]. Among the non-certifying backends is a generic SMT-LIB-based backend for SMT solvers, and a decision procedure for Presburger arithmetic. We plan to replace these with certifying implementations such as the SMT solver veriT [5] and certifying implementations of decision procedures [6].

TLAPS is intended for avoiding high-level errors in systems, not for providing a formal foundation for mathematics. It is far more likely for a system error to be caused by an incomplete or incorrect specification than by an incorrect proof inadvertently accepted as correct due to bugs in TLAPS. Although we prefer certifying backends whenever possible, we include non-certifying backends for automated reasoning in important theories such as arithmetic.

3 Proof Management

A TLA$^+$ specification consists of a root module that can (transitively) import other modules by extension and parametric instantiation. Each module consists of a number of parameters (state variables and uninterpreted constants), definitions, and theorems that may have proofs. TLAPS is run by invoking the Proof Manager (TLAPM) on the root module and telling it which proofs to check. In the current version, we use pragmas to indicate the proofs that are not to be checked, but this will change when TLAPS is integrated into the TLA$^+$ Toolbox IDE [2]. The design of TLAPM for the simple constant expressions of TLA$^+$ was described in [7]; this section explains the further processing required to support more of the features of TLA$^+$. TLAPM first flattens the module structure, since the module language of TLA$^+$ is not supported by backend verifiers, which will likely remain so in the future.

Non-constant reasoning: A TLA$^+$ module parameter is either a *constant* or a (state) *variable*. Constants are independent of behaviors and have the same value in each state of the behavior, while a variable can have different values in different states. Following the tradition of modal and temporal logics, TLA$^+$ formulas do not explicitly refer to states. Instead, action formulas are built from two copies v and v' of variables that refer to the values before and after the transition. More generally, the prime operator $'$ can be applied to an entire expression e, with e' representing the value of e at the state after a step. A *constant expression* e is one that does not involve any state variables, and is therefore equal to e'. (Double priming is not allowed in TLA$^+$; the TLA$^+$ syntactic analyzer catches such errors.)

Currently, all TLAPS backends support logical reasoning only on constant expressions. The semantics of the prime operator is therefore syntactically approximated as follows: it is commuted with all ordinary operators of mathematics and is absorbed by constant parameters. Thus, if e is the expression $(u = v + 2 * c)$ where u and v are variables and c a constant, then e' equals $u' = v' + 2 * c$. TLAPM currently performs such rewrites and its rewrite engine is trusted.

Operators and substitutivity: At any point in the scope of its definition, a user-defined operator is in one of two states: *usable* or *hidden*. A usable operator is one whose definition may be *expanded* in a proof; for example, if the operator P defined by $P(x, y) \triangleq x + 2 * y$ is usable, then TLAPM may replace $P(2, 20)$ with $2 + 2 * 20$ (but *not* with 42, which requires proving that $2 + 2 * 20 = 42$). A user-defined operator is hidden by default; it is made usable in a particular leaf proof by explicitly citing its definition, or for the rest of the current subproof by a USE step (see [7] for the semantics of USE).

Because TLA$^+$ is a modal logic, it contains operators that do not obey substitutivity, which underlies Leibniz's principle of equality. For example, from $(u = 42) = \text{TRUE}$ one cannot deduce $(u = 42)' = \text{TRUE}'$, *i.e.*, $u' = 42$. A unary operator $O(\_)$ is *substitutive* if $e = f$ implies $O(e) = O(f)$, for all expressions e and f. This definition is extended in the obvious way to operators with multiple arguments. Most of the modal primitive operators of TLA$^+$ are not substitutive; and an operator defined in terms of non-substitutive operators can be non-substitutive. If a non-substitutive operator is usable, then TLAPM expands its definition during preprocessing, as described in the previous paragraph; if it is hidden, then TLAPM replaces its applications by cryptographic hashes of its text to prevent unsound inferences by backend verifiers. This is a conservative approximation: for example, it prevents proving $O(e \land f) = O(f \land e)$ for a hidden non-substitutive operator O. Users rarely define non-substitutive operators, so there seems to be no urgent need for a more sophisticated treatment.

Subexpression references: A fairly novel feature of the TLA$^+$ proof language is the ability to refer to arbitrary subexpressions and instances of operators, theorems, and proof steps that appear earlier in the module or in imported modules, reducing the verbosity and increasing the maintainability of TLA$^+$ proofs. *Positional* references denote a path through the abstract syntax; for example, for the definition, $O(x, y) \triangleq x = 20 * y + 2$, the reference $O(3, 4)!2!1$ resolves to the first subexpression of the second subexpression of $O(3, 4)$, *i.e.*, $20 * 4$. Subexpressions can also be labelled and accessed via *labelled* references. For example, for $O(x, y) \triangleq x = l{::}(y * 20) + 2$, the reference $O(3, 4)!l$ refers to $4 * 20$ and will continue to refer to this expression even if the definition of O is later modified to $O(x, y) \triangleq x = 7 * y^2 + l{::}(20 * y) + 2$. TLAPM replaces all subexpression references with the expressions they resolve to prior to further processing.

Verifying obligations: Once an obligation is produced and processed as described before, TLAPM invokes backend verifiers on the proof obligations corresponding to the leaf proofs. The default procedure is to invoke the Zenon theorem prover first. If Zenon succeeds in verifying the obligation, it produces an Isabelle/Isar proof script that can be checked by Isabelle/TLA$^+$. If Zenon fails to prove an obligation, then Isabelle/TLA$^+$ is instructed to use one of its automated proof methods. The default procedure can be modified through pragmas that instruct TLAPM to bypass Zenon, use particular Isabelle tactics, or use other backends. Most users will invoke the pragmas indirectly by using particular theorems from the standard `TLAPS` module. For instance, using the theorem named `SimpleArithmetic` in a leaf proof causes TLAPM to invoke a decision procedure for Presburger arithmetic for that proof. The user can learn what standard theorems can prove what kinds of assertions by reading the documentation, but she does not need to know how such standard theorems are interpreted by TLAPM.

4 Backend Verifiers

4.1 Isabelle/TLA$^+$

Isabelle/TLA$^+$ is an axiomatization of TLA$^+$ in the generic proof assistant Isabelle [?]. It embodies the semantics of the constant fragment of TLA$^+$ in TLAPS; as mentioned in Section 2, it is used to certify proofs found by automatic backend verifiers. We initially considered encoding TLA$^+$ in one of the existing object logics that come with the Isabelle distribution, such as Isabelle/ZF or Isabelle/HOL. However, this turned out to be inconvenient, mainly because TLA$^+$ is untyped. (Indeed, TLA$^+$ does not even distinguish between propositions and terms.) We would have had to define a type of TLA$^+$ values inside an existing object logic and build TLA$^+$-specific theories for sets, functions, arithmetic *etc.*, essentially precluding reuse of the existing infrastructure.

Isabelle/TLA$^+$ defines classical first-order logic based on equality, conditionals, and Hilbert's choice operator. All operators take arguments and return values of the single type `c` representing TLA$^+$ values. Set theory is based on the uninterpreted predicate symbol $\in$ and standard Zermelo-Fränkel axioms. Unlike most presentations of ZF, TLA$^+$ considers functions to be primitive objects rather than sets of ordered pairs. Natural numbers with zero and successor are introduced using Hilbert's choice as some set satisfying the Peano axioms; the existence of such a set is established from the ZF axioms. Basic arithmetic operators over natural numbers such as $\leq$, $+$, and $*$ are defined by primitive recursion, and division and modulus are defined in terms of $+$ and $*$. Tuples and sequences are defined as functions whose domains are initial intervals of the natural numbers. Characters are introduced as pairs of hexadecimal digits, and strings as sequences of characters. Records are functions whose domains are finite sets of strings. Isabelle's flexible parser and pretty-printer transparently converts between the surface syntax and the internal representation. The standard library introduces basic operations for these data structures and proves elementary lemmas about them. It currently provides more than 1400 lemmas and theorems, corresponding to about 200 pages of pretty-printed Isar text. Isabelle/TLA$^+$ sets up Isabelle's generic automated proof methods (rewriting, tableau and resolution provers, and their combinations).

It is a testimony to the genericity of Isabelle that setting up a new object logic was mostly a matter of perseverance and engineering. Because TLA$^+$ is untyped, many theorems come with hypotheses that express "typing conditions". For example, proving

$n + 0 = n$ requires proving that n is a number. When the semantics of TLA$^+$ allowed us to do so, we set up operators so that they return the expected “type”; for example, $p \wedge q$ is guaranteed to be a Boolean value whatever its arguments p and q are. In other cases, typechecking is left to Isabelle’s automatic proof methods; support for conditional rewrite rules in Isabelle’s simplifier was essential to make this work.

4.2 Zenon

Zenon is a theorem prover for first-order logic with Hilbert’s choice operator and equality. It is a *proof-producing* theorem prover: it outputs formal proof scripts for the theorems it proves. Zenon was extended with a backend that produces proofs in Isar syntax; these proofs use lemmas based on the Isabelle/TLA$^+$ object logic and are passed to Isabelle for verification. Zenon is therefore not part of the trusted code base of TLAPS.

Zenon had to be extended with deduction rules specific to TLA$^+$: rules for reasoning about set-theoretic operators, for the CASE operator of TLA$^+$, for set extensionality and function extensionality, for reasoning directly on bounded quantifiers (which is not needed in theory but is quite important for efficiency), and for reasoning about functions, strings, *etc*. Interestingly, Hilbert’s choice operator was already used in Zenon for Skolemization, so we were easily able to support the CHOOSE operator of TLA$^+$.

Future work includes adding rules to deal with tuples, sequences, records, and arithmetic, and improving the handling of equality. While there is some overlap between Zenon and Isabelle’s automatic methods as they are instantiated in Isabelle/TLA$^+$, in practice they have different strong points and there are many obligations where one succeeds while the other fails. Zenon uses Isabelle’s automatic proof tactics for some of the elementary steps when it knows they will succeed, in effect using these tactics as high-level inference rules.

4.3 Other Backends

The first release of TLAPS comes with some additional non-certifying backends. For arithmetic reasoning we have:

– An SMT-LIB based backend that can be linked to any SMT solver. Obligations are rewritten into the AUFLIRA theory of SMT-LIB, which generally requires omitting assumptions that lie outside this theory. This backend is needed for reasoning about real numbers. We have successfully used Yices, CVC3, Z3, veriT and Alt-Ergo in our examples. In future work we might specialize this generic backend for particular solvers that can reason about larger theories.
– A Presburger arithmetic backend, for which we have implemented Cooper’s algorithm. Our implementation is tailored to certain elements of TLA$^+$ that are not normally part of the Presburger fragment, but can be (conservatively) injected.

For both these backends, TLAPM performs a simple and highly conservative sort detection pass for bound identifiers. Both backends are currently non-certifying, but we plan to replace them with certifying backends in the future. In particular, we are integrating the proof-producing SMT solver veriT [5], with the goal of tailoring it for discharging TLA$^+$ proof obligations.

5 Proof Development

Writing proofs is hard and error-prone. Before attempting to prove correctness of a TLA$^+$ specification, we first check finite instances with the TLC model checker [12]. This usually catches numerous errors quickly – much more quickly than by trying to prove it correct. Only after TLC can find no more errors do we try to write a proof.

The TLA$^+$ language supports a hierarchical, non-linear proof development process that we find indispensable for larger proofs [**?**]. The highest-level proof steps are derived almost without thinking from the structure of the theorem to be proved. For example, a step of the form $P_1 \lor \ldots \lor P_n \Rightarrow Q$ is proved by the sequence of steps asserting $P_i \Rightarrow Q$, for each i. When the user reaches a simple enough step, she first tries a fully automatic proof using a leaf directive citing the facts and definitions that appear relevant. If that fails, she begins a new level with a sequence of proof-less assertion steps that simplify the assertion, and a final QED step asserting that the goal follows from these steps. These new lower-level steps are tuned until the QED step is successfully verified. Then, the steps are proved in any order. (The user can ask TLAPM what steps have no proofs.) The most common reason that leaf proofs fail to verify is that the user has forgotten to use some fact or definition. When a proof fails, TLAPM prints the usable hypotheses and the goal, with usable definitions expanded. Examining this output often reveals the omission.

This kind of hierarchical development cries for a user interface that allows one to see what has been proved, hide irrelevant parts of the proof, and easily tell TLAPM what it should try to prove next. Eventually, these functions will be provided by the TLA$^+$ Toolbox. (It now performs only the hiding.) When TLAPS is integrated into the Toolbox, writing the specification, model-checking it, and writing a proof will be one seamless process. Meanwhile, we have written an Emacs mode that allows hierarchical viewing of proofs and choosing which parts to prove.

We expect most users to assume simple facts about data structures such as sequences rather than spending time proving them – especially at the beginning, before we have developed libraries of such facts for common data structures. Relying on unchecked assumptions would be a likely source of errors; it is easy to make a mistake when writing an “obviously true” assumption. Such assumptions should therefore be model-checked with TLC.

5.1 Example Developments

We have written a number of proofs, mainly to find bugs and see how well the prover works. Most of them are in the `examples` sub-directory of the TLAPS distribution. Here are the most noteworthy:

– *Peterson’s Mutual Exclusion Algorithm.* This is a standard shared memory mutual exclusion algorithm. The algorithm (in its 2-process version) is described in a dozen lines of PlusCal, an algorithm language that is automatically translated to TLA$^+$. The proof of mutual exclusion is about 130 lines long.
– *The Bakery Algorithm with Atomic Reads and Writes.* This is a more complicated standard mutual exclusion example; its proof (for the N-process version) is 800 lines long.

– *Paxos.* We have specified a high-level version of the well-known Paxos consensus algorithm as a trivial specification of consensus and two refinement steps—a total of 100 lines of TLA$^+$. We have completed the proof of the first refinement and most of the proof of the second. The first refinement proof is 550 lines long; we estimate that the second will be somewhat over 1000 lines.

Tuning the back-end provers has made them more powerful, making proofs easier to write. While writing machine-checked proofs remains tiresome and more time consuming than we would like, it has not turned out to be difficult once the proof idea has been understood.

Acknowledgements. Georges Gonthier helped design the TLA$^+$ proof language. Jean-Baptiste Tristan wrote the (incomplete) Paxos proof.

References

1. TLAPS web-site, `http://www.msr-inria.inria.fr/~doligez/tlaps`
2. TLA$^+$ Toolbox, `http://www.tlaplus.net/tools/tla-toolbox/`
3. Alpern, B., Schneider, F.B.: Defining liveness. Inf. Process. Lett. 21(4), 181–185 (1985)
4. Bonichon, R., Delahaye, D., Doligez, D.: Zenon: An extensible automated theorem prover producing checkable proofs. In: Dershowitz, N., Voronkov, A. (eds.) LPAR 2007. LNCS (LNAI), vol. 4790, pp. 151–165. Springer, Heidelberg (2007)
5. Bouton, T., de Oliveira, D.C., Déharbe, D., Fontaine, P.: veriT: An open, trustable and efficient SMT-solver. In: Schmidt, R.A. (ed.) CADE 2009. LNCS, vol. 5663, pp. 151–156. Springer, Heidelberg (2009)
6. Chaieb, A., Nipkow, T.: Proof synthesis and reflection for linear arithmetic. Journal of Automated Reasoning 41, 33–59 (2008)
7. Chaudhuri, K., Doligez, D., Lamport, L., Merz, S.: A TLA$^+$ Proof System. In: Sutcliffe, G., Rudnicki, P., Schmidt, R., Konev, B., Schulz, S. (eds.) Workshop on Knowledge Exchange: Automated Provers and Proof Assistants. CEUR Workshop Proceedings, vol. 418, pp. 17–37 (2008)
8. Corbineau, P.: A declarative proof language for the Coq proof assistant. In: Miculan, M., Scagnetto, I., Honsell, F. (eds.) TYPES 2007. LNCS, vol. 4941, pp. 69–84. Springer, Heidelberg (2008)
9. Gafni, E., Lamport, L.: Disk Paxos. Distributed Computing 16(1), 1–20 (2003)
10. Lamport, L.: Proving the correctness of multiprocess programs. IEEE Trans. Softw. Eng. SE-3(2), 125–143 (1977)
11. Lamport, L.: How to write a proof. American Mathematical Monthly 102(7), 600–608 (1995)
12. Lamport, L.: Specifying Systems. Addison-Wesley, Boston (2003)
13. Paulson, L.C. (ed.): Isabelle. LNCS, vol. 828. Springer, Heidelberg (1994)
14. Rudnicki, P.: An overview of the Mizar project. In: Workshop on Types for Proofs and Programs, Bastad, Sweden, pp. 311–332 (1992)
15. Wenzel, M.: The Isabelle/Isar reference manual (December 2009), `http://isabelle.in.tum.de/dist/Isabelle/doc/isar-ref.pdf`

MUNCH - Automated Reasoner for Sets and Multisets

Ruzica Piskac and Viktor Kuncak

Swiss Federal Institute of Technology Lausanne (EPFL)
`firstname.lastname@epfl.ch`

Abstract. This system description provides an overview of the MUNCH reasoner for sets and multisets. MUNCH takes as the input a formula in a logic that supports expressions about sets, multisets, and integers. Constraints over collections and integers are connected using the cardinality operator. Our logic is a fragment of logics of popular interactive theorem provers, and MUNCH is the first fully automated reasoner for this logic. MUNCH reduces input formulas to equisatisfiable linear integer arithmetic formulas. MUNCH reasoner is publicly available. It is implemented in the Scala programming language and currently uses the SMT solver Z3 to solve the generated integer linear arithmetic constraints.

1 Introduction

Applications in software verification and interactive theorem proving often involve reasoning about sets of objects. Cardinality constraints on such collections also arise in these scenarios. Multisets arise for analogous reasons as sets: abstracting the content of linked data structure with duplicate elements leads to multisets. Multisets (and sets) are widely present in the theorem proving community. Interactive theorem provers such as Isabelle [3], Why [1] or KIV [4] specify theories of multisets with cardinality constraints. They prove a number of theorems about multisets to enable their use in interactive verification. However, all those tools require a certain level of interaction. Our tool is the first automated theorem prover for multisets with cardinality constraints, which can check satisfiability of formulas belonging to a very expressive logic (defined in Figure 1) entirely automatically.

This system description presents the implementation of the decision procedure for satisfiability of multisets with cardinality constraints [8]. We evaluated our implementation by checking the unsatisfiability of negations of verification conditions for the correctness of mutable data structure implementations. If an input formula is satisfiable, our tool generates a model, which can be used to construct a counterexample trace of the checked program.

2 Description of the MUNCH Implementation

2.1 Input Language

A multiset (bag) is a function m from a fixed finite set E to $\mathbb{N}$, where $m(e)$ denotes the number of times an element e occurs in the multiset (multiplicity of

e). Our logic includes multiset operations such as multiplicity-preserving union and the intersection. In addition, it supports an infinite family of relations on multisets defined point-wise, one relation for each Presburger arithmetic formula. For example, $(m_1 \cap m_2)(e) = \min(m_1(e), m_2(e))$ and $m_1 \subseteq m_2$ means $\forall e. m_1(e) \leq m_2(e)$. Our logic supports using such point-wise operations for arbitrary quantifier-free Presburger arithmetic formulas. The logic also supports the cardinality operator that returns the number of elements in a multiset. Figure 1 summarizes the language of multisets with cardinality constraints (MAPA). There are two levels at which integer linear arithmetic constraints occur: to define point-wise operations on multisets (inner formulas) and to define constraints on cardinalities of multisets (outer formulas). Integer variables from outer formulas cannot occur within inner formulas.

Top-level formulas:

$$F ::= M{=}M \mid M \subseteq M \mid \forall e.\mathsf{F}^{\mathsf{in}} \mid \mathsf{A}_{\mathsf{out}} \mid F \wedge F \mid \neg F$$

Outer linear arithmetic formulas:

$$\mathsf{F}_{\mathsf{out}} ::= \mathsf{A}_{\mathsf{out}} \mid \mathsf{F}_{\mathsf{out}} \wedge \mathsf{F}_{\mathsf{out}} \mid \neg \mathsf{F}_{\mathsf{out}}$$

$$\mathsf{A}_{\mathsf{out}} ::= \mathsf{t}_{\mathsf{out}} \leq \mathsf{t}_{\mathsf{out}} \mid \mathsf{t}_{\mathsf{out}}{=}\mathsf{t}_{\mathsf{out}} \mid (\mathsf{t}_{\mathsf{out}}, \ldots, \mathsf{t}_{\mathsf{out}}){=}\textstyle\sum_{\mathsf{F}^{\mathsf{in}}}(\mathsf{t}^{\mathsf{in}}, \ldots, \mathsf{t}^{\mathsf{in}})$$

$$\mathsf{t}_{\mathsf{out}} ::= k \mid C \mid \mathsf{t}_{\mathsf{out}} + \mathsf{t}_{\mathsf{out}} \mid C \cdot \mathsf{t}_{\mathsf{out}} \mid \mathsf{ite}(\mathsf{F}_{\mathsf{out}}, \mathsf{t}_{\mathsf{out}}, \mathsf{t}_{\mathsf{out}}) \mid \; |M|$$

Inner linear arithmetic formulas:

$$\mathsf{F}^{\mathsf{in}} ::= \mathsf{t}^{\mathsf{in}} \leq \mathsf{t}^{\mathsf{in}} \mid \mathsf{t}^{\mathsf{in}}{=}\mathsf{t}^{\mathsf{in}} \mid \mathsf{F}^{\mathsf{in}} \wedge \mathsf{F}^{\mathsf{in}} \mid \neg \mathsf{F}^{\mathsf{in}}$$

$$\mathsf{t}^{\mathsf{in}} ::= m(e) \mid C \mid \mathsf{t}^{\mathsf{in}} + \mathsf{t}^{\mathsf{in}} \mid C \cdot \mathsf{t}^{\mathsf{in}} \mid \mathsf{ite}(\mathsf{F}^{\mathsf{in}}, \mathsf{t}^{\mathsf{in}}, \mathsf{t}^{\mathsf{in}})$$

Multiset expressions:

$$M ::= m \mid \emptyset \mid M \cap M \mid M \cup M \mid M \uplus M \mid M \setminus M \mid M \setminus\setminus M \mid \mathsf{setof}(M)$$

C - integer constant   Variables: e - fixed index, k - integer, m - multiset

Fig. 1. Quantifier-Free Multiset Constraints with Cardinality Operator (MAPA)

This logic subsumes the BAPA logic [5]. If a formula reasons only about sets, this can be added by explicitly stating for each set variable S that it is a set: $\forall e.(S(e) = 0 \vee S(e) = 1)$.

2.2 NP vs. NEXPTIME Algorithm in Implementations

Checking satisfiability of MAPA formulas is an NP-complete problem [9]. Our first implementation was based on the algorithm used to establish this optimal complexity, but we found that the running times were impractical due to large constants. MUNCH therefore currently uses the conceptually simpler algorithm in [8]. Despite its NEXPTIME worst-case complexity, we have found that the algorithm from [8], when combined with additional simplifications, results in a tool that exhibits acceptable performance. Our implementation often avoids the worst-case complexity of the most critical task, the computation of semilinear sets, by leveraging the special structure of formulas that we need to process (see Section 2.4).

2.3 System Overview

Figure 2 provides a high-level overview of the reasoner.

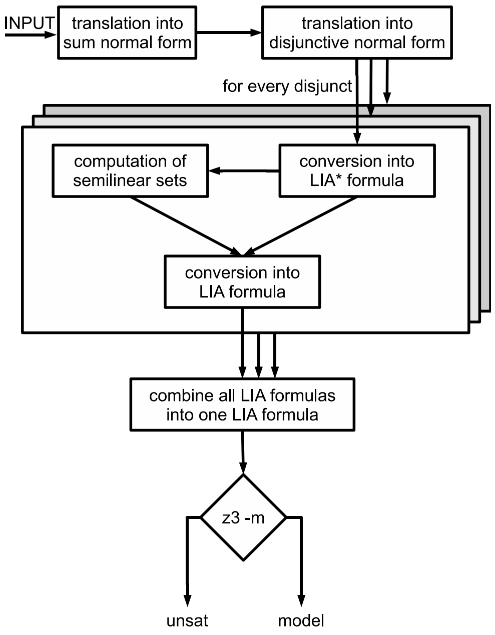

INPUT
translation into sum normal form
translation into disjunctive normal form
for every disjunct
computation of semilinear sets
conversion into LIA* formula
conversion into LIA formula
combine all LIA formulas into one LIA formula
z3 -m
unsat
model

Fig. 2. Phases in checking formula satisfiability. MUNCH translates the input formula through several intermediate forms, preserving satisfiability in each step.

Given an input formula (Figure 1), MUNCH converts it into the *sum normal form*

$$P \ \wedge \ (u_1, \ldots, u_n) = \Sigma_{e \in E}(t_1, \ldots, t_n) \ \wedge \ \forall e.F$$

where

– P is a quantifier-free Presburger arithmetic formula without any multiset variables, and sharing integer variables only with terms $u_1, \ldots, u_n$
– the variables in $t_1, \ldots, t_n$ and F occur only as expressions of the form $m(e)$ for m a multiset variable and e the fixed index variable

The algorithm that reduces a formula to its sum normal form runs in polynomial time and is applicable to every input formula.

The derived formula is further translated into the logic that we call LIA^* [9]. LIA^* is linear integer arithmetic extended with the * operator. The * operator is defined on sets of vectors by $C^* = \{\boldsymbol{v}_1 + \ldots + \boldsymbol{v}_n \mid \boldsymbol{v}_1, \ldots \boldsymbol{v}_n \in C \wedge n \geq 0\}$. The new atom that we add to the linear integer arithmetic syntax is $\boldsymbol{u} \in \{\boldsymbol{x} \mid F(\boldsymbol{x})\}^*$, where F is a Presburger arithmetic formula.

A formula in the sum normal form

$$P \ \wedge \ (u_1, \ldots, u_n) = \sum_{e \in E} (t_1, \ldots, t_n) \ \wedge \ \forall e.F$$

is equisatisfiable with the formula

$$P \ \wedge \ (u_1, \ldots, u_n) \in \{(t'_1, \ldots, t'_n) \mid F'\}^*$$

where the terms t'_i and the formula F' are formed from the terms t_i and the formula F in the following way: for each multiset expression $m_j(e)$ we introduce a fresh new integer variable x_j and then we substitute each occurrence of $m_j(e)$ in the terms t_i and the formula F with the corresponding variable x_j. The equisatisfiability theorem between the two formulas follows from the definitions. Given a finite set of vectors $(t'_1, \ldots, t'_n)$ such that their sum is $(u_1, \ldots, u_n)$, we define the carrier set E to have as many elements as there are summands and define $m_j(e)$ to have the value of x_j in the corresponding summand. We use this theorem in the model reconstruction.

Model Reconstruction. Our tool outputs a model if an input formula is satisfiable. After all transformations, we obtain a linear arithmetic formula equisatisfiable to the input formula. If there is a satisfying assignment for the final formula, we use the constructive proofs of the equisatisfiability theorems to construct a model for the original formula.

2.4 Efficient Computation of Semilinear Sets and Elimination of the * Operator

The elimination of the * operator is done using semilinear sets. Let $S \subseteq \mathbb{Z}^m$ be a set of integer vectors and let $\boldsymbol{a} \in \mathbb{Z}^m$ be a integer vector. A linear set $LS(\boldsymbol{a}; S)$ is defined as $LS(\boldsymbol{a}; S) = \{\boldsymbol{a} + \boldsymbol{x}_1 + \ldots + \boldsymbol{x}_n \mid x_i \in S \wedge n \geq 0\}$. Note that vectors $\boldsymbol{x}_j$ and $\boldsymbol{x}_j$ can be equal and this way we can define a multiplication of a vector with a positive integer constant. A semilinear set Z is defined as a finite union of linear sets: $Z = \cup_{i=1}^{k} LS(a_i; S_i)$.

All vectors belonging to a semilinear set can be described as a solution set of a Presburger arithmetic formula. A classic result [2] shows that the converse also holds: the set of satisfying assignments of a Presburger arithmetic formula is a semilinear set.

Consider the set $\{(t'_1, \ldots, t'_n) \mid F'\}^*$. The set of all vectors which are the solution of formula F' is a semilinear set. It was shown in in [8,6] that applying the * operator on a semilinear set results in a set which can be described by a Presburger arithmetic formula. Consequently, applying the star operator on a semilinear set results in a new semilinear set. Because $\{(t'_1, \ldots, t'_n) \mid F'\}^*$ is a semilinear set, checking whether $(u_1, \ldots, u_n) \in \{(t'_1, \ldots, t'_n) \mid F'\}^*$ is effectively expressible as a Presburger arithmetic formula. This concludes that the elimination of the * operator results in a equisatisfiable Presburger arithmetic formula.

Efficient Computation of Semilinear Sets. The problem with this approach is that computing semilinear sets is expensive. The best know algorithms still run in the exponential time and are fairly complex [10].

For complexity reasons, we are avoiding to compute semilinear sets. Still, the exponential running time is unavoidable in this approach. Therefore, instead of developing an algorithm which computes semilinear sets for an arbitrary Presburger arithmetic formula, we split a formula into simpler parts for which we can easier compute semilinear sets. Namely, we convert formula F into a disjunctive normal form: $F'(\boldsymbol{t}) \equiv A_1(\boldsymbol{t}) \vee \ldots \vee A_m(\boldsymbol{t})$. This way checking whether $\boldsymbol{u} \in \{\boldsymbol{t} \mid F'(\boldsymbol{t})\}^*$ reduces to $\boldsymbol{u} = \boldsymbol{k}_1 + \ldots + \boldsymbol{k}_m \wedge \bigwedge_{j=1}^{m} \boldsymbol{k}_j \in \{\boldsymbol{t} \mid A_j(\boldsymbol{t})\}^*$. The next task is to eliminate the $*$ operator for the formula $\boldsymbol{k}_j \in \{\boldsymbol{t} \mid A_j(\boldsymbol{t})\}^*$, where A_j is a conjunction of linear arithmetic atoms. A_j can also be rewritten as a conjunction of equalities by introducing fresh non-negative variables. In most of the cases, computing a semilinear set is actually computing a linear set which can be done effectively, for example, using the Omega-test [11]. Since A_j is a conjunction of equations, we use simple rewriting rules. The problem of inequalities expressing that a term is non-negative in most cases is resolved by implicitly using them as non-negative coefficients. As an illustration, consider formula $m_0 = y + x$, where all variables have to be non-negative. All solutions are described with a linear set: $(m_0, y, x) = (0,0,0) + y(1,1,0) + x(1,0,1)$, i.e. $LS((0,0,0), \{(1,1,0),(1,0,1)\})$. This approach of using equalities and rewriting is highly efficient and works in most of the cases. We also support a simple version of the Omega test.

However, our implementation is not complete for the full logic described in Figure 1. There are cases where one cannot avoid the computation of a semilinear set. One of the examples where the MUNCH tool cannot find a solution is when there exists an inner formula of the form $\forall e.F_{in}(e)$ and $F_{in}(e)$ is a formula where none of the variables have coefficient 1. An example of such a formula is $\forall e.5m_1(e) + 7m_2(e) \leq 6m_3(e)$. If at least one variable has coefficient 1 after the simplifications, our tool works. In our experimental results, while processing formulas derived in verification, we did not encounter such a problem. Notice also that our tool is always complete for sets, so it can also be used as a complete reasoner for sets with cardinality constraints (with a doubly exponential worst-case bound on running time).

To summarize, out of each conjunct we derive an equisatisfiable Presburger arithmetic formula and this way the initial multiset constraints problem reduces to satisfiability of quantifier-free Presburger arithmetic formulas. To check satisfiability of such a formula, we invoke the SMT solver Z3 [7] with the option "-m". This option ensures that Z3 returns a model in case that the input formula is satisfiable. Since all our transformations are satisfiability preserving, we either return `unsat` or reconstruct a model for the initial multiset formula from the model returned by Z3.

3 Examples and Benchmarks

First we illustrate how the MUNCH reasoner works on a simple example, and then we show some benchmarks that we did.

Consider a simple multiset formula $|x \uplus y| = |x| + |y|$. Its validity is proved by showing that $|x \uplus y| \neq |x| + |y|$ is unsatisfiable. We chose such a simple formula so that we can easily present and analyze the tool's output. The intermediate formulas in the output correspond to the result of the individual reduction step described in Section 2.

```
Formula f3:
 NOT (|y PLUS x| = |y| + |x|)
Normalized formula f3:
 NOT (k0 = k1 + k2) AND FOR ALL e IN E. (m0(e) = y(e) + x(e)) AND
    (k0, k1, k2) = SUM {e in E,  TRUE } (m0(e), y(e), x(e))
Translated formula f3:
 NOT (k0 = k1 + k2) AND (k0, k1, k2) IN {(m0, y, x) | m0 = y + x }*
No more disjunctions:
  NOT (k0 = k1 + k2) AND k0 = u0  AND k1 = u1 AND k2 = u2 AND
   (u0, u1, u2) IN {(m0, y, x) | m0 = y + x }*
Semilinear set computation :
 ( m0, y, x )  | m0 = y + x,
semilinear set describing it is:
  List(0, 0, 0), List( List(1, 1, 0), List(1, 0, 1))
No more stars:
 NOT (k0 = k1 + k2) AND k0 = u0  AND k1 = u1 AND k2 = u2 AND
 u2 = 0 + 1*nu1 + 0 AND u1 = 0 + 0 + 1*nu0 AND u0 = 0 + 1*nu1 + 1*nu0
 AND ( NOT (mu0 = 0)  OR (nu1 = 0 AND nu0 = 0) )
---------------------
This formula is unsat
---------------------
```

The main problem we are facing for a more comprehensive evaluation of our tool is the lack of similar tools and benchmarks. Most benchmarks we were using are originally derived for reasoning about sets. Sometimes those formulas contain conditions that we do not need to consider when reasoning about multisets. This can especially be seen in Figure 3. Checking that an invariant on the size field of a data structure that implements a multiset is preserved after inserting 3

Property	#set vars	#multiset vars	time (s)
Correctness of efficient emptiness check	1	0	0.40
Correctness of efficient emptiness check	0	1	0.40
Size invariant after inserting an element in a list	2	1	0.46
Size invariant after inserting an element in a list	0	2	0.40
Size invariant after deleting an element from a list	0	2	0.35
Allocating and inserting 3 objects into a container	5	0	3.23
Allocating and inserting 3 objects into a container	0	5	0.40
Allocating and inserting 4 objects into a container	6	0	8.35

Fig. 3. Measurement of running times for checking verification conditions that arise in proving correctness of container data structures. Please see tool web page for more details.

objects requires 0.4 seconds. Checking the same property for a data structure implementing a set requires 3.23 seconds.

We could also not compare the MUNCH tool with interactive theorem provers since our tool is completely automated and does not require any interaction.

In the future we plan to integrate our tool into theorem provers for expressive higher-order logics and to incorporate it into software verification systems. This will also enable us to obtain further sets of benchmarks. Our tool and the presented examples can be found at the following URL:

`http://icwww.epfl.ch/~piskac/software/MUNCH/`

References

1. Filliâtre, J.C., Marché, C.: The Why/Krakatoa/Caduceus platform for deductive program verification. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 173–177. Springer, Heidelberg (2007), `http://www.lri.fr/~filliatr/ftp/publis/cav07.pdf`
2. Ginsburg, S., Spanier, E.: Semigroups, Pressburger formulas and languages. Pacific Journal of Mathematics 16(2), 285–296 (1966)
3. Isabelle: Isabelle - a generic proof assistant, `http://www.cl.cam.ac.uk/research/hvg/Isabelle/`
4. KIV: KIV (Karlsruhe Interactive Verifier), `http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/kiv/`
5. Kuncak, V., Nguyen, H.H., Rinard, M.: Deciding Boolean Algebra with Presburger Arithmetic. J. of Automated Reasoning (2006), `http://dx.doi.org/10.1007/s10817-006-9042-1`
6. Lugiez, D.: Multitree automata that count. Theor. Comput. Sci. 333(1-2), 225–263 (2005)
7. 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), `http://dx.doi.org/10.1007/978-3-540-78800-3_24`
8. Piskac, R., Kuncak, V.: Decision procedures for multisets with cardinality constraints. In: Logozzo, F., Peled, D.A., Zuck, L.D. (eds.) VMCAI 2008. LNCS, vol. 4905, pp. 218–232. Springer, Heidelberg (2008)
9. Piskac, R., Kuncak, V.: Linear arithmetic with stars. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 268–280. Springer, Heidelberg (2008)
10. Pottier, L.: Minimal solutions of linear diophantine systems: Bounds and algorithms. In: Book, R.V. (ed.) RTA 1991. LNCS, vol. 488. Springer, Heidelberg (1991)
11. Pugh, W.: A practical algorithm for exact array dependence analysis. ACM Commun. 35(8), 102–114 (1992)

A Slice-Based Decision Procedure for Type-Based Partial Orders

Elena Sherman, Brady J. Garvin, and Matthew B. Dwyer

Department of Computer Science and Engineering
University of Nebraska–Lincoln
Lincoln, NE 68588-0115
{esherman,bgarvin,dwyer}@cse.unl.edu

Abstract. Automated software verification and path-sensitive program analysis require the ability to distinguish executable program paths from those that are infeasible. To achieve this, program paths are encoded symbolically as a conjunction of constraints and submitted to an SMT solver; satisfiable path constraints are then analyzed further.

In this paper, we study *type*-related constraints that arise in path-sensitive analysis of object-oriented programs with forms of multiple inheritance. The dynamic type of a value is critical in determining program branching related to dynamic dispatch, type casting, and explicit type tests. We develop a custom decision procedure for queries in a theory of *type-based partial orders* and show that the procedure is sound and complete, has low complexity, and is amenable to integration into an SMT framework. We present an empirical evaluation that demonstrates the speed and robustness of our procedure relative to Z3.

1 Introduction

Recent years have witnessed an explosion in research on path-sensitive program analysis, e.g., [1–4]. These approaches have the potential to achieve significantly greater precision than more traditional program flow analyses. Increased precision is possible because the analyses are able to ignore program behavior on *infeasible paths*—sequences of program statements that are not executable in any run of the program—and thus more accurately reflect the program's semantics. Analyses avoid infeasible paths by calculating a symbolic characterization of constraints on input values that govern the execution of a path. This characterization is referred to as the *path condition*.

A path condition includes a constraint for each conditional branch in the program. For example, an integer constraint would be generated both for explicit branches like `if(x > 0){ ... }` and implicit branches such as those embedded in array bounds and divide-by-zero checks. Constraints over a variety of domains, and theories, are needed to encode path conditions for non-trivial programs. In addition to the theory of linear integer arithmetic, the theories of uninterpreted functions, extensional arrays and fixed-size bit vectors are commonly used [2, 3]. This diversity of constraints makes Satisfiability Modulo Theory (SMT) solvers, such as CVC3 [5] and Z3 [6], particularly well suited to reason about path-sensitive program behavior.

The theories supported by modern SMT solvers are somewhat limited, and mapping the data types in modern programming languages onto those theories can lead to inefficiency and imprecision. Consequently, there is significant interest in enriching the theories supported by SMT solvers to better match the needs of program analysis clients. For example, last year alone there were several papers reporting on decision procedures for theories of strings and on the reductions in cost and improvements in precision that arise from using those theories in reasoning about programs [7–9].

In this paper, we identify a fragment of the theory of partial orders, *type-based partial orders* (TPO), that is sufficient for reasoning about type constraints arising in object-oriented programs. In particular, it comprises the constraints due to dynamic dispatch, explicit subtyping tests (such as Java's `instanceof`), and typecasts. Furthermore, in contrast to the general theory of partial orders, under this smaller fragment we are able to adapt existing approaches for evaluating type tests at runtime to implement a standalone decision procedure for TPO, TPO-DP. It is amenable to inclusion in the DPLL(T) framework for SMT solvers [10] because it is incremental, restartable, and capable of calculating equalities for propagation as well as unsatisfiable cores.

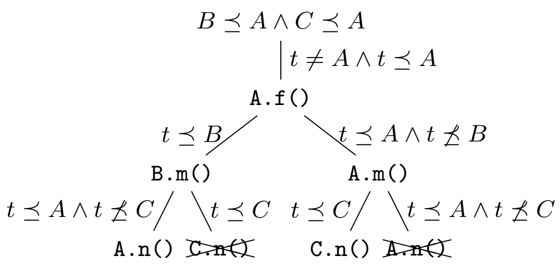
We have evaluated TPO-DP on a set of challenging benchmarks that are generated from a characterization of the type constraints encountered in path-sensitive program analyses. The results of the evaluation demonstrate that TPO-DP performs significantly better than Z3 using the theory of uninterpreted functions and an axiomatization of TPO.

In Sect. 2 we provide background on the nature of the type constraints that require support, discuss decision procedures that are capable of reasoning about partial orders, and describe existing approaches to efficiently evaluating type tests at runtime. We present a TPO-DP in Sect. 3; proofs of soundness, completeness and time and space complexity are included. An evaluation of TPO-DP and a discussion of our findings are presented in Sect. 4. Sect. 5 concludes with a discussion of several approaches that might be taken to further extend TPO-DP.

2 Background and Related Work

Figure 1 illustrates how type constraints arise when performing a path-sensitive analysis. Consider a modular analysis of method `A.f()` that begins on entry to the method. Initially, it can be inferred that the dynamic type of the implicit receiver object, `this`, is a subtype of `A`, but is not `A` itself because there can be no instances of an abstract class. The right side of the figure illustrates the constraints on the type of `this`, denoted t, that arise during the analysis; $t \preceq A$ means that t is a subtype of `A`. The root of the tree expresses the constraints arising from the definition of the type hierarchy. The edge from the root corresponds to the constraints on entry. At the call site to method `m()` there are two possibilities: either `A.m()` or `B.m()` is invoked depending on the type of `this`;

```
abstract class A {
  void m() {...};
  void n() {...};
  void f() { m(); n(); }
}
class B extends A {
  void m() {...}; }
class C extends A {
  void n() {...}; }
```



$B \preceq A \wedge C \preceq A$
$t \neq A \wedge t \preceq A$
A.f()
$t \preceq B$
$t \preceq A \wedge t \not\preceq B$
B.m()
A.m()
$t \preceq A \wedge t \not\preceq C$
$t \preceq C$
$t \preceq C$
$t \preceq A \wedge t \not\preceq C$
A.n()
~~C.n()~~
C.n()
~~A.n()~~

Fig. 1. Simple dynamic dispatch example (left) and type-related branching in symbolic execution tree (right)

the type constraints describing these situations label the second level of tree edges. Similarly at the call site to method `n()` the receiver type determines the invocation of `A.n()` or `C.n()`.

An analysis that does not consider type constraints must consider all four of the inter-procedural paths through `A.f()`. This can be costly in terms of both analysis time and precision since, for example, the analysis of the sequence `B.m()` followed by `C.n()` may produce results that are not reflective of executable program behavior. In this example, two of the four paths, marked with $\times$ in the figure, are infeasible and can be eliminated from analysis, thereby increasing both analysis speed and precision.

2.1 A Fragment of the Theory of Partial Orders

A type hierarchy is a partial order $(T, \preceq)$ where T is the set of types and $t_0 \preceq t_1$ holds if and only if t_0 is a subtype of t_1. The relevant semantics of the subtype relation are completely captured by the definition of a partial order: $\preceq$ is reflexive (every type is a subtype of itself), transitive (a subtype of a subtype is itself a subtype), and antisymmetric (mutual subtypes must be identical).

Let x be the dynamic type of a given value on a given path through the program. The domain of x is T. An analysis seeks to determine whether a path permits a constraint-satisfying assignment to x; otherwise the path must be infeasible.

Since most widely used object-oriented languages, such as Java and C++, do not support computing with types directly[1], constraints of the form $x \preceq y$, $x = y$, or $t_0 \preceq x$ (where y is another type variable) will not arise during analysis. TPO is the fragment of partial orders that is free of such constraints.

Within this fragment, we distinguish two classes of constraints. Constraints of the form $t_0 \preceq t_1$ and $t_0 \not\preceq t_1$ are used to encode the type hierarchy. These constraints are known ahead of time and do not directly arise during path-sensitive analyses. Constraints that do arise in path conditions may take on four forms: $x \neq t_0$ (e.g., no dynamic type is abstract), $x = t_0$ (e.g., a newly constructed

[1] Language support for type reflection can provide a form of type based computation, but we defer its treatment to future work.

object's type is known exactly), $x \preceq t_0$ (e.g., when successful dynamic type checks imply one of the object's supertypes), or $x \not\preceq t_0$ (e.g., when failed dynamic type checks eliminate a possible supertype). We refer to constraints with negations as *negative* and all others as *positive*.

When encoding a tree of TPO constraints, as in Fig. 1, we observe that the type hierarchy constraints can only appear in the root of the tree. This fact can be leveraged by incremental TPO decision procedures. The lack of constraints between type variables in TPO can also be leveraged since a TPO query can be decomposed into separate per-variable sub-queries that are each solved independently.

2.2 Deciding Partial Order Queries

Techniques for deciding partial order queries are readily available. There have been a rich body of work and also tool development related to the Bernays-Schönfinkel class of formulae, which subsumes partial order queries. Tools such as iProver [11] and Darwin [12] have been shown to be quite effective on such formulae. But in our setting, these tools have the disadvantage of not supporting incremental query checking, a common occurrence in path-sensitive analyses. While we could, in principle, compare TPO-DP to these tools, the comparison would be unfairly and severely biased in our favor. Therefore we chose a different approach for comparing our procedure to the state of the art.

Researchers have also explored support for the Bernays-Schönfinkel class of formulae in DPLL(T) solvers [13], but to the best of our knowledge no widely available solver implements those techniques. However, Z3 does offer heuristic quantifier instantiation, the underlying technique exploited by, for example, iProver. TPO queries can be encoded in Z3's theory of uninterpreted functions with an appropriate axiomatization of partial orders. Moreover, using Z3's support for the SMT-LIB Command language [14], complex TPO queries can be solved faster than on other tools we experimented with. In Sect. 4 we provide more detail on exactly how Z3 was configured for our evaluation.

2.3 Efficient Type Tests

As with runtime type tests, the key to an efficient TPO decision procedure is the encoding of the type hierarchy T. Two obvious alternatives are the transitive closure of $\preceq$ represented as a Boolean matrix and the transitive reduction of $\preceq$ as a linked structure. The former offers constant time subtyping tests, but it requires $\Theta(|T|^2)$ space and as much time to setup. Thus it can be costly on realistically sized hierarchies, e.g., Java 6 has 8865 classes in its standard library [15]. On the other hand, representing the transitive reduction as a linked structure needs only $\Theta(|T| + r)$ space, where r is the size of the transitive reduction, though subtype tests take $\Theta(h)$ time, where h is the height of the hierarchy.

There has been a significant body of research on finding novel encodings to improve this space/time trade-off, especially on the memory front [16–19]. But viewed as solving a satisfiability problem, these runtime tests never consider anything except a conjunction of two constraints in the pattern $(x = t_0) \wedge (x \preceq t_1)$ where t_0 is the object's known type, and t_1 is the type it is being checked against. As modular symbolic execution doesn't always have exact type, information we concentrated on encodings where longer conjunctions of constraints could be supported. This requirement led us to use the Type Slicing (TS) encoding [17] as the basis for TPO-DP.

TS constructs a compact representation of a type hierarchy by partitioning T and ordering the types within each partition. Let $T = \bigcup_{i=1\ldots k} \mathcal{T}_i$, where all $\mathcal{T}_i$ are disjoint; each individual $\mathcal{T}_i$ is called a *slice*, and the partitioning is termed a *slicing*. Further define $D_i(t)$ to be the descendants of t in slice i, namely $\{t' \mid t' \preceq t\} \cap \mathcal{T}_i$. Then we denote the ordered elements of a slice with square brackets, e.g., $[\mathcal{T}_i]$. Similarly, $[D_i(t)]$ designates the elements of $D_i(t)$ in the order given for $\mathcal{T}_i$. The essence of TS is the requirement that every $[D_i(t)]$ be a substring of the corresponding $\mathcal{T}_i$. In other words, the descendants of every type must be ordered contiguously in every slice. Once this property is established, determining whether one type t_0 is a subtype of another, t_1, is a two-step process. First we must locate t_0 in the slicing. Then we must compare its position to the bounds of the interval occupied by $[D_i(t_1)]$ in the same slice. The operation is constant time.

The TS encoding uses two integers per type to store that location information: one index indicating which slice it occupies and another giving its position in that slice's order. Additionally, for every type/slice pair there must be an entry to track the upper and lower bounds of $[D_i(t)]$. Hence, the space complexity is in $\Theta(k\,|T|)$ where, recall, k is the number of slices. As we show later in Sect. 3.2, for TPO-DP we also want to minimize k.

Algorithms for constructing a minimal number of slices are exponential, but greedy algorithms have proven effective on real type hierarchies. They operate by building a slice by repeatedly adding a single type. That type is added by attempting to insert it into each existing slice in succession. If insertion into an existing slice is not possible without violating the contiguity requirement, a new slice is created with t as the sole member.

Even though in the worst case $k \in \Theta(|T|)$, the number of slices k is usually very small compared to $|T|$. For instance the Java 6 hierarchy of `java.*` and `javax.*` packages contains 5,632 types which can be partitioned into 12 slices. Table 3 in [19] provides more k values for different Java releases using different slicing encodings. That data confirms that k is at least two orders of magnitude smaller than $|T|$ in practice. While the TS encoding is not in that table, we found it better in practice than the ESE encoding that is mentioned.

We illustrate the TS encoding and type test evaluation approach on the simple type hierarchy from Fig. 1. Assume that A, B and C types are divided (sub-optimally) into 2 slices: $[A, B]$ and $[C]$. According to TS, the encoding is $A = (1, 1, ([1\ldots2], [1\ldots1]))$, $B = (1, 2, ([2\ldots2], []))$ and $C = (2, 1, ([], [1\ldots1]))$. The first element of each tuple is the slice the type belongs to, the second is the type's position in that slice's order, and the third is the per-slice descendant intervals. To evaluate $C \preceq A$, we first determine that C lies in slice 2 at index

1. A's descendant interval for slice 2 is $[1\ldots1]$, which includes C's position. So $C \preceq A$ holds. Checking $C \preceq B$, C's index cannot lie in the second descendant interval of B because that interval is empty; $C \not\preceq B$.

3 Decision Procedure for Type Partial Orders (TPO-DP)

We determine the set of assignments that satisfy a conjunction of literals by computing the set of assignments that satisfy each literal on its own and then taking the intersection of these sets. A formula is satisfiable if this intersection is nonempty.

Under the slicing used in TS, a set of assignments can be expressed as the union of a set of intervals in the slices' orderings. Consider the example encoding at the end of the previous section. The constraint $x \preceq A$ would have its assignments encoded as $\{[1\ldots2]\}$ for the first slice and $\{[1\ldots1]\}$ for the second. Because of the contiguity of descendants (the contiguity of equal types is trivial), assignments for a positive constraint will form at most one interval in each slice. Similarly, negative constraints are satisfied by the complement set of assignments, which can be expressed in at most two intervals per slice. In contrast the general theory of partial orders affords no such guarantees.

To support pushing and popping constraints, we keep an explicit stack of these sets for each slice. Take, for instance, the same example encoding and two operations: pushing $x \neq A$ and $x \not\preceq C$. The first would push the interval set $\{[2\ldots2]\}$ on the first slice's stack and $\{[1\ldots1]\}$ on the second's. $x \not\preceq C$ would be encoded as $\{[1\ldots2]\}$ and $\varnothing$, so the intersections of the respective interval unions would be $\{[2\ldots2]\}$ and $\varnothing$; these sets become the new tops of the stacks when $x \not\preceq C$ is pushed.

3.1 Soundness and Completeness

Because we are computing intersections of sets of satisfying assignments, soundness and completeness depend on the correctness of the TS slicing algorithm.

Unfortunately, the TS paper does not contain a proof of correctness. However correctness follows directly from two facts. First, the algorithm guards every insertion of a type into a slice with explicit checks for the contiguity of descendants. Therefore, the intervals resulting from the TS slicing do not contain any assignments which are not satisfying. Second, every element is inserted into some slice; as a last resort a new slice can always be allocated. Hence, as all satisfying assignments are in T, every one of them lies in some interval.

In [20], we review the algorithm from Appendix A of [17] and present this argument in more detail.

3.2 Time and Space Complexity

Because TPO-DP uses the TS encoding of T we separate the description of time and space complexity into the TS slicing complexity and the decision procedure complexity.

The time complexity of greedily creating slices is determined by the cost of trying to insert each type into a slice, multiplied by the number of types $|T|$ and the number of slices k. The TS insertion attempts take $O(|A(t)|)$ time where $A(t)$ designates the type's ancestors [17]. So the worst case for preprocessing is certainly in $O(k\,|T|^2)$. Note that this is only an upper bound—we have not developed a tighter bound. Also recall that k grows very slowly with respect to $|T|$, as discussed in Sect. 2.3.

The space complexity is dominated by the encoding at $\Theta(k\,|T|)$.

As for the online decision procedure, time and space complexity are closely coupled. We proceed by considering two cases: the cost per slice of pushing a positive constraint and the same cost for pushing a negative one.

For a positive constraint we must take a union of intervals already on the stack and retain only the elements within the interval that correspond to the new constraint. In processing each interval we test whether it should be kept, narrowed, or discarded, so the operation is in the worst case linear in the number of intervals in the union. Note that positive constraints have only one interval per slice, and it is impossible for the push to divide a single interval into many.

Similarly, when a negative constraint is added, each interval in the slice's union may be kept, narrowed, or discarded, but it is also possible to split one of the intervals into two if it encloses the constraint-violating assignments. Thus, the complexity of adding a negative constraint is the same as for a positive constraint—linear in the number of intervals in the union—but in the worst case it may increase this count by one for the next push. The growth in the number of intervals across all slices is bounded by $|T|/2$ though; the maximum is reached when the types in every slice are alternately included and excluded, and all slices have even length.

So let c_p be the number of positive constraints and c_n the number that are negative. In the worst case, the negative constraints come first, amassing as many as $\sum_{i=1}^{c_n} k(i+1)$ intervals on the stack when $k(c_n+1) \leq |T|/2$ and the $|T|/2$ bound is not encountered. If the bound is struck, at the b^{th} constraint, the number of intervals is $\sum_{i=1}^{b} k(i+1)$ for the first b pushes and $\sum_{i=b+1}^{c_n} k(b+1)$ for those that come later. After that, the worst that the positive constraints can do is eliminate no intervals from the tops of the stacks. There will be an additional $\sum_{i=1}^{c_p} k(c_n+1)$ intervals in the first case and $\sum_{i=1}^{c_p} k(b+1)$ in the second.

In total, the first case creates $\Theta(kc_p + kc_nc_p + kc_n^2)$ intervals and the latter builds $\Theta(c_p\,|T| + c_n\,|T| - |T|^2)$. Each interval is processed exactly once, so these expressions give the time complexity. It also follows that the worst-case memory complexity is the sum of these counts and the $\Theta(k\,|T|)$ space taken by the slicing.

Designating the total number of constraints as c, we observe that c_n controls where the complexity of the algorithm lies between $\Theta(kc)$ and $\Theta(|T|\,c)$. Therefore we believe it may be fruitful to explore transformations that eliminate negative constraints. We propose one such transformation in Sect. 5.

3.3 Incrementality, Restartability, and Unsatisfiable Cores

For a decision procedure to be incorporated into the DPLL(T) framework it should be incremental, restartable, able to propagate equalities, and able to generate explanations for UNSAT cases, i.e. unsatisfiable cores. The TPO-DP meets each of these requirements.

TPO-DP is inherently incremental thanks to the explicit stack. Restarting is also straightforward: the procedure is reset by clearing its stack. For propagating equalities we must merely check after each push if there is exactly one possible assignment.

We have developed two approaches for calculating unsatisfiable cores. Each approach has its own advantages in terms of complexity and core size. However neither approach is completely incremental in nature or guaranteed to produce minimum cores. In the paragraph below we present the simpler of the two. Another method and the discussion of tradeoffs are presented in our tech report [20].

Let u be the first constraint on variable x to make the top of every slice's stack empty. Then the constraints up through u constitute an (likely large) unsatisfiable core U. All sub-cores of U must contain u, because the formula was SAT before u was pushed. Furthermore, if we define a sub-domain T' of T that includes exactly those assignments to x that satisfy u, $U \setminus \{u\}$ is UNSAT on T'. Therefore, we mark u and push it onto a new solver instance (effectively limiting the domain to T'), followed by the constraints that preceded it in order, until we obtain u' in the same manner that we obtained u. If u' is unmarked, as it will likely be the first time, we can repeat the procedure with a chance to eliminate more constraints from the core. Otherwise, we terminate, returning the set of constraints pushed on the last solver instance.

The complexity of finding unsatisfiable cores by this process is the cost of checking the formula at most c times, c being the total number of contraints as defined in Sect. 3.2.

3.4 Implementation Choices

We implemented the TPO-DP in Java without any significant effort to optimize its performance. To perform a fair comparison with Z3, we implemented a parser for SMT-LIB command syntax that prepares inputs for our decision procedure; this allows our DP and Z3 to use exactly the same input.

After initial evaluation, we identified two optimizations that yield a performance benefit. First, when multiple type variables are present, we produce independent instances of the TPO solver for each variable; a problem is SAT if and only if it is SAT in each solver instance. Second, we cache the results of satisfiability check which allows us to avoid processing redundant queries. Such queries can be common in path-sensitive analysis of real programs. For instance, [21] reports on an analysis that caches queries outside of the decision procedure and observes hit-rates above 80%.

4 Evaluation

Our primary research question centers on performance: *How does the performance of the TPO-DP compare to state-of-the-art solvers on queries arising in path-sensitive program analyses?* We begin with an assessment of path-sensitive analysis techniques and their use of decision procedures.

4.1 Categories of Path-Sensitive Analyses

A path-sensitive analysis generates a set of closely related queries where longer queries are extended from shorter ones by conjoining additional clauses. There are multiple ways to extend a query, e.g., producing two queries where one conjoins a constraint for a branch predicate and another that conjoins its negation. Such an analysis gives rise to a *query tree* where nodes correspond to calls to check for satisfiability and edges in the tree encode the assertion of clause sets.

Path-sensitive analyses are expensive to apply to real programs. Consequently, *intra-procedural* path-sensitive analyses, i.e., analyses limited to individual methods, were the first to be explored by researchers. It is natural that papers on these techniques report results on leaf methods – methods that do not call other methods. For example, red-black tree implementations can be analyzed to detect complex corner cases in their logic [2]; when tree height is no greater than six such an analysis generates a tree with a few thousand queries. In intra-procedural analysis, dynamic dispatch is not an issue, since calls from the method will not be analyzed, so there is, arguably, little need for a TPO-DP.

Recently, researchers have begun to develop *inter-procedural* path-sensitive analyses and apply them to larger portions of programs. This strategy has been applied, for example, to produce crash-inducing inputs [22] and to detect security faults [21]. Such analyses consider larger portions of programs whose behavior involves many method calls and, consequently, there is a need to reason about non-trivial type constraints related to dynamic dispatch. The authors of [21] applied their technique to 3 large programs that required from 22 to 188 thousand queries. We note that these query counts were taken after significant optimization of query checks were performed, e.g., caching of query results outside of the decision procedure.

Spurred by advances in automated decision procedures research in path-sensitive program analysis appears to be accelerating. We conjecture that the *next-generation* of path-sensitive analyses will continue to scale to larger portions of program behavior.

4.2 A Population of TPO Queries

Since we do not have access to a next-generation path-sensitive analysis, we performed a pilot study to characterize the size and diversity of type queries across a set of three open-source Java programs of varying size and complexity. For each program, we instrumented its implementation to record the total number of branches taken and, for each object, the sequence of type related branches

Table 1. Type Constraint-related Observations from Program Trace Data

Program	$\|T\|$	Ex.	k	#Trace	#Obj.	Len.	DD	CC	IN	Type
NanoXML[23]	79	106	3	5	366	41.0	68k	5.5k	0	41%
Weka[24]	611	893	6	14	2.8k	35.5	134k	1.2M	383	36%
Soot[25]	3259	4019	7	1	32k	191.7	458k	1.6M	4.1M	69%

evaluated and the nature of the predicate being tested. The instrumented programs were executed on a subset of their test suites, and the recorded data were analyzed to produce the summary in Table 1.

We measured the number of classes and interfaces used in each program including all of the application classes' super-types and super-interfaces. This number is shown under the second ($|T|$) column in Table 1. The third column (Ex.) corresponds to the number of subtype, i.e., `extends` declarations, or interface implementation declarations, i.e., `implements`. The number of type slices for each program (k) illustrate the significant compression achieved by the TS encoding. The recorded program traces (#Trace) are partitioned into sub-traces for each object allocated in the program. (#Obj.) reports the average number of objects in a trace that are involved in type constraints and (Len.) the average length of the per-object sub-traces. Across all of the traces for a program we recorded the number of branches related to dynamic dispatch (DD), class casts (CC), and instanceof tests (IN). Finally, we report the percentage of all branches across the traces that involved type tests (Type); these data clearly indicate the need to support TPO constraints. We also note that gathering fine-grain trace data is extremely expensive, and to reduce the cost of our study we ran Soot just once and only collected information for objects of Soot-defined type. Even then the data collection took many hours.

To characterize the per-variable constraints, we processed the per-object trace data to discard type tests that occurred in the same method where the object was instantiated, since they could be decided trivially with the equality constraints introduced by object allocation. In this way we retained only the tests that modular path-sensitive analysis might see.

From those raw traces we built a summary in the form of a Markov model where each state corresponded to exactly one type test. The training could then proceed deterministically by frequency counting. We used no prior distribution.

Finally, we built the benchmarks directly from the Markov models for each program. For each variable our generator kept a stack of states. Then, to push constraints, it chose a variable randomly and treated its state stack as the prefix of a path in the Markov model; the next state was chosen according to the transition probabilities from the topmost state. This state and one of the corresponding sets of constraints were pushed. As for popping, the generator removed the most recently pushed state, along with its constraints. Moreover, to better mimic the constraints generated by a path-sensitive analysis, if a popped state still had other alternatives to explore, one of these was subsequently pushed.

The generator can be parameterized by the number of objects involved in a trace and the depth of the query tree, thereby allowing us to produce a population of TPO query trees that resemble the three programs but are scaled in several dimensions. Query trees are emitted in the SMT-LIB command format using push/pop to encode the query tree edges[2].

4.3 Comparing to a State-of-the-Art SMT Solver

We selected Z3 as a point of comparison since it is known to be efficient, supports incremental solving, and can answer TPO queries by instantiating quantifiers in the partial order axioms. Initially our use of Z3 for checking TPO queries resulted in poor performance. It is widely understood that the appropriate selection of solver heuristics is crucial to using a tool like Z3 effectively. To be as fair as possible in our evaluation we contacted the developers of Z3 asking for their advice on configuring the tool and modifying the input files to maximize performance. The developers were very supportive and supplied us with a modified input file that was better suited for Z3's quantifier instantiation techniques [26]. They also suggested that for our queries we should run Z3 with `AUTO_CONFIG=false` to disable heuristics that were unnecessary for TPO problems and, in fact, were hurting performance significantly. Finally, they informed us that for TPO problems Z3 is "effectively" sound and complete, thus when the solver returns an UNKNOWN result it can be interpreted as SAT; the results of our evaluation confirmed this to be the case.

4.4 Results

We decomposed the population of TPO query trees into three groupings based on the number of queries in the tree. The data from [21] combined with the fact that between 36% and 69% of the queries in our pilot studies were related to type tests led us to the following breakdown. Inter-procedural (Inter) analyses perform between a few thousand up to several tens-of-thousands of type-related queries; for our TPO data we selected 4096 as the lower and 65535 as the upper bound for defining this category. Queries trees that were smaller were classified as those that intra-procedural (Intra) analyses could produce, and those that were larger we consider the province of next-generation analyses.

Table 2 reports performance data of TPO-DP and Z3 on the population of TPO queries broken down by grouping. The number of query trees (Num. Trees) and the average number of queries (Avg. Size) in a tree are listed for each grouping. In addition, we report the average number of SAT and UNSAT problems per tree; TPO-DP and Z3 agreed on this in every case—when Z3's UNKNOWN is interpreted as SAT. We ran all jobs under Linux on a 2.4GHz Opteron 250 with 4 Gigabytes of RAM. With a timeout of four hours, Z3 failed to complete some of the larger query trees, and we give the number aborted (Z3 TO); TPO-DP

[2] The data and benchmarks used in this study are available at `http://esquared.unl.edu/wikka.php?wakka=TpoDp`

Table 2. Solver performance data across size-based problem categories

Category	Num. Trees	Avg. Size	Avg. SAT	Avg. UNSAT	TPO-DP all	TPO-DP non-TO	Z3 non-TO	Z3 TO
Intra	388	857.6	535.4	322.1	10.2	2.8	464.1	119
Inter	46	22359.1	11665.4	10693.7	12.3	2.4	2233.1	21
Next-Gen.	16	124576.7	64668.6	59908.1	21.1	3.2	831.6	10

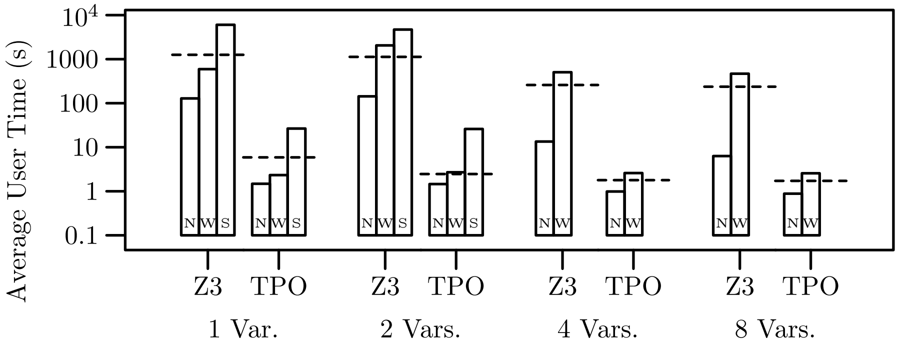


Fig. 2. Z3 and TPO User Times versus Variable Count

never took more than 72 seconds and completed all problems. We provide the average time to solve a TPO query tree across each category for TPO-DP in seconds (TPO-DP all). For the problems on which Z3 completed, we report the average TPO query tree solve times for TPO-DP (TPO-DP non-TO) and for Z3 (Z3 non-TO) in seconds of user time.

We conjectured that other factors might influence the relative effectiveness of TPO-DP and Z3. Specifically, the diversity in programs that informed our generation strategy was significant and had a non-trivial influence on the degree of branching in the query tree. In addition the number of free variables in the constraints and the depth of the query tree (i.e. the number of the conjuncts) could cause the two techniques to perform differently. Figures 2 and 3 present log-scale plots of the cumulative cost of running the queries while varying the two parameters of our generator. This time we break the data down by program, where N, W, and S stand for nanoXML, Weka, and Soot, respectively. In addition, we indicate the mean run times with the dashed line.

Note that these plots only reflect benchmarks solved by both implementations —the problems which Z3 could complete in four hours. In particular all of the Soot benchmarks with more than two variables timed out, and the bar for Soot at two variables represents a single data point. Similarly, at the depths of 10 and 15 only one Z3 run with the Soot hierarchy completed. In contrast, the data for the nanoXML and Weka benchmarks was not significantly hampered by timeouts.

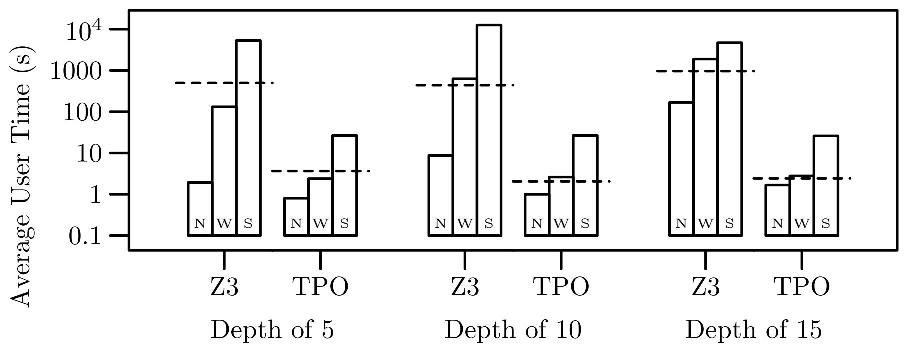


Fig. 3. Z3 and TPO User Times versus Stack Depth

4.5 Discussion

We believe that these results strongly suggest that in advanced path-sensitive program analyses there is a need for custom decision procedures for TPO queries. Such queries arise frequently, and when they do, even state-of-the-art solvers such as Z3 struggle to scale to the size of problems that are characteristic of advanced analyses.

Across all of the different decompositions of the data we collected, TPO-DP outperformed Z3 by a wide margin. Varying the program, the number of variables, or tree depth seemed to have only a modest effect on TPO-DP's performance, except when the number of variables grows large. We believe this latter observation to be more a property of the generated problems, because when the number of variables grows large, for a fixed depth of query tree, the number of constraints that simultaneously involve a single variable will likely decrease. TPO-DP appears to scale well with query tree size, it is relatively stable when moving from the intra-procedural to inter-procedural categories, a 26-fold increase in size on average. This appears to reflect a true benefit of TPO-DP, since Z3's run time increases almost 5-fold across those same categories. We believe that the performance of TPO-DP in moving from the inter-procedural to next-generation categories also suggests good scalability, a 5.6-fold increase in size gives rise to only a 70% increase in solver time. The analogous data for Z3 is not informative since it times out on 10 of the 16 problems.

5 Conclusions and Future Work

As path-sensitive analyses scale to consider larger portions of object-oriented programs they will invariably encounter large numbers of type-related constraints. Existing methods for solving those constraints are not well-integrated with SMT solvers that support the integer, array, string, and bit-vector theories needed for program reasoning. We developed TPO-DP—a custom decision procedure that leverages results from efficient language runtime systems to efficiently process constraints related to type-based partial orders. We designed and conducted a significant evaluation producing problems representative of those that would be generated by path-sensitive analyses and took care in conducting this evaluation that we did not bias the results in favor of our tool. In this context, TPO-DP outperformed Z3 by a wide margin.

In future work, we plan to explore additional optimizations to our method that will compute type slices for commonly used libraries ahead of time. In addition we will investigate hierarchy transformations that will allow us to introduce a concreteness pseudo-type and thereby convert the many negative constraints for abstract classes into one positive subtype constraint. Our fine-grained complexity characterization suggests that this may further reduce solver time. Finally, we have been approached by the developers of Kiasan [2] who wish to integrate TPO-DP into their analysis engine to study its effectiveness on a variety of analysis problems. In support of that work, we plan to explore extensions of TPO and TPO-DP that support dynamic loading of types and type reflection.

Acknowledgements

The authors would like to thank Robby for bringing the need for a TPO decision procedure to our attention and Xiao Qu for working on a early prototype. Cesare Tinelli, Aaron Stump, and Leonardo de Moura provided helpful comments on an early version of this work and suggestions for developing it further, and we thank them. We could not have conducted a fair comparative evaluation without the help of Nikolaj Bjørner, and we thank him for sharing his insights with us.

This work was supported in part by the National Science Foundation through awards CCF-0541263 and CCF-0747009, the National Aeronautics and Space Administration under grant number NNX08AV20A, the Air Force Office of Scientific Research through award FA9550-09-1-0129, and the Army Research Office through DURIP award W91NF-04-1-0104.

References

1. Păsăreanu, C.S., Visser, W.: A survey of new trends in symbolic execution for software testing and analysis. STTT 11, 339–353 (2009)
2. Deng, X., Lee, J., Robby: Bogor/Kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems. In: Proceedings of ASE, pp. 157–166 (2006)
3. Anand, S., Godefroid, P., Tillmann, N.: Demand-driven compositional symbolic execution. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 367–381. Springer, Heidelberg (2008)
4. Godefroid, P., de Halleux, J., Nori, A.V., Rajamani, S.K., Schulte, W., Tillmann, N., Levin, M.Y.: Automating software testing using program analysis. IEEE Software 25, 30–37 (2008)
5. Barrett, C., Tinelli, C.: CVC3. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 298–302. Springer, Heidelberg (2007)
6. 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)

7. Bjørner, N., Tillmann, N., Voronkov, A.: Path feasibility analysis for string-manipulating programs. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 307–321. Springer, Heidelberg (2009)
8. Yu, F., Bultan, T., Ibarra, O.H.: Symbolic string verification: Combining string analysis and size analysis. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 322–336. Springer, Heidelberg (2009)
9. Hooimeijer, P., Weimer, W.: A decision procedure for subset constraints over regular languages. In: Proceedings of PLDI, pp. 188–198 (2009)
10. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). J. ACM 53, 937–977 (2006)
11. Korovin, K.: iProver – an instantiation-based theorem prover for first-order logic (system description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 292–298. Springer, Heidelberg (2008)
12. Baumgartner, P., Tinelli, C.: The model evolution calculus as a first-order DPLL method. Artif. Intell. 172, 591–632 (2008)
13. de Moura, L., Bjørner, N.: Deciding effectively propositional logic using DPLL and substitution sets. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 410–425. Springer, Heidelberg (2008)
14. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: Version 2.0. Technical Report BarST-RR-10, Department of Computer Science, The University of Iowa (2010), http://www.SMT-LIB.org
15. Java: Package java.lang, Java^{TM}platform standard ed. 6, http://java.sun.com/javase/6/docs/api/java/lang/package-summary.html
16. Zibin, Y., Gil, J.Y.: Efficient subtyping tests with PQ-encoding. In: Proceedings of OOPSLA, pp. 96–107 (2001)
17. Zibin, Y., Gil, J.Y.: Fast algorithm for creating space efficient dispatching tables with application to multi-dispatching. In: Proceedings of OOPSLA, pp. 142–160 (2002)
18. Baehni, S., Barreto, J., Eugster, P., Guerraoui, R.: Efficient distributed subtyping tests. In: Proceedings of DEBS, pp. 214–225 (2007)
19. Alavi, H.S., Gilbert, S., Guerraoui, R.: Extensible encoding of type hierarchies. In: Proceedings of POPL, pp. 349–358 (2008)
20. Sherman, E., Garvin, B.J., Dwyer, M.B.: A slice-based decision procedure for type-based partial orders. Technical Report TR-UNL-CSE-2010-0004, University of Nebraska–Lincoln, Lincoln, NE 68588-0115 (2010)
21. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: Exe: Automatically generating inputs of death. ACM Trans. Inf. Syst. Secur. 12 (2008)
22. Cadar, C., Dunbar, D., Engler, D.R.: Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of z, pp. 209–224 (2008)
23. SIR: Software-artifact infrastructure repository, http://sir.unl.edu
24. Weka: Machine learning software, http://sourceforge.net/projects/weka
25. Soot: a java optimization framework, http://www.sable.mcgill.ca/soot/
26. Bjørner, N.: Personal Communication (2009)

Hierarchical Reasoning for the Verification of Parametric Systems

Viorica Sofronie-Stokkermans

Max-Planck-Institut für Informatik, Campus E 1.4, D-66123 Saarbrücken, Germany
sofronie@mpi-sb.mpg.de

Abstract. We study certain classes of verification problems for parametric reactive and hybrid systems, and identify the types of logical theories which can be used for modeling such systems and the reasoning tasks which need to be solved in this context. We identify properties of the underlying theories which ensure that these classes of verification problems can be solved efficiently, give examples of theories with the desired properties, and illustrate the methods we use on several examples.

1 Introduction

Most of the applications in verification require reasoning about complex domains. In this paper we identify several classes of verification problems for parametric reactive systems (modeled by transition constraints) and for simple hybrid systems, and point out the reasoning tasks in the associated theories which need to be solved. The type of parametricity we consider refers to parametric data (including parametric change and environment) specified using functions with certain properties and parametric topology, specified using data structures.

The first problem we address is to check whether a safety property – expressed by a suitable formula – is an invariant, or holds for paths of bounded length, *for given instances of the parameters*, or *under given constraints on parameters*. For this type of problems, we aim at identifying situations in which decision procedures exist. We show that this is often the case, by investigating consequences of locality phenomena in verification. If unsafety is detected, the method we use allows us to generate counterexamples to safety, i.e. concrete system descriptions which satisfy all the imposed constraints and are unsafe.

We also analyze the dual problem – related to system synthesis – of *deriving constraints between parameters* which guarantee that a certain safety property is an invariant of the system or holds for paths of bounded length. Such problems were studied before for the case when the parameters are constants [1,6,20,15,13]. We present a new approach which can be used also in the case when some of the parameters are allowed to be functional and show that sound and complete hierarchical reduction for SMT checking in local extensions allows to reduce the problem of checking that certain formulae are invariants to testing the satisfiability of certain formulae w.r.t. a standard theory. Quantifier elimination is used for generating constraints on the parameters of the system (be they data or functions) which guarantee safety. These constraints on the parameters may also be used to solve optimization problems (maximize/minimize some of the parameters) such that safety is guaranteed. If we also express invariants in a parametric form, this method can also be used for identifying conditions which guarantee that formulae with a certain shape are invariants, and ultimately for generating invariants with a certain shape. There exist approaches to the verification of parametric reactive infinite state systems and timed automata (e.g. by Ghilardi et al. [8], Hune et al. [10], Cimatti et al. [3]) and for parametric hybrid automata (e.g. by Henzinger et al. [1], Frehse, [6], Wang [20], and Cimatti et al. [4]), but in most cases only situations in which the parameters are constants were considered. The idea of using hierarchical reasoning and quantifier elimination for obtaining constraints on the parameters (constants or functions) is new.

Structure of the paper. The paper is structured as follows: In Section 2 we present existing results on local theory extensions which allow us to identify decidable theories interesting in verification. In Section 3 we identify situations in which decision procedures exist for invariant checking and bounded model checking, as well as methods for obtaining constraints between the parameters which guarantee that certain properties are invariants. We consider both systems modeled using transition constraints and a similar model of hybrid systems. In Section 4 we draw conclusions and present plans for future work.

1.1 Idea and Running Examples

We illustrate the ideas on the following examples:

Example 1. Consider a discrete water level controller in which the inflow (in) in the interval of time for one step in the evolution of the system is fixed. If the water level becomes greater than an alarm level L_{alarm} (below the overflow level L_{overflow}) a valve is opened and a fixed quantity of water (out) is left out. Otherwise, the valve remains closed. We want to check whether, assuming that we start from a state in which the water level L satisfies $L \leq L_{\mathsf{overflow}}$, the water level always remains below L_{overflow}. Let $\mathcal{T}_S$ be $\mathbb{R}$, the theory of real numbers.

$$\begin{tikzcd}
L \geq L_{\mathsf{alarm}} \arrow[loop left, "L':=L+\mathsf{in}-\mathsf{out}"] \arrow[r, bend right, "L':=L+\mathsf{in}-\mathsf{out}"'] & L < L_{\mathsf{alarm}} \arrow[loop right, "L':=L+\mathsf{in}"] \arrow[l, bend right, "L':=L+\mathsf{in}"']
\end{tikzcd}$$

Assume that a set Γ of constraints on the parameters is given, e.g. $\Gamma = \{\mathsf{in} = \mathsf{out}-10, \mathsf{in} = L_{\mathsf{overflow}} - L_{\mathsf{alarm}} - 10\}$.

Then $L \leq L_{\mathsf{overflow}}$ is an invariant iff formulae (i), (ii) are unsatisfiable w.r.t. $\mathcal{T}_S \cup \Gamma$.

(i) $\exists L, L'(L_{\mathsf{overflow}} \geq L \geq L_{\mathsf{alarm}} \wedge L' = L + \mathsf{in} - \mathsf{out} \wedge L' > L_{\mathsf{overflow}})$.
(ii) $\exists L, L'(L < L_{\mathsf{alarm}} \wedge L' = L + \mathsf{in} \wedge L' > L_{\mathsf{overflow}})$.

It is easy to check that formulae (i) and (ii) above are unsatisfiable w.r.t. $\mathcal{T}_S \cup \Gamma$ by using a decision procedure for the theory of real numbers.

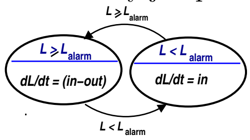
Assume now that we do not a priori impose any constraints Γ on the parameters of the systems. We still know that the safety condition is an invariant iff the formulae in (i), (ii) are unsatisfiable w.r.t. $\mathcal{T}_S$. We can eliminate the existentially quantified variables L, L' using a method for quantifier elimination in $\mathbb{R}$ and thus show that (assuming that $L_{\mathsf{alarm}} < L_{\mathsf{overflow}}$) the formula in (i) is equivalent to $(\mathsf{in} > \mathsf{out})$ and the formula in (ii) is equivalent to $(\mathsf{in} > L_{\mathsf{overflow}} - L_{\mathsf{alarm}})$. We can therefore conclude (under the assumption that $L_{\mathsf{alarm}} < L_{\mathsf{overflow}}$) that $L > L_{\mathsf{overflow}}$ is an invariant iff $(\mathsf{in} \leq \mathsf{out}) \ \wedge \ (\mathsf{in} \leq L_{\mathsf{overflow}} - L_{\mathsf{alarm}})$.

Example 2. Consider a variant of Example 1 in which the inflow varies in time. In all transitions, we will therefore replace in by $\mathsf{in}(t)$ and add the time change $t' = t+1$. Assume that we describe the initial states using the formula $\mathsf{Init}(L) := L_a \leq L \leq L_b$, where L_a, L_b are parameters with $L_a < L_b$. Then $L \leq L_{\mathsf{overflow}}$ is an invariant iff the following formulae are unsatisfiable w.r.t. $\mathcal{T}_S = \mathbb{R} \cup \mathbb{Z}$ (the many-sorted combination of the theory of reals and integers (for modeling time)):

(1) $\exists L (L_a \leq L \leq L_b \wedge L \geq L_{\mathsf{overflow}}) \models_{\mathcal{T}_S} \perp$.
(2) Safety is invariant under transitions:
  (i) $\exists L, L', t, t' (L_{\mathsf{overflow}} \geq L \geq L_{\mathsf{alarm}} \wedge L' = L + \mathsf{in}(t) - \mathsf{out} \wedge t' = t+1 \wedge L' > L_{\mathsf{overflow}})$.
  (ii) $\exists L, L', t, t' (L < L_{\mathsf{alarm}} \wedge L' = L + \mathsf{in}(t) \wedge t' = t+1 \wedge L' > L_{\mathsf{overflow}})$.

Under the assumption that $L_a < L_b$ we can prove (using quantifier elimination in the theory of reals [19]) that (1) iff $L_b < L_{\mathsf{overflow}}$. It is not immediately clear how to eliminate the quantifiers in the formulae in (2)(i) and (2)(ii) because of the occurrences of the function in. In this paper we identify situations in which the satisfiability problems can be reduced, in a sound and complete way, to satisfiability problems over the base theory, by using locality properties of these theories. Locality allows us to perform a reduction to satisfiability checks w.r.t. $\mathbb{R} \cup \mathbb{Z}$, where we can eliminate all quantified variables except for the parameters and the variables which stand for arguments of the parametric functions; we then interpret the result back in the theory extension. This way we proved that (2)(i) iff $\forall t (\mathsf{in}(t) - \mathsf{out} \leq 0)$, and (2)(ii) iff $\forall t (\mathsf{in}(t) \leq L_{\mathsf{overflow}} - L_{\mathsf{alarm}})$.

Example 3. We can also model the water tank controller as a hybrid system, with two discrete states (state invariants $L \geq L_{\mathsf{alarm}}$ and $L < L_{\mathsf{alarm}}$); and changes described by jumps between these states and flows within each state.



We model inflow and outflow by functions $\mathsf{infl}, \mathsf{outfl}$, where $\mathsf{infl}(t)$ ($\mathsf{outfl}(t)$) is the inflow (resp. outflow) in time t. Assume that the inflow and outflow rates are constant and equal to in, resp. out. The problems we consider are:

(1) Check whether the safety condition $\Psi = L < L_{\mathsf{overflow}}$ is invariant (under jumps and flows), assuming that $\mathsf{in}, \mathsf{out}$ satisfy certain given properties.
(2) Generate conditions on the parameters which guarantee that Ψ is invariant.

$L < L_{\mathsf{overflow}}$ is invariant under flows iff the following formulae are unsatisfiable:

(i). $\exists L, t (L < L_{\mathsf{alarm}} \wedge \forall t' (0 \leq t' \leq t \rightarrow L + \mathsf{in} * t' < L_{\mathsf{alarm}}) \wedge L + \mathsf{in} * t > L_{\mathsf{overflow}})$.
(ii) $\exists L, t (L \geq L_{\mathsf{alarm}} \wedge \forall t' (0 \leq t' \leq t \rightarrow L + (\mathsf{in} - \mathsf{out}) * t' \geq L_{\mathsf{alarm}}) \wedge L + (\mathsf{in} - \mathsf{out}) * t > L_{\mathsf{overflow}})$.

These are formulae with alternations of quantifiers. Task (1) can be solved using a decision procedure for the satisfiability of the $\exists\forall$ fragment of the theory of reals, task (2) uses quantifier elimination. In Section 3.2 we will present in detail this situation and also the case when the evolution rules in a state are specified by giving bounds on the rate of growth of the continuous variables.

2 Decision Problems in Complex Theories

In this section we analyze a class of theories used for modeling reactive, real time and hybrid systems for which we can obtain decidability results.

Theories, theory extensions. First-order theories are sets of formulae (closed under logical consequence), typically all consequences of a set of axioms. (Alternatively, we may consider a set of models which defines a theory.) Let $\mathcal{T}_0$ be a theory with signature $\Pi_0 = (S, \Sigma_0, \mathsf{Pred})$, where S is a set of sorts, Σ_0 a set of function symbols, and Pred a set of predicate symbols. We consider extensions $\mathcal{T}_1 = \mathcal{T}_0 \cup \mathcal{K}$ of $\mathcal{T}_0$ with signature $\Pi = (S, \Sigma, \mathsf{Pred})$, where $\Sigma = \Sigma_0 \cup \Sigma_1$ (i.e. the signature is extended by new function symbols Σ_1 whose properties are axiomatized by a set $\mathcal{K}$ of formulae). We consider two cases:

- $\mathcal{K}$ consists of clauses $C(x_1, \dots, x_n)$ containing Σ_1-functions.
- $\mathcal{K}$ consists of *augmented clauses*, i.e. of axioms of the form $(\Phi(x_1, \dots, x_n) \vee C(x_1, \dots, x_n))$, where $\Phi(x_1, \dots, x_n)$ is an *arbitrary first-order formula* in the base signature Π_0 and $C(x_1, \dots, x_n)$ is a *clause* containing Σ_1-functions.

The free variables $x_1, \dots, x_n$ are considered to be universally quantified.

Locality of an extension. The notion of locality for theory extensions was introduced in [16,7]. We focus on the following types of locality[1] of an extension $\mathcal{T}_0 \subseteq \mathcal{T}_1 = \mathcal{T}_0 \cup \mathcal{K}$ with $\mathcal{K}$ a set of clauses (resp. augmented clauses for (ELoc)).

(Loc) For every finite set G of ground clauses $\mathcal{T}_1 \cup G \models \perp$ iff $\mathcal{T}_0 \cup \mathcal{K}[G] \cup G \models \perp$
(ELoc) For every formula $\Gamma = \Gamma_0 \cup G$, where Γ_0 is a Π_0^c-sentence and G is a finite set of ground Π^c-clauses, $\mathcal{T}_1 \cup \Gamma \models \perp$ iff $\mathcal{T}_0 \cup \mathcal{K}[\Gamma] \cup \Gamma \models \perp$

where $\mathcal{K}[G]$ is the set of instances of $\mathcal{K}$ where all terms starting with an extension function are in the set $\mathsf{st}(\mathcal{K}, G)$ of all ground terms occurring in $\mathcal{K}$ or G.

We say that an extension $\mathcal{T}_0 \subseteq \mathcal{T}_1$ is *local* if it satisfies condition (Loc). We refer to condition (ELoc) as *extended locality* condition. The notions of Ψ-locality and Ψ-extended locality of a theory extension were introduced in [11] to encompass situations in which the instances to be considered are described by a closure operation Ψ. We say that $\mathcal{K}$ is Ψ-local if it satisfies:

(Loc^Ψ) for every finite set G of ground clauses, $\mathcal{K} \cup G \models \perp$ iff $\mathcal{K}[\Psi_{\mathcal{K}}(G)] \cup G \models \perp$

where $\Psi_{\mathcal{K}}(G) = \Psi(\mathsf{st}(\mathcal{K}, G))$. Condition ($\mathsf{ELoc}^\Psi$) is defined analogously. If $\Psi_{\mathcal{K}}(G) = \mathsf{st}(\mathcal{K}, G)$ we recover the notions (Loc) resp. (ELoc).

[1] In what follows, when we refer to *sets* G of ground clauses we assume that they are in the signature $\Pi^c = (S, \Sigma \cup \mathcal{C}, \mathsf{Pred})$, where $\mathcal{C}$ is a set of new constants.

Hierarchical reasoning. Consider a Ψ-local theory extension $\mathcal{T}_0 \subseteq \mathcal{T}_0 \cup \mathcal{K}$. Condition ($\mathsf{Loc}^{\Psi}$) requires that, for every set G of ground Π^c clauses, $\mathcal{T}_0 \cup \mathcal{K} \cup G \models \perp$ iff $\mathcal{T}_0 \cup \mathcal{K}[\Psi_{\mathcal{K}}(G)] \cup G \models \perp$. All clauses in $\mathcal{K}[\Psi_{\mathcal{K}}(G)] \cup G$ have the property that the function symbols in Σ_1 have as arguments only ground terms, so $\mathcal{K}[\Psi_{\mathcal{K}}(G)] \cup G$ can be flattened and purified[2]. The set of clauses thus obtained has the form $\mathcal{K}_0 \cup G_0 \cup \mathsf{Def}$, where Def consists of ground unit clauses of the form $f(g_1, \dots, g_n) = c$, where $f \in \Sigma_1$, c is a constant, $g_1, \dots, g_n$ are ground terms without Σ_1-function symbols, and $\mathcal{K}_0$ and G_0 do not contain Σ_1-function symbols. (In what follows we always flatten and then purify $\mathcal{K}[\Psi_{\mathcal{K}}(G)] \cup G$ to ensure that the ground unit clauses in Def are $f(c_1, \dots, c_n) = c$, where $c_1, \dots, c_n, c$ are constants.)

Theorem 1 ([16,11]). *Let $\mathcal{K}$ be a set of clauses. Assume that $\mathcal{T}_0 \subseteq \mathcal{T}_1 = \mathcal{T}_0 \cup \mathcal{K}$ is a Ψ-local theory extension. For any set G of ground clauses, let $\mathcal{K}_0 \cup G_0 \cup \mathsf{Def}$ be obtained from $\mathcal{K}[\Psi_{\mathcal{K}}(G)] \cup G$ by flattening and purification, as explained above. Then the following are equivalent to $\mathcal{T}_1 \cup G \models \perp$:*

(1) $\mathcal{T}_0 \cup \mathcal{K}[\Psi_{\mathcal{K}}(G)] \cup G \models \perp$.
(2) $\mathcal{T}_0 \cup \mathcal{K}_0 \cup G_0 \cup \mathsf{Def} \models \perp$.
(3) $\mathcal{T}_0 \cup \mathcal{K}_0 \cup G_0 \cup \mathsf{Con}[G]_0 \models \perp$, *where*

$$\mathsf{Con}[G]_0 = \{ \bigwedge_{i=1}^{n} c_i = d_i \rightarrow c = d \mid f(c_1, \dots, c_n) = c, f(d_1, \dots, d_n) = d \in \mathsf{Def} \}.$$

A similar equivalence holds for extended Ψ-local extensions, with the remark that in that case $\mathcal{K}_0$ and G_0 may contain arbitrary Π_0-sentences.

Decidability, parameterized complexity. Theorem 1 allows us to show that if for every finite set T of terms $\Psi_{\mathcal{K}}(T)$ is finite then (i) decidability of satisfiability w.r.t. a Ψ-local extension $\mathcal{T}_1$ of a theory $\mathcal{T}_0$ is a consequence of the decidability of the satisfiability of a certain fragment of $\mathcal{T}_0$, and (ii) the complexity of such satisfiability tests in $\mathcal{T}_1$ can be expressed as a function of the complexity of satisfiability checking for a suitable fragment of $\mathcal{T}_0$.

Theorem 2 ([16,11]). *Assume that the theory extension $\mathcal{T}_0 \subseteq \mathcal{T}_1 = \mathcal{T}_0 \cup \mathcal{K}$ satisfies condition ((E)Loc^{Ψ}). Satisfiability of $\Gamma_0 \cup G$ as in the definition of ((E)Loc^{Ψ}) w.r.t. $\mathcal{T}_1$ is decidable provided $\mathcal{K}[\Psi_{\mathcal{K}}(G)]$ is finite and $\mathcal{K}_0 \cup G_0 \cup \Gamma_0 \cup \mathsf{Con}[G]_0$ belongs to a decidable fragment $\mathcal{F}$ of $\mathcal{T}_0$. If (i) the complexity of testing the satisfiability of a set of formulae in $\mathcal{F}$ of size m w.r.t. $\mathcal{T}_0$ can be described by a function $g(m)$ and (ii) G is a set of $\mathcal{T}_1$-clauses of size n, then the complexity of checking whether $G \models_{\mathcal{T}_1} \perp$ is $g(n^k)$, where k is the maximum number of free variables in a clause in $\mathcal{K}$ (but at least 2).*

The (Ψ-)locality of an extension can be recognized by proving embeddability of partial into total models assuming that the extension clauses are flat and linear

[2] The function symbols in Σ_1 are separated from the other symbols by introducing, in a bottom-up manner, new constants c_t for subterms $t{=}f(g_1, \dots, g_n)$ with $f{\in}\Sigma_1$, g_i ground Π_0^c-terms together with corresponding definitions $c_t{=}t$ (C is a set of constants which contains the constants introduced by flattening, resp. purification).

[16,17,11]. (The converse implication also holds.) If we can guarantee that the support of the total model which we obtain is the same as the support of the partial model we start with, then condition (ELoc^{Ψ}) is guaranteed. We present some examples of theory extensions which proved to be Ψ-local in previous work:

Free and bounded functions. Any extension of a theory $\mathcal{T}_0$ with free function symbols is local. Assume $\mathcal{T}_0$ contains a binary predicate $\leq$, and $f \not\in \Sigma_0$. For $1 \leq i \leq m$ let $t_i(x_1, \dots, x_n)$ and $s_i(x_1, \dots, x_n)$ be terms in the signature Π_0 and $\phi_i(x_1, \dots, x_n)$ be Π_0-formulae with (free) variables among $x_1, \dots, x_n$, such that (i) $\mathcal{T}_0 \models \forall \overline{x}(\phi_i(\overline{x}) \rightarrow \exists y(s_i(\overline{x}) \leq y \leq t_i(\overline{x})))$, and (ii) if $i \neq j$, $\phi_i \wedge \phi_j \models_{\mathcal{T}_0} \perp$. The extension $\mathcal{T}_0 \subseteq \mathcal{T}_0 \cup \mathsf{GB}(f)$ satisfies condition ELoc [17,11], if

$$\mathsf{GB}(f) = \bigwedge_{i=1}^{m} \mathsf{GB}^{\phi_i}(f), \text{ where } \quad \mathsf{GB}^{\phi_i}(f): \quad \forall \overline{x}(\phi_i(\overline{x}) \rightarrow s_i(\overline{x}) \leq f(\overline{x}) \leq t_i(\overline{x})).$$

Monotonicity, boundedness for monotone functions. Any extension of a theory for which $\leq$ is a partial order (or at least reflexive) with functions satisfying[3]: $\mathsf{Mon}^{\sigma}(f)$ and $\mathsf{Bound}^{t}(f)$ is local [17,11]. The extensions satisfy condition (ELoc) if e.g. in $\mathcal{T}_0$ all finite and empty infima (or suprema) exist.

$$\mathsf{Mon}^{\sigma}(f) \quad \bigwedge_{i \in I} x_i {\leq_i}^{\sigma_i} y_i \wedge \bigwedge_{i \not\in I} x_i = y_i \rightarrow f(x_1, .., x_n) \leq f(y_1, .., y_n)$$

$$\mathsf{Bound}^{t}(f) \; \forall x_1, \dots, x_n (f(x_1, \dots, x_n) \leq t(x_1, \dots, x_n))$$

where $t(x_1, \dots, x_n)$ is a Π_0-term with variables among $x_1, \dots, x_n$ whose associated function has the same monotonicity as f in any model.

Convexity/concavity [18]. Let f be a unary function, and $I = [a, b]$ a subset of the domain of definition of f. We consider the axiom:

$$\mathsf{Conv}^{I}(f) \;\; \forall x, y, z \left(x, y{\in}I \wedge x \leq z \leq y \rightarrow \frac{f(z){-}f(x)}{z{-}x} \leq \frac{f(y){-}f(x)}{y{-}x} \right).$$

Then $\mathcal{T}_0{\subseteq}\mathcal{T}_0{\cup}\mathsf{Conv}_f^I$ satisfies condition (ELoc) if $\mathcal{T}_0 = \mathbb{R}$ (the theory of reals), or $\mathcal{T}_0 = \mathbb{Z}$ (e.g. Presburger arithmetic), or $\mathcal{T}_0$ is the many-sorted combination of the theories of reals (sort real) and integers (sort int) and f has arity $\mathsf{int} \rightarrow \mathsf{real}$.

Bounds on the slope. The extension $\mathbb{R} \subseteq \mathbb{R}{\cup}\mathsf{BS}_f^{c_1, c_2}$ satisfies condition (ELoc), where $\mathsf{BS}_f^{c_1, c_2}$ is the following boundedness condition on the slope of f:

$$\mathsf{BS}_f^{c_1, c_2} \quad \forall x, y \left(x \neq y \rightarrow c_1 \leq \tfrac{f(x) - f(y)}{x - y} \leq c_2 \right).$$

It is easy to see that condition ($\mathsf{BS}_f^{c_1, c_2}$) can be relativized to (unions of) intervals $I = [a, b]$ and combined with boundedness conditions of the form $(x \in I \rightarrow s_I(x) \leq f(x) \leq t_I(x))$ if (i) s_I is convex and t_I concave on I or (ii) the slope of s_I is smaller than c_1 and the slope of t_I larger than c_2 on I.

In [18] we proved that the conditions above can be combined with continuity and sometimes also with the derivability of the functions.
The following locality transfer result proved to be very useful.

[3] For $i \in I$, $\sigma_i{\in}\{-, +\}$, and for $i \not\in I$, $\sigma_i = 0$; $\leq^{+}{=}{\leq}, \leq^{-}{=}{\geq}$.

Theorem 3 ([12]). *Let $\Pi_0 = (\Sigma_0, \mathsf{Pred})$ be a signature, $\mathcal{T}_0$ a Π_0-theory, Σ_1 and Σ_2 two disjoint sets of new function symbols, $\Pi_i = (\Sigma_0 \cup \Sigma_i, \mathsf{Pred})$, $i = 1, 2$, and $\mathcal{K}$ a set of flat and linear Π_1-clauses. Assume that the extension $\mathcal{T}_0 \subseteq \mathcal{T}_0 \cup \mathcal{K}$ satisfies condition* ELoc *as a consequence of an embeddability condition in which the support of the models does not change. Let $\mathcal{T}_2$ be a Π_2-theory such that $\mathcal{T}_0 \subseteq \mathcal{T}_2$. Then the extension $\mathcal{T}_2 \subseteq \mathcal{T}_2 \cup \mathcal{K}$ satisfies condition* ELoc *as well.*

3 Verification Problems for Parametric Systems

We identify situations in which decision procedures exist for invariant checking and bounded model checking, as well as methods for obtaining constraints between the parameters which guarantee that certain properties are invariants.

3.1 Systems Modeled Using Transition Constraints

We specify reactive systems using tuples $(\Pi_S, \mathcal{T}_S, T_S)$ where Π_S is a signature and $\mathcal{T}_S$ is a Π_S-theory (describing the data types used in the specification and their properties), and $T_S = (V, \Sigma, \mathsf{Init}, \mathsf{Update})$ is a transition constraint system which specifies: the variables (V) and function symbols (Σ) whose values change over time; a formula Init specifying the properties of initial states; a formula Update with variables in $V \cup V'$ and function symbols in $\Sigma \cup \Sigma'$ (where V' and Σ' are copies of V resp. Σ, denoting the variables resp. functions after the transition) which specifies the relationship between the values of variables x (function f) before a transition and their values x' (f') after the transition.
We consider *invariant checking* and *bounded model checking* problems[4], cf. [14]:

Invariant checking. A formula Ψ is an inductive invariant of a system S with theory $\mathcal{T}_S$ and transition constraint system $T_S{=}(V, \Sigma, \mathsf{Init}, \mathsf{Update})$ if:
(1) $\mathcal{T}_S, \mathsf{Init} \models \Psi$ and (2) $\mathcal{T}_S, \Psi, \mathsf{Update} \models \Psi'$, where Ψ' results from Ψ by replacing each $x \in \mathcal{V}$ by x' and each $f \in \Sigma$ by f'.

Bounded model checking. We check whether, for a fixed k, unsafe states are reachable in at most k steps. Formally, we check whether:

$$\mathcal{T}_S \wedge \mathsf{Init}_0 \wedge \bigwedge_{i=1}^{j} \mathsf{Update}_i \wedge \neg\Psi_j \models \perp \qquad \text{for all } 0 \leq j \leq k,$$

where Update_i is obtained from Update by replacing every $x{\in}V$ by x_i, every $f{\in}\Sigma$ by f_i, and each $x'{\in}V'$, $f'{\in}\Sigma'$ by x_{i+1}, f_{i+1}; Init_0 is Init with x_0 replacing $x \in V$ and f_0 replacing $f{\in}\Sigma$; Ψ_i is obtained from Ψ similarly.

Let Γ be a formula describing additional constraints on the parameters. To check whether a formula Ψ is an inductive invariant under the constraints Γ we need to analyze whether the following holds (where $\mathcal{T}_1 = \mathcal{T}_S \cup \Gamma$):

$$\begin{array}{lr} (1) & \exists \overline{x}(\mathsf{Init}(\overline{x}) \wedge \neg\Psi(\overline{x})) \models_{\mathcal{T}_1} \perp \\ (2) & \exists \overline{x}, \overline{x}'(\Psi(\overline{x}) \wedge \mathsf{Update}(\overline{x}, \overline{x}') \wedge \neg\Psi(\overline{x}')) \models_{\mathcal{T}_1} \perp\ . \end{array}$$

[4] In what follows we only address invariant checking; the problems which occur in bounded model checking are similar.

These are satisfiability problems for possibly quantified formulae w.r.t. an underlying theory $\mathcal{T}_1$. We are interested in identifying situations in which the problems above are decidable, and also in the dual problem of inferring a set $\overline{\Gamma}$ of *most general constraints* on the parameters which guarantee that Ψ is an invariant.

Definition 4. *Let $\overline{\Gamma}$ be a formula expressing constraints on the parameters of a verification problem. We say that $\overline{\Gamma}$ is the* weakest condition under which Ψ is an inductive invariant *iff for every formula Γ expressing constraints on the parameters, Ψ is an inductive invariant under Γ iff $\mathcal{T}_S \cup \Gamma \models \overline{\Gamma}$.*

We distinguish the following, increasingly more complicated situations.

Case 1. There are no functional parameters, and only variables change value in updates. Theorem 5 is a consequence of the form of the formulae in (1) and (2).

Theorem 5. *Assume that the formulae $\mathsf{Init}(\overline{x}), \Psi(\overline{x}), \neg\Psi(\overline{x}), \mathsf{Update}(\overline{x}, \overline{x}')$ belong to a fragment $\mathcal{F}$ of the theory $\mathcal{T}_S$ closed under conjunction.*

(a) Assume that checking satisfiability of formulae in $\mathcal{F}$ w.r.t. $\mathcal{T}_S$ is decidable. If in the description of S no parameters are used or a set of constraints on the parameters $\Gamma(\overline{p})$ belonging to the fragment $\mathcal{F}$ is given, then checking whether the formula Ψ is an invariant (under conditions $\Gamma(\overline{p})$) is decidable.
(b) If $\mathcal{T}$ allows quantifier elimination then we can use this to effectively construct a weakest condition $\overline{\Gamma}$ on the parameters under which Ψ is an invariant.

Example 1 in Section 1.1 illustrates the way Theorem 5 can be used. Theorem 5(b) can also be used for generating invariants with a given shape, expressed using undetermined constants which can be considered to be parameters.

Case 2. Only variables change their value in updates, but some parameters of the system are functions, possibly satisfying a set of axioms $\mathcal{K}$, and $\mathcal{T}_S = \mathcal{T}_0 \cup \mathcal{K}$.

Theorem 6. *Assume that the formulae $\mathsf{Init}(\overline{x}), \Psi(\overline{x}), \neg\Psi(\overline{x}), \mathsf{Update}(\overline{x}, \overline{x}')$ belong to a fragment $\mathcal{F}$ of the theory $\mathcal{T}_S$ closed under conjunction. Assume that satisfiability of formulae in $\mathcal{F}$ w.r.t. $\mathcal{T}_S$ is decidable. Let $\Gamma(\overline{p}) = \Gamma_0 \cup \Gamma_\Sigma$ be a set of constraints on the parameters of T_S, where: (i) Γ_0 is a set of constraints on the non-functional parameters which belongs to the fragment $\mathcal{F}$, and (ii) Γ_Σ is a set of axioms containing functional parameters such that satisfiability of formulae in $\mathcal{F}$ w.r.t. $\mathcal{T}_S \cup \Gamma_\Sigma$ is decidable. Then checking whether the formula Ψ is an invariant (under conditions $\Gamma(\overline{p})$ on the parameters) is decidable.*

Proof: Decidability of problems of type (1) above is a consequence of the fact that $\mathsf{Init}(\overline{x}) \wedge \neg\Psi(\overline{x}) \models_{\mathcal{T}_S \cup \Gamma} \perp$ iff $\Gamma_0 \wedge \mathsf{Init}(\overline{x}) \wedge \neg\Psi(\overline{x})$ (which by assumption is in $\mathcal{F}$) is unsatisfiable w.r.t. $\mathcal{T}_S \cup \Gamma_\Sigma$. To prove decidability of problem (2) note that $\Psi(\overline{x}) \wedge \mathsf{Update}(\overline{x}, \overline{x}') \wedge \neg\Psi(\overline{x}') \models_{\mathcal{T}_S \cup \Gamma} \perp$ iff $\Gamma_0 \wedge \Psi(\overline{x}) \wedge \mathsf{Update}(\overline{x}, \overline{x}') \wedge \neg\Psi(\overline{x}')$ (which is again in $\mathcal{F}$) is unsatisfiable w.r.t. $\mathcal{T}_S \cup \Gamma_\Sigma$. $\square$

The conditions of Theorem 6 hold e.g. if the decidability of the problems above is a consequence of locality properties of certain theory extensions, i.e. if the following conditions are satisfied:

Condition 1: $\mathcal{T}_S$ is an extension of a Π_0-theory $\mathcal{T}_0$ with a set $\mathcal{K}$ of flat and linear clauses (properties of the parameters in $\Sigma_1 \subseteq \Sigma$) satisfying ELoc (and the additional requirement in Thm. 3), and s.t. all variables occurring in clauses in $\mathcal{K}$ occur below a Σ-function symbol.

Condition 2: $\mathsf{Init}(\overline{x}), \Psi(\overline{x})$, and $\neg\Psi(\overline{x})$ are quantifier-free Π_0-formulae.

Condition 3: $\mathsf{Update}(\overline{x}, \overline{x}')$ are quantifier-free formulae possibly containing also functional parameters in Σ.

For clarity, in the conditions above and Theorem 7 we consider the particular case in which locality allows us to reduce all proof tasks to satisfiability checks for *ground* formulae w.r.t. $\mathcal{T}_0$. We will then briefly discuss the way the results extend in the presence of extended locality conditions.

Theorem 7. *Assume that Conditions 1–3 hold.*

(a) Assume that ground satisfiability of formulae in $\mathcal{T}_0$ is decidable. Let $\Gamma = \Gamma_0 \cup \Gamma_\Sigma$ be a set of constraints on $\Sigma_2 \subseteq \Sigma$, where $\Sigma_1 \cap \Sigma_2 = \emptyset$, s.t. Γ_0 is a quantifier-free Π_0-formula with no variables (only with parameters) and Γ_Σ is a set of flat and linear clauses such that $\mathcal{T}_0 \subseteq \mathcal{T}_0 \cup \Gamma_\Sigma$ is a local extension. Then checking whether Ψ is an invariant (under conditions Γ) is decidable.

(b) If the theory $\mathcal{T}_0$ has quantifier elimination, this can be used to construct a weakest condition $\overline{\Gamma}$ on the parameters under which Ψ is an invariant.

Proof: (a) follows from Thm. 2 since by Thm. 3, $\mathcal{T}_0 \cup \Gamma \subseteq \mathcal{T}_0 \cup \mathcal{K} \cup \Gamma$ satisfies ELoc.

(b) The fact that the initial states satisfy Ψ can clearly be expressed as a satisfiability problem w.r.t. $\mathcal{T}_0$. We can eliminate the existentially quantified variables in $\overline{x}$ using a quantifier elimination method for $\mathcal{T}_0$ to obtain a weakest condition $\overline{\Gamma_1}(\overline{p})$ on the parameters under which $\Psi \models \mathsf{Init}$. We now analyze updates. Let Γ be an arbitrary set of constraints for parametric functions not in Σ_1. Ψ is invariant under updates (under conditions Γ) iff $\exists \overline{x} \exists \overline{x}' \, (\Psi(\overline{x}) \wedge \mathsf{Update}(\overline{x}, \overline{x}') \wedge \neg\Psi(\overline{x}')) \models_{\mathcal{T}_0 \cup \mathcal{K} \cup \Gamma} \perp$. As $\mathcal{T}_0 \cup \Gamma \subseteq \mathcal{T}_0 \cup \mathcal{K} \cup \Gamma$ is local, the following are equivalent:

– $\exists \overline{x} \exists \overline{x}' \, (\Psi(\overline{x}) \wedge \mathsf{Update}(\overline{x}, \overline{x}') \wedge \neg\Psi(\overline{x}')) \models_{\mathcal{T}_0 \cup \Gamma \cup \mathcal{K}} \perp$.
– $\mathcal{T}_0 \cup \Gamma \cup G' \models \perp$, where G' is a set of ground Π_0-clauses (containing in addition to the parameters and the constants obtained from $\overline{x}, \overline{x}'$ by Skolemization also additional constants x_f obtained from renaming extension terms).
– $\exists \overline{x} \exists \overline{x}' \exists \overline{x}_f G' \models_{\mathcal{T}_0 \cup \Gamma} \perp$.
– $\exists \overline{x} \Gamma_2(\overline{y}, \overline{p}) \models_{\mathcal{T}_0 \cup \Gamma} \perp$, where Γ_2 is obtained by eliminating (using QE in $\mathcal{T}_0$) all existentially quantified variables from $\exists \overline{x} \exists \overline{x}' \exists \overline{x}_f G'$ except for those used for terms starting with Σ-functions and their arguments.
– $\mathcal{T}_0 \cup \Gamma \models \forall \overline{y} \overline{\Gamma}_2(\overline{y}, \overline{p})$, where $\overline{\Gamma}_2$ is obtained from $\neg\Gamma_2$ by replacing back the newly introduced constants with the terms they represent.

If $\mathcal{K}$ does not contain any constraints, $\overline{\Gamma} = \overline{\Gamma_1} \wedge \overline{\Gamma_2}$ is the weakest condition under which Ψ is an invariant. □

Example 4. Consider the water controller in Example 2 in Section 1.1 where the inflow in depends on time. We assume that time is discrete (modeled by the integers), and that the values of the water level are real numbers. Let $\mathcal{T}_S = \mathcal{T}_0 \cup \mathsf{Free}(\mathsf{in})$ be the extension of the many-sorted combination $\mathcal{T}_0$ of $\mathbb{Z}$ (Presburger arithmetic) and $\mathbb{R}$ (the theory of reals) with the free function in. Then for every set Γ of constraints, $L \leq L_{\mathsf{overflow}}$ is an invariant under Γ iff the following formulae are unsatisfiable w.r.t. $\mathcal{T}_1 = \mathcal{T}_S \cup \Gamma$:

(i) $\exists L, L', t, t' (L_{\mathsf{overflow}} {\geq} L {\geq} L_{\mathsf{alarm}} \wedge L'{=}L{+}\mathsf{in}(t){-}\mathsf{out} \wedge t'{=}t{+}1 \wedge L'{>}L_{\mathsf{overflow}})$
(ii) $\exists L, L', t, t' (L{<}L_{\mathsf{alarm}} \wedge L'{=}L{+}\mathsf{in}(t) \wedge t'{=}t{+}1 \wedge L'{>}L_{\mathsf{overflow}})$

Consider formula (i). By the locality of the extension with the free function in, the following are equivalent:

(a) $(L_{\mathsf{overflow}} {\geq} L {\geq} L_{\mathsf{alarm}} \wedge L'{=}L{+}\mathsf{in}(t){-}\mathsf{out} \wedge t'{=}t{+}1 \wedge L'{>}L_{\mathsf{overflow}}) \models_{\mathcal{T}_1} \perp$.
(b) $(L_{\mathsf{overflow}} {\geq} L {\geq} L_{\mathsf{alarm}} \wedge L'{=}L{+}\mathsf{in}_0{-}\mathsf{out} \wedge t'{=}t{+}1 \wedge L'{>}L_{\mathsf{overflow}}) \wedge \mathsf{Def} \models_{\mathcal{T}_1} \perp$, where Def consists of the formula $(\mathsf{in}_0 = \mathsf{in}(t))$.
(c) $(L_{\mathsf{overflow}} {\geq} L {\geq} L_{\mathsf{alarm}} \wedge L'{=}L{+}\mathsf{in}_0{-}\mathsf{out} \wedge t'{=}t{+}1 \wedge L'{>}L_{\mathsf{overflow}}) \models_{\mathcal{T}_0 \cup \Gamma} \perp$.

We now eliminate all variables except for the parameters $L_{\mathsf{overflow}}, L_{\mathsf{alarm}}, \mathsf{in}_0$ and the variable t (argument for in) and obtain that under the assumption that $L_{\mathsf{alarm}} < L_{\mathsf{overflow}}$ the formula in (c) is equivalent to $\mathsf{in}_0{-}\mathsf{out} > 0$ i.e. – replacing in_0 with $\mathsf{in}(t)$ – with $\exists t(\mathsf{in}(t){-}\mathsf{out} > 0)$. Thus, (c) holds iff

(d) $\mathcal{T}_0 \cup \Gamma \models \overline{\Gamma}_1$, where $\overline{\Gamma}_1 = \forall t(\mathsf{in}(t) - \mathsf{out} \leq 0)$.

For formula (ii) we can similarly construct $\overline{\Gamma}_2 = \forall t(\mathsf{in}(t) \leq L_{\mathsf{overflow}} {-} L_{\mathsf{alarm}})$, so $\overline{\Gamma} = \overline{\Gamma}_1 \wedge \overline{\Gamma}_2$ is the weakest constraint under which $L > L_{\mathsf{overflow}}$ is invariant.

Comment. For fully exploiting the power of extended locality, we can relax Conditions 1–3 and allow $\mathcal{K}$ and Γ to consist of augmented clauses, require that $\mathcal{T}_0 \subseteq \mathcal{T}_0 \cup \Gamma$ satisfies ELoc; allow $\mathsf{Init}(\overline{x}), \Psi(\overline{x}), \neg\Psi(\overline{x})$ to be arbitrary Π_0-formulae and require that $\mathsf{Update}(\overline{x}, \overline{x}')$ consists of augmented clauses in which arbitrary Π_0-formulae are allowed to appear. The decidability results still hold if we can guarantee that the formulae we obtain with the hierarchical reduction belong to a fragment for which satisfiability w.r.t. $\mathcal{T}_0$ is decidable.

Case 3. Variables in V and functions in Σ may change their values during the transitions. We consider transition constraint systems T_S in which the formulae in Update contain variables in X and functions in Σ and possibly parameters in Σ_p. We assume that $\Sigma_p {\cap} \Sigma = \emptyset$. We therefore assume that the background theory $\mathcal{T}_S$ is an extension of a Π_0-theory $\mathcal{T}_0$ with axioms $\mathcal{K}$ specifying the properties of the functions in $\Sigma_1 \subseteq \Sigma \backslash \Sigma_p$. We make the following assumptions:

Assumption 1: $\mathsf{Init}(\overline{x})$ and $\Psi(\overline{x})$ are conjunctions of clauses with free variables $\overline{x}$ which do not contain functional parameters in Σ_p.
Assumption 2: $\mathcal{T}_0 {\cup} \mathcal{K} {\cup} \mathsf{Init}$ and $\mathcal{T}_0 {\cup} \mathcal{K} {\cup} \Psi$ are flat and linear extensions of $\mathcal{T}_0$ satisfying ELoc (and the additional requirements in Thm. 3).
Assumption 3: For every $f \in \Sigma$, $\mathsf{Update}(f, f')$ – describing the update rules for f – is a set of clauses which, for every Π_S-theory $\mathcal{T}$, defines an extension with a new function $f' \notin \Sigma$, such that $\mathcal{T} {\subseteq} \mathcal{T} {\cup} \mathsf{Update}(f, f')$ satisfies ELoc.[5]

[5] This is always the case if $\mathsf{Update}(f, f')$ are updates by definitions for f' depending on a partition of the state space, or updates by guarded boundedness conditions.

Theorem 8. *Under Assumptions 1–3 the following hold:*

(a) Assume that ground satisfiability of formulae in $\mathcal{T}_0$ is decidable. Let Γ be a set of clauses expressing constraints on parameters in Σ_p (and not containing functions in Σ) s.t. $\mathcal{T}_0 \subseteq \mathcal{T}_0 \cup \Gamma$ is a local extension. Then checking whether Ψ is an invariant (under conditions Γ) is decidable.
(b) If the theory $\mathcal{T}_0$ has quantifier elimination, this can be used to construct a weakest condition $\overline{\Gamma}$ on the parameters under which Ψ is an invariant.

Proof: (a) By Theorem 3, Assumption 3 implies that $\mathcal{T}_0 \cup \Gamma \subseteq \mathcal{T}_0 \cup \Gamma \cup \mathcal{K} \cup \mathsf{Init}$ and $\mathcal{T}_0 \cup \Gamma \subseteq \mathcal{T}_0 \cup \Gamma \cup \mathcal{K} \cup \Psi$ satisfy condition (ELoc). We first analyze the problem of showing that initial states satisfy Ψ under conditions Γ. Since $\mathcal{T}_0 \cup \Gamma \subseteq \mathcal{T}_0 \cup \Gamma \cup \mathcal{K} \cup \mathsf{Init}$ satisfies condition (ELoc), the following are equivalent:

(1) $\exists \overline{x}(\mathsf{Init}(\overline{x}, \overline{f}) \wedge \neg \Psi(\overline{x}, \overline{f})) \models_{\mathcal{T}_0 \cup \Gamma \cup \mathcal{K}} \perp$.
(2) $\mathcal{T}_0 \cup \Gamma \cup G' \models \perp$ where G' is obtained from $\neg\Psi$, Init and $\mathcal{K}$ using the hierarchical reduction in Theorem 1.

$\mathcal{T}_0 \subseteq \mathcal{T}_0 \cup \Gamma$ is a local extension, so (2) can be reduced to a ground satisfiability check w.r.t. $\mathcal{T}_0$ (which is decidable). Invariance under updates is similar.
(b) Let Γ be a set of constraints referring to functional parameters in Σ_p; without Σ symbols. We use the equivalence of (1) and (2) established in (a), and note that (2) is equivalent to $\exists \overline{x} \exists \overline{x}_f G' \models_{\mathcal{T}_0 \cup \Gamma} \perp$, hence also to (3) and to (4) below:

(3) $\exists \overline{y} \exists \overline{x}_f \Gamma_1(\overline{y}, \overline{x}_f) \models_{\mathcal{T}_0 \cup \Gamma} \perp$, where Γ_1 is obtained from $\exists \overline{x} \exists \overline{x}_f G'$ by eliminating all existentially quantified variables apart from the variables $\overline{y}$ which occur as arguments of functional parameters (using QE in $\mathcal{T}_0$).
(4) $\mathcal{T}_0 \cup \Gamma \models \forall \overline{y} \overline{\Gamma}_1(\overline{y})$, where $\overline{\Gamma}_1(\overline{y})$ is obtained from $\neg\Gamma_1$ by replacing back every x_f with the term it represents.

Invariance under transitions can be solved similarly and yields a constraint $\overline{\Gamma}_2$. As before, $\overline{\Gamma} = \overline{\Gamma}_1 \wedge \overline{\Gamma}_2$ is the weakest constraint under which Ψ is invariant. □

Comment. We can extend this result to fully exploit extended locality by allowing, in Assumptions 1–3, $\mathcal{K}$, Init, Ψ, Update to consist of augmented clauses, requiring ELoc for $\mathcal{T}_0 \subseteq \mathcal{T}_0 \cup \Gamma$, and decidability of $\mathcal{T}_0$-satisfiability for the fragment to which the formulae obtained after the hierarchical reduction belong.

Example 5. Consider an algorithm for inserting an element c into a sorted array a at a (fixed, but parametric) position i_0. We want to derive constraints on the value of c which guarantee that the array remains sorted after the insertion. Let $\mathcal{T}_S$ be the disjoint combination of Presburger arithmetic ($\mathbb{Z}$, sort index) and a theory $\mathcal{T}_e$ of elements (here, $\mathbb{R}$, sort elem). We model the array a by using a function a of sort $\mathsf{index} \rightarrow \mathsf{elem}$, and a constant ub of sort index (for the size of the array). The safety condition is the condition that the array is sorted, i.e.

$\mathsf{Sorted}(a, ub)$ $\quad \forall i, j : \mathsf{index}(0 \leq i \leq j \leq ub \rightarrow a(i) \leq a(j))$.

The update rules are described by the following formula:

$\mathsf{Update}(a, a', ub, ub')$ $\quad \forall i : \mathsf{index}(0 \leq i < i_0 \rightarrow a'(i) = a(i)) \wedge \; a'(i_0) = c \; \wedge$
$\forall i : \mathsf{index}(i_0 < i \leq ub \rightarrow a'(i) = a(i-1)) \wedge \; ub' = ub + 1$

We want to determine conditions on c and a s.t. sortedness is preserved, i.e.:

$$\mathsf{Sorted}(a, ub) \wedge \mathsf{Update}(a, a', ub, ub') \wedge \exists j(0 \leq j \leq ub' - 1 \wedge a'(j) > a'(j+1)) \models \perp .$$

The examples of local extensions in Section 2 show that Assumptions 1–3 are fulfilled, hence the following are equivalent for every set Γ of constraints:

– $\mathsf{Sorted}(a, ub) \wedge \mathsf{Update}(a, a', ub, ub') \wedge (0 \leq d \leq ub' - 1) \wedge a'(d) > a'(d+1) \models_{\mathcal{T}_S \cup \Gamma} \perp$.
– $\mathsf{Sorted}(a, ub)[G'] \wedge G' \models_{\mathcal{T}_S \cup \Gamma} \perp$, where
  $G' = 0 \leq d \leq ub' - 1 \wedge a'(d) > a'(d+1) \wedge a'(i_0) = c \wedge ub' = ub + 1 \wedge$
  $(0 \leq d < i_0 \rightarrow a'(d) = a(d)) \wedge (i_0 < d \leq ub \rightarrow a'(d) = a(d-1)) \wedge a'(i_0) = c \wedge$
  $(0 \leq d+1 < i_0 \rightarrow a'(d+1) = a(d+1)) \wedge (i_0 < d+1 \leq ub \rightarrow a'(d+1) = a(d))$.
– $a(d-1) \leq a(d) \wedge a(d) \leq a(d+1) \wedge G' \models_{\mathcal{T}_S \cup \Gamma} \perp$.

Any set of clauses $\bigwedge_{i=1}^n (\phi_i(x, f) \rightarrow C_i(x, x', f, f'))$, where $\phi_i \wedge \phi_j \models_{\mathcal{T}} \perp$ for $i \neq j$ and $\models_{\mathcal{T}} \bigvee_{i=1}^n \phi_i$ is equivalent to $\bigvee_{i=1}^n (\phi_i(x, f) \wedge C_i(x, x', f, f'))$. The two instances of the update axioms in G' have this form. By the transformation above and distributivity we obtain the following equivalent DNF formula:

– $(\psi_1 \wedge \psi) \vee (\psi_2 \wedge \psi) \vee (\psi_3 \wedge \psi) \vee (\psi_4 \wedge \psi) \models_{\mathcal{T}_S \cup \Gamma} \perp$, where

$$\begin{aligned}
\psi &= a(d-1) \leq a(d) \wedge a(d) \leq a(d+1) \wedge a'(d) > a'(d+1) \wedge a'(i_0) = c \wedge ub' = ub - 1 \\
\psi_1 &= (0 \leq d < d+1 < i_0 \wedge a'(d) = a(d) \wedge a'(d+1) = a(d+1)) \\
\psi_2 &= (0 \leq d < d+1 = i_0 \leq ub \wedge a'(d) = a(d) \wedge a'(d+1) = c) \\
\psi_3 &= (0 \leq d = i_0 < d+1 \leq ub \wedge a'(d) = c \wedge a'(d+1) = a(d)) \\
\psi_4 &= (0 \leq i_0 < d < d+1 \leq ub \wedge a'(d) = a(d-1) \wedge a'(d+1) = a(d)).
\end{aligned}$$

$\psi_1 \wedge \psi$ and $\psi_4 \wedge \psi$ are clearly unsatisfiable. Consider now $\psi_2 \wedge \psi$. We purify the formulae and eliminate all constants (i.e. existentially quantified variables) with the exception of c, i_0, d, $d-1$, $d+1$, and those which rename $a(d)$, $a(d-1)$, $a(d+1)$. Then $\psi_2 \wedge \psi$ is unsatisfiable w.r.t. $\mathcal{T}_S \cup \Gamma$ iff

$$\mathcal{T}_S \cup \Gamma \models \overline{\Gamma}_2, \text{ where } \overline{\Gamma}_2 = \forall d(0 \leq d < i_0 \wedge a(d) \leq a(i_0) \wedge a(d-1) \leq a(d) \rightarrow a(d) \leq c).$$

Under the assumption of sortedness for a, we obtain the equivalent condition: $\overline{\Gamma}'_2 = \forall x(x < i_0 \rightarrow a(x) \leq c)$. Similarly, from $\psi_3 \wedge \psi$ we obtain condition $\overline{\Gamma}'_3 = \forall x(i_0 \leq x \rightarrow c \leq a(x))$. Thus, the weakest condition under which Ψ is an invariant (assuming sortedness) is $\overline{\Gamma}' = \forall x[(x < i_0 \rightarrow a(x) \leq c) \wedge (i_0 \leq x \rightarrow c \leq a(x))]$.

We also consider the problem of determining conditions on a and c under which a' is sorted, without a priori assuming sortedness for a. Then $\psi_1 \wedge \psi \models \perp$ yields $\overline{\Gamma}_1 = \forall x(0 \leq x < x+1 < i_0 \rightarrow a(x) \leq a(x+1))$ and $\psi_4 \wedge \psi \models \perp$ yields $\overline{\Gamma}_4 = \forall x(i_0 \leq x < x+1 < ub \rightarrow a(x) \leq a(x+1))$. Hence, the overall condition we obtain is in this case

$$\overline{\Gamma} = \mathsf{Sorted}(a) \wedge \forall x((x < i_0 \rightarrow a(x) \leq c) \wedge (i_0 \leq x \rightarrow c \leq a(x))).$$

Note that these simple formulae were obtained after various simplifications depending on the properties of the parametric functions we consider. At the moment, no system (neither for automated reasoning nor for symbolic computation) provides such simplification facilities. Simplification and redundancy removal modulo theories is a topic which we plan to explore in the future.

3.2 Hybrid Automata

We use descriptions of hybrid automata $(Q, V, \Sigma, \mathsf{Init}, \mathsf{Inv}, \mathsf{Jump}, \mathsf{Flow})$ specifying the *variables* (V) whose values change over time; a formula Init describing the initial states; the *discrete states* (Q); for every $q \in Q$ a formula Inv_q representing the invariant for state q; rules $\mathsf{Flow}_q(\overline{x}, t, t')$ describing the *evolution* of the variables $\overline{x}$ in state q; *transitions* between discrete states, specified using formulae $\mathsf{Jump}(q, q', \overline{x}, \overline{x}')$. We here restrict to *abstractions* of hybrid systems in which the rules $\mathsf{Flow}_q(\overline{x}, t, t')$ are represented as formulae. As before, for the sake of simplicity we only consider *invariant checking.* (Bounded model checking can be handled analogously.) A formula Ψ is an inductive invariant of a hybrid system $T{=}(V, \Sigma, \mathsf{Init}, \mathsf{Inv}, \mathsf{Flow}, \mathsf{Jump})$ if:

(1) $\mathcal{T}_S \cup \mathsf{Init}(x) \models \Psi(x)$;
(2) $\mathcal{T}_S \cup \Psi(x(t_0)) \cup \mathsf{Flow}_q(\overline{x}, t_0, t) \models \Psi(x(t))$.
(3) $\mathcal{T}_S \cup \Psi(x) \cup \mathsf{Jump}(q, q', x, x') \models \Psi(x')$.

Invariance under jumps can be checked as for discrete systems. Invariance under flows can be expressed as the unsatisfiability of formulae of the following form:

$$\textbf{(F)}\ \exists t, t'\, (\Psi(\overline{x}(t)) \wedge \mathsf{Inv}_q(\overline{x}(t)) \wedge \mathsf{Flow}_q(\overline{x}, t, t') \\ \wedge \forall t''(t \leq t'' \leq t' \rightarrow (\mathsf{Flow}_q(\overline{x}, t, t'') \wedge \mathsf{Inv}_q(\overline{x}(t'')))) \quad \wedge \quad \neg \Psi(\overline{x}'))$$

stating that there exist two moments t, t' such that $x(t)$ satisfies the invariant of state q and Ψ, and x changes until moment t' (at which Ψ does not hold) using the flow rule from state q, without leaving this state. These are satisfiability problems for quantified formulae w.r.t. an underlying theory.

Constants bounds on slope. We analyze the case when the evolution of the continuous variable x within any discrete state q is described by giving *constant bounds* $l_q \leq u_q$ on the rate of growth of the values of x (cf. e.g. [1]). For simplicity of presentation, we here assume that there is only one continuous variable x. Then for every discrete state q with invariant $\mathsf{Inv}_q(x)$, the formula describing the update of x after a flow from time t to time t' within state q is:

$$\mathsf{Flow}_q(x, t, t') : \quad t < t' \wedge l_q(t' - t) \leq x(t') - x(t) \leq u_q(t' - t).$$

Theorem 9. *Assume that (i) all continuous variables are modeled as functions over $\mathbb{R}$, (ii) the formulae describing $\mathsf{Inv}_q(x)$ and Ψ are quantifier-free and (ii) $\mathbb{R} \cup \mathcal{K}$ is an extension of $\mathbb{R}$ with a new function symbol x which satisfies condition* ELoc*, where $\mathcal{K}$ is of the form (with t, t' constants and t'' universally quantified):*

$$\{t \leq t'' \leq t' \rightarrow x(t) + l \cdot (t'' - t) \leq x(t'') \leq x(t) + u \cdot (t'' - t), t \leq t'' \leq t' \rightarrow \mathsf{Inv}_q(x(t''))\}.$$

(a) Let Γ be an additional constraint on the values of x expressed again by a set of flat and linear clauses such that $\mathbb{R} \subseteq \mathbb{R} \cup \mathcal{K} \cup \Gamma$ satisfies Loc*. Then checking whether the formula Ψ is an invariant (under conditions Γ) is decidable.*

(b) We can use the QE procedure for the theory of reals to effectively construct a weakest constraint $\overline{\Gamma}$ on the parameters under which Ψ is an invariant.

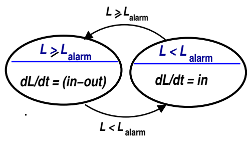
Proof: Similar to the proof of Theorem 8 □

The state invariants Inv_q are often conjunctions of boundedness conditions on the continuous variables.

Corollary 10. *Assume that* Inv_q *are of the form* $l_q(t) \leq x(t) \leq u_q(t)$, *where* l_q *and* u_q *are polynomials such that for every partially defined function* x *weakly satisfying the boundedness condition for the slope in* $\mathsf{BS}_x^{l_q,u_q}$, *the piecewise linear function* $\mathsf{lin\text{-}appr}(x)$ *which passes through all points* $\{(t, x(t)) \mid x \in Dom(x)\}$ *has the property that for every* t *we have* $l_q(t) \leq \mathsf{lin\text{-}appr}(x) \leq u_q(t)$.[6] *Then the conclusions (a) and (b) in Theorem 9 hold.*

The conditions in Theorem 9 subsume the case of hybrid systems in which the flows within the discrete states are given by specifying a *fixed* rate of growth, i.e. $x(t) = x(t_0) + c(t - t_0)$ (equal upper and lower bounds), also written $dx/dt = c$.

Example 6. Consider the following hybrid system introduced in Example 3. There are two discrete states (state invariants $L \geq L_{\mathsf{alarm}}$ and $L < L_{\mathsf{alarm}}$).



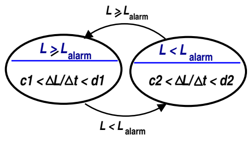
Assume that the inflow/outflow rates are constant and equal to in/out. $L < L_{\mathsf{overflow}}$ is invariant under flows iff (i) and (ii) are unsatisfiable (where $\mathsf{in}' = \mathsf{in} - \mathsf{out}, \mathsf{green}(t) = L(t) \leq L_{\mathsf{alarm}}$ and $\mathsf{red}(t) = L_{\mathsf{alarm}} \leq L(t) \leq L_{\mathsf{overflow}}$):

(i) $\exists t, t'(\mathsf{green}(t) \wedge t < t' \wedge \forall t''(t \leq t'' \leq t' \rightarrow L(t) + \mathsf{in}(t'' - t) < L_{\mathsf{alarm}}) \wedge L(t) + \mathsf{in}(t' - t) > L_{\mathsf{overflow}})$

(ii) $\exists t, t'(\mathsf{red}(t) \wedge t < t' \wedge \forall t''(t \leq t'' \leq t' \rightarrow L(t) + \mathsf{in}'(t'' - t) \geq L_{\mathsf{alarm}}) \wedge L(t) + \mathsf{in}'(t' - t) > L_{\mathsf{overflow}})$.

Using the method described in Theorem 9 and QE over $\mathbb{R}$ we can show that (i) is always unsatisfiable and that assuming that $(\mathsf{in} > 0)$ and $(L_{\mathsf{alarm}} < L_{\mathsf{overflow}})$ (ii) is equivalent to $(\mathsf{in} > \mathsf{out}) \wedge (L_{\mathsf{alarm}} < L(t))$, thus (ii) holds iff $(\mathsf{in} \leq \mathsf{out})$.

Example 7. We consider a variant of the previous example in which we assume that the rate with which the water level changes satisfies the following conditions:



$(L \geq L_{\mathsf{alarm}}): \quad \forall t_1, t_2 (t_1 \neq t_2 \rightarrow c_1 \leq \frac{L(t_2) - L(t_1)}{t_2 - t_2} \leq d_1)$

$(L < L_{\mathsf{alarm}}): \quad \forall t_1, t_2 (t_1 \neq t_2 \rightarrow c_2 \leq \frac{L(t_2) - L(t_1)}{t_2 - t_2} \leq d_2)$

where c_1, d_1, c_2, d_2 are parameters. As before, $L \leq L_{\mathsf{overflow}}$ is an invariant iff the following formulae are unsatisfiable:

(i) $\exists t, t'(L(t) < L_{\mathsf{alarm}} \wedge t < t' \wedge L(t') > L_{\mathsf{overflow}} \wedge \forall t''(t \leq t'' \leq t' \rightarrow L(t'') < L_{\mathsf{alarm}}))$
$\wedge \forall t''(t \leq t'' \leq t' \rightarrow c_2(t'' - t) \leq L(t'') - L(t) \leq d_2(t'' - t))$

(ii) $\exists t, t'(L_{\mathsf{overflow}} \geq L(t) \geq L_{\mathsf{alarm}} \wedge t < t' \wedge L(t') > L_{\mathsf{overflow}} \wedge \forall t''(t \leq t'' \leq t' \rightarrow L(t'') \geq L_{\mathsf{alarm}}))$
$\wedge \forall t''(t \leq t'' \leq t' \rightarrow c_1(t'' - t) \leq L(t'') - L(t) \leq d_1(t'' - t))$

Let $\Gamma = \{c_1 \leq d_1, c_2 \leq d_2\}$. Using the method in Theorem 9 and QE over $\mathbb{R}$ we can show that under the assumption that $L_{\mathsf{alarm}} < L_{\mathsf{overflow}}$ (i) is always

[6] This happens for instance if l_q is convex and u_q concave on $[0, t]$ or the slope of l_q is smaller than c_1 and that of u_q greater than c_2 (under these conditions, we can even allow l_q, u_q to contain additional functions).

unsatisfiable. For (ii) note that due to the locality of the extensions defined by the universally quantified formulae in (ii), (ii) is unsatisfiable iff (ii') is unsatisfiable:

(ii') $\exists t, t'(L_{\mathsf{overflow}} \geq L(t) \geq L_{\mathsf{alarm}} \ \wedge \ t < t' \ \wedge \ L(t') > L_{\mathsf{overflow}} \wedge$
$(c_1(t'-t) \leq L(t') - L(t) \leq d_1(t'-t)) \wedge (L(t) \geq L_{\mathsf{alarm}}) \wedge (L(t') \geq L_{\mathsf{alarm}}))$.

After purification we use QE to eliminate the existential variables (t, t' and the variables standing for $x(t), x(t')$). Using Redlog [5] we obtained a formula which could be simplified (as $c_1 \leq d_1$) to $d_1 > 0$. Thus, safety is guaranteed for all flows within this state as long as $d_1 \leq 0$ (i.e. the water level is not increasing).

4 Conclusion

In this paper we studied certain classes of verification problems for parametric reactive and simple hybrid systems. We identified some deductive problems which need to be solved, and properties of the underlying theories which ensure that these verification problems are decidable. We gave examples of theories with the desired properties, and illustrated the methods on several examples.

Parametricity in hybrid systems was addressed before in e.g. [1,15,6,20]. Some approaches to invariant generation use a parametric form for the invariants and use constraint solving for generating invariants with a certain shape [2,9]. In all these approaches, the parameters are constants occurring in the description of the systems or in the invariants. In e.g. [8,4] also functions are used in the description of reactive or hybrid systems. In this paper we go one step further: we allow both functions and data to be parametric, and present ways of constructing (weakest) constraints on such parameters which guarantee safety – which turns out to be very useful. In ongoing work we extended these ideas to linear hybrid automata: we obtained locality results for boundedness conditions of linear combinations of values of the continuous variables and for linear combinations of their slopes and established several complexity results. In this paper we did not study the link between the abstractions of hybrid systems we consider w.r.t. hybrid systems they may model (we do not impose any restrictions on the derivability of the continuous variables). This is planned for future work. We also did not formally analyze situations in which the flows within the discrete states are described by differential equations. We tackled some examples (e.g. a temperature controller in which the continuous variable is the temperature, and the evolution of the external temperature is a functional parameter) by generating abstractions similar to those used in Section 3.2 and identified situations in which constraints on these parametric functions which *imply* safety can be derived. (We showed e.g. that the "cooling" state of the temperature controller is safe provided the outside temperature is smaller than the interior temperature.) Since we use an abstraction, we cannot always guarantee that these constraints are "weakest". A formal study of such more general hybrid systems is in progress.

Acknowledgments. We thank the referees for their helpful comments. This work was partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS). See `www.avacs.org` for more information.

References

1. Alur, R., Henzinger, T.A., Ho, P.H.: Automatic Symbolic Verification of Embedded Systems. IEEE Trans. Software Eng. 22(3), 181–201 (1996)
2. Beyer, D., Henzinger, T., Majumdar, R., Rybalchenko, A.: Invariant Synthesis for Combined Theories. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 378–394. Springer, Heidelberg (2007)
3. Cimatti, A., Palopoli, L., Ramadian, Y.: Symbolic Computation of Schedulability Regions Using Parametric Timed Automata. In: IEEE Real-Time Systems Symposium 2008, pp. 80–89. IEEE Computer Society, Los Alamitos (2008)
4. Cimatti, A., Roveri, M., Tonetta, S.: Requirements Validation for Hybrid Systems. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 188–203. Springer, Heidelberg (2009)
5. Dolzmann, A., Sturm, T.: Redlog: Computer Algebra Meets Computer Logic. ACM SIGSAM Bulletin 31(2), 2–9 (1997)
6. Frehse, G., Jha, S.K., Krogh, B.H.: A Counterexample-Guided Approach to Parameter Synthesis for Linear Hybrid Automata. In: Egerstedt, M., Mishra, B. (eds.) HSCC 2008. LNCS, vol. 4981, pp. 187–200. Springer, Heidelberg (2008)
7. Ganzinger, H., Sofronie-Stokkermans, V., Waldmann, U.: Modular proof systems for partial functions with Evans equality. Information and Computation 204(10), 1453–1492 (2006)
8. Ghilardi, S., Nicolini, E., Ranise, S., Zucchelli, D.: Combination Methods for Satisfiability and Model-Checking of Infinite-State Systems. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 362–378. Springer, Heidelberg (2007)
9. Gulwani, S., Tiwari, A.: Constraint-Based Approach for Analysis of Hybrid Systems. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 190–203. Springer, Heidelberg (2008)
10. Hune, T., Romijn, J., Stoelinga, M., Vaandrager, F.: Linear Parametric Model Checking of Timed Automata. Journal of Logic and Algebraic Programming 52-53, 183–220 (2002)
11. Ihlemann, C., Jacobs, S., Sofronie-Stokkermans, V.: On Local Reasoning in Verification. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 265–281. Springer, Heidelberg (2008)
12. Ihlemann, C., Sofronie-Stokkermans, V.: On Hierarchical Reasoning in Combinations of Theories. In: Giesl, J., Hähnle, R. (eds.) Proceedings of IJCAR 2010. LNCS (LNAI), vol. 6173, pp. 30–45. Springer, Heidelberg (2010)
13. Jacobs, S., Sofronie-Stokkermans, V.: Applications of Hierarchical Reasoning in the Verification of Complex Systems. Electr. Notes Theor. Comput. Sci. 174(8), 39–54 (2007)
14. Manna, Z., Pnueli, A.: Temporal Verification of Reactive Systems: Safety. Springer, Heidelberg (1995)
15. Platzer, A., Quesel, J.-D.: European Train Control System: A Case Study in Formal Verification. In: Cavalcanti, A. (ed.) ICFEM 2009. LNCS, vol. 5885, pp. 246–265. Springer, Heidelberg (2009)

16. Sofronie-Stokkermans, V.: Hierarchic Reasoning in Local Theory Extensions. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 219–234. Springer, Heidelberg (2005)
17. Sofronie-Stokkermans, V., Ihlemann, C.: Automated Reasoning in some Local Extensions of Ordered Structures. Journal of Multiple-Valued Logics and Soft Computing 13(4-6), 397–414 (2007)
18. Sofronie-Stokkermans, V.: Efficient Hierarchical Reasoning about Functions over Numerical Domains. In: Dengel, A.R., Berns, K., Breuel, T.M., Bomarius, F., Roth-Berghofer, T.R. (eds.) KI 2008. LNCS (LNAI), vol. 5243, pp. 135–143. Springer, Heidelberg (2008)
19. Tarski, A.: A Decision Method for Elementary Algebra and Geometry, 2nd edn. University of California Press, Berkeley (1951)
20. Wang, F.: Symbolic Parametric Safety Analysis of Linear Hybrid Systems with BDD-Like Data-Structures. IEEE Trans. Software Eng. 31(1), 38–51 (2005)

Interpolation and Symbol Elimination in Vampire*

Kryštof Hoder[1], Laura Kovács[2], and Andrei Voronkov[1]

[1] University of Manchester
[2] TU Vienna

Abstract. It has recently been shown that proofs in which some symbols are colored (e.g. local or split proofs and symbol-eliminating proofs) can be used for a number of applications, such as invariant generation and computing interpolants. This tool paper describes how such proofs and interpolant generation are implemented in the first-order theorem prover Vampire.

1 Introduction

Interpolation offers a systematic way to generate auxiliary assertions needed for software verification techniques based on theorem proving [7,10], predicate abstraction [5,7], constraint solving [15], and model-checking [12,1].

In [9] it was shown that symbol-eliminating inferences extracted from proofs can be used for automatic invariant generation. Further, [10] gives a new proof of a result from [7] on extracting interpolants from colored proofs;[1] this proof contains an algorithm for building (from colored proofs) interpolants that are boolean combinations of symbol-eliminating steps. Thus, [10] brings interpolation and symbol elimination together.

Based on the results of [9,10] we implemented colored proof generation in the first-order theorem prover Vampire [14]. Colored proofs form the base for our interpolation and symbol elimination algorithms.

The purpose of this paper is to describe how interpolation and symbol elimination are implemented and can be used in Vampire. We do not overview Vampire itself but only describe its new functionalities. The presented features have been explicitly designed for making Vampire appropriate for formal software verification: symbol elimination for automated assertion (invariant) synthesis and computation of Craig interpolants for abstraction refinement. Unlike its predecessors, the "new" Vampire thus provides functionalities which extend the applicability of state-of-the-art first theorem provers in verification. To the best of our knowledge, it is the first theorem prover that supports both invariant generation and interpolant computation.

The obtained symbol eliminating inferences and interpolants contain quantifiers, and can be further used as invariant assertions to verify properties of programs manipulating arrays and linked lists [13,9]. We believe that software verification may benefit from the interpolant generation engine of Vampire.

Implementation. The new version of Vampire is available from `http://www.vprover.org` and runs under most recent versions of Linux (both 32 and 64 bits),

* This work has been partly done while the second authors was at ETH Zürich.

[1] Such proofs are also called *local* and *split proofs*, in this paper we will call them colored.

MacOS and Windows. Vampire is implemented in C++ and has about 73,000 lines of code.

Experiments. We successfully applied Vampire on benchmarks taken from recent work on interpolants and invariants [6,19,3,4,15,8] – see Section 4 and the mentioned URL. Our methods can discover required invariants and interpolants in all examples, suggesting its potential for automated software verification.

Related work. There are several interpolant generation algorithms for various theories. For example, [12,5,7,1] derive interpolants from resolution proofs in the combined ground theory of linear arithmetic and uninterpreted functions. The approach described in [15] generates interpolants in the combined theory of arithmetic and uninterpreted functions using constraint solving techniques over an a priori defined interpolants template. The method presented in [13] computes quantified interpolants from first-order resolution proofs over scalars, arrays and uninterpreted functions.

Our algorithm implemented in Vampire automatically extracts interpolants from colored first-order proofs in the superposition calculus. Theories, such as arithmetic or theories of arrays, can be handled by adding theory axioms to the first-order problem to be proved. Thus, interpolation in Vampire is not limited to decidable theories for which interpolation algorithms are known. One can use arbitrary first-order axioms. However, a consequence of this generality is that we do not guarantee finding interpolants even for decidable theories. Moreover, if a theory is not finitely axiomatisable, we can only use its incomplete first-order axiomatisation.

As far as we know, symbol elimination has not been implemented in any other system. A somehow related approach to symbol elimination is presented in [13,16] where theorem proving is used for inferring loop invariants. Contrary to our approach, the cited works are adapted to prove given assertions as opposed to generating arbitrary invariants. Using the saturation-based theorem prover SPASS [18], [13] generates interpolants as quantified invariants that are strong enough to prove given assertions. In [16] templates over predicate abstraction are used, reducing the problem of invariant discovery to that of finding solutions, by the Z3 SMT solver [2], for unknowns in an invariant template formula. Unlike [13,16], we automatically generate invariants as symbol eliminating inferences in full-first order logic, without using predefined predicate templates or assertions.

2 Colored Proofs, Symbol Elimination and Interpolation

Colored proofs are used in a context when some (predicate and/or function) symbols are declared to have colors. In colored proofs every inference can use symbols of at most one color, as a consequence, every term or atomic formula used in such proofs can use symbols of at most one color, too. We will call a symbol, term, clause etc. *colored* if it uses a color, otherwise it is called *transparent*.

In **symbol elimination** [9] we are interested in inferences having at least one colored premise and a transparent conclusion; such inferences are called *symbol-eliminating*. Conclusions of symbol-eliminating inferences can be used to find loop invariants. Symbol elimination can be reformulated as *consequence-finding*: we are trying to find transparent consequences of a theory including both colored and transparent formulas.

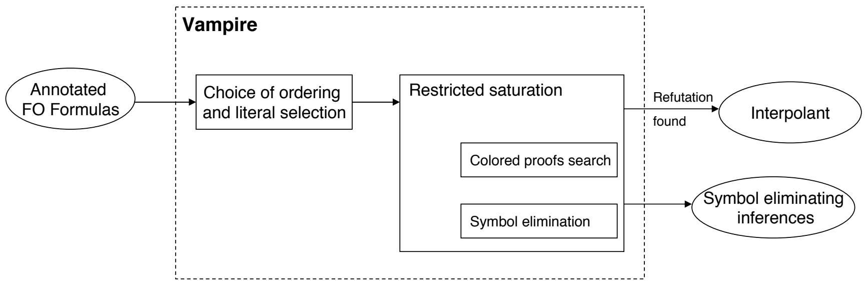


Fig. 1. Interpolation and Symbol Elimination in Vampire

Note that, unlike traditional applications of first-order theorem proving, we are not interested in finding a refutation: symbol-eliminating inferences can be obtained by running a theorem prover on a satisfiable formula, for which no refutation exists.

A formula I is called an **interpolant** of formulas L and R (with respect to a theory T) if the following conditions are satisfied:

(1) $T \vdash L \rightarrow I$;
(2) $T \vdash I \rightarrow R$;
(3) I uses only symbols occurring either in T or in both L and R.

Interpolation can be reformulated in terms of colors as follows: we assign one color to symbols occurring only in L and another color to symbols occurring only in R: then the last condition on interpolants can be reformulated as *I is transparent*. For extracting interpolants from colored proofs we use the algorithm described in [10].

The notion of interpolant has been changed in the model-checking community starting with [12]. Namely, the condition (2): $T \vdash I \rightarrow R$ has been replaced by (2a): $T \vdash I \wedge R \rightarrow \bot$. To avoid any confusion between the two notions of interpolant, in [10] any formula I satisfying conditions (1), (2a) and (3) is called a *reverse interpolant of L and R*. Clearly, reverse interpolants for L and R are exactly interpolants of L and $\neg R$.

In the sequel, we reserve the notation L and R for the two formulas whose interpolant is to be computed.

3 Tool Overview

Vampire [14] is a general purpose first-order theorem prover based on the resolution and superposition calculus. To implement symbol elimination and interpolation in Vampire, we had to extend it by new functionalities, change the inference mechanism to be able to generate colored derivations, and implement an algorithm for extracting interpolants. The workflow of interpolation and symbol elimination in Vampire is illustrated in Figure 1.

Annotated formulas. Vampire reads problems expressed in the TPTP syntax [17]: a Prolog-like syntax allowing one to specify input axioms and conjecture for theorem provers. We had to extend the input syntax to make it rich enough to define colors and

```
vampire(symbol,function,a,0,left).
vampire(symbol,function,b,0,left).
vampire(symbol,predicate,q,1,left).
vampire(symbol,function,c,0,right).
vampire(option,show_interpolant,on).
```

```
vampire(left_formula).
  fof(a1,axiom, q(f(a))).
  fof(a1,axiom, ~q(f(b))).
vampire(end_formula).
vampire(right_formula).
  fof(a2,conjecture, ? [V] : (f(V)!=c)).
vampire(end_formula).
```

Fig. 2. Specification of Interpolation

interpolation requests. In fact, we had to extend it even more since in the application of interpolation and symbol elimination the set of symbols that can occur in the interpolants is not necessarily the intersection of the languages of L and R with addition of theory symbols. We extended the TPTP syntax with Vampire-specific declarations. Their use is illustrated in Figure 2 and detailed in Example 1 taken from [13].

EXAMPLE 1. [13] Consider the problem of computing an interpolant of $q(f(a)) \wedge \neg q(f(b))$ (i.e. L) and $\exists v(f(v) \neq c)$ (i.e. R).

The first three declarations shown in the left column of Figure 2 say that a, b are constants (function symbols of arity 0) and q is a unary predicate symbol colored in the "left" color (that is, in the language of L). Likewise, the fourth declaration in the left column says that c is a constant colored in the "right" color (that is, in the language of R). Finally, the left column contains an option that sets interpolant generation. This option can also be passed in the command line. The declarations `fof(..)` are TPTP declarations for introducing formulas. The vampire declarations `left_formula`, `right_formula` and `end_formula` are used to define L and R. If we have formulas not in the scope of the `left_formula` or `right_formula` declarations, they are considered as part of the theory T.

To use Vampire for symbol elimination, we can simply assign all symbols to be eliminated the left color and leave the right color unused. For concrete examples see `http://www.vprover.org`.

Colored proof generation. In order to support the generation of colored proofs, the following had to be implemented.

1. We had to block inferences that have premises of two different colors.
2. We had to change the simplification ordering and literal selection, so that colored terms are larger than transparent ones, and that (when possible) transparent literals are selected only when there are no colored ones. To make colored terms bigger than transparent, we had to implement the Knuth-Bendix ordering with ordinals as defined in [11] and make colored symbols to have the weight ω, while transparent symbols to have finite weights.

To output the *conclusions of symbol eliminating inferences*, we check premises of each inference that produced a transparent clause, and if one of the premises is colored, we output the resulting clause.

Interpolants are generated from refutations using the algorithm described in [10]. For instance, given the input shown in Example 1, Vampire outputs the interpolant $\neg\forall x \forall y(f(x) = f(y))$.

Table 1. Interpolation with Vampire

Formulas	Coloring	Reverse Interpolant
$L: z < 0 \wedge x \leq z \wedge y \leq x$ $R: y \leq 0 \wedge x + y \geq 0$	left: z right: -	$x < 0$
$L: g(a) = c + 5 \wedge f(g(a)) \geq c + 1$ $R: h(b) = d + 4 \wedge d = c + 1 \wedge f(h(b)) < c + 1$	left: g, a right: h, b	$c + 1 \leq f(c + 5)$
$L: p \leq c \wedge c \leq q \wedge f(c) = 1$ $R: q \leq d \wedge d \leq p \wedge f(d) = 0$	left: c right: d	$p \leq q \wedge (q > p \vee f(p) = 1)$
$L: f(x_1) + x_2 = x_3 \wedge f(y_1) + y_2 = y_3 \wedge y_1 \leq x_1$ $R: x_2 = g(b) \wedge y_2 = g(b) \wedge x_1 \leq y_1 \wedge x_3 < y_3$	left: f right: g, b	$x_1 > y_1 \vee x_2 \neq y_2 \vee x_3 = y_3$
$L: c_2 = car(c_1) \wedge c_3 = cdr(c_1) \wedge \neg(atom(c_1))$ $R: \neg(c_1) = cons(c_2, c_3)$	left: $car, cons$ right: -	$\neg atom(c_1) \wedge c_1 = cons(c_2, c_3)$
$L: Q(f(a)) \wedge \neq Q(f(b))$ $R: f(V) = c$	left: Q, a, b right: c	$\exists x, y : f(x) \neq f(y)$
$L: a = c \wedge f(c) = a$ $R: c = b \wedge \neq (b = f(c))$	left: a right: b	$c = f(c)$
$L: True \wedge a'[x'] = y \wedge x' = x \wedge y' = y + 1 \wedge z' = x'$ $R: \neg(a'[z'] = y' - 1)$	left: x, y right: -	$1 + a'[x'] = y' \wedge x' = z'$

Symbol Elimination. To make Vampire output conclusions of symbol-eliminating inferences, one should set the option `show_symbol_elimination` to `on`. As Vampire is not supposed to terminate in the symbol-eliminating mode, it is wise to specify a time limit when it is run in this mode.

4 Experiments

We have successfully run Vampire on benchmark examples taken from recent literature on interpolation and invariant generation. In this section we present two different sets of experimental results that underline the effectiveness of our implementation. The reported results were obtained on a machine with 2 GHz processor and 2GB of RAM.

Interpolation. Table 1 summarises some of our results for computing interpolants on examples that have been used as motivating examples by previous techniques [6,19,15,13]. The first column of Table 1 presents the input formulas L and R whose interpolants is going to be computed. The second column shows symbols declared colored, whereas the third column shows the interpolant generated by Vampire.

All interpolants given in Table 1 were computed by Vampire in essentially no time (e.g. in less than 0.1 second). In the first four examples of Table 1 a simple axiomatisation of arithmetic with the greater-than relation and successor function was used. The fifth example of Table 1 uses the theory of lists, whereas the last example of Table 1 uses the combined theory of arrays and arithmetic.

The last example of Table 1 originates from an example taken from [6], and is a request to prove the infeasibility of the following one-path program annotated by a pre- and a post-condition:

$$\{\top\} \quad a[x] := y; y := y + 1; z := x \quad \{a[z] \neq y - 1\}.$$

Let x, y, z, a denote the initial and x', y', z', a' the final values of program variables. Based on the bounded-model checking approach [12], proving infeasibility of the above program path boils down to computing an interpolant for the formulas $\top \wedge T(\{x, y, z, a\}, \{x', y', z', a'\})$ and $a'[z'] \neq y' - 1$, where $T(\{x, y, z, a\}, \{x', y', z', a'\}) \equiv a'[x'] = y \wedge x' = x \wedge y' = y + 1 \wedge z' = x'$ is the transition relation defined by the program. The

Table 2. Symbol Elimination with Vampire on Array Programs

Loop	♯ of SEI	♯ of Min SEI	SEI as Invariant
`Initialisation` [8] $a = 0;$ **while** $(a < m)$ **do** $aa[a] = 0; a = a + 1$ **end do**	399	15	$\forall x : 0 \leq x < a \rightarrow aa[x] = 0$
`Copy` [8] $a = 0;$ **while** $(a < m)$ **do** $bb[a] = aa[a]; a = a + 1$ **end do**	379	14	$\forall x : 0 \leq x < a \rightarrow bb[x] = aa[x]$
`Vararg` [8] $a = 0;$ **while** $(aa[a] > 0)$ **do** $a = a + 1$ **end do**	1	1	$\forall x : 0 \leq x < a \rightarrow aa[x] > 0$

`Partition` [4]
$a = 0; b = 0; c = 0;$
while $(a < m)$ **do**
if $(aa[a] >= 0)$
then $bb[b] = aa[a]; b = b + 1$
else $cc[c] = aa[a]; c = c + 1$
end if;
$a = a + 1$
end do
	150	61	$\forall x : 0 \leq x < b \rightarrow$
			$\exists y : 0 \leq y < a \rightarrow bb[x] = aa[y]$

`Partition_Init` [8]
$a = 0; c = 0;$
while $(a < m)$ **do**
if $(aa[a] == bb[a])$
then $cc[c] = a; c = c + 1$
end if;
$a = a + 1$
end do
	18	13	$\forall x : 0 \leq x < c \wedge 0 \leq x < a \rightarrow$
			$aa(cc(x)) = bb(cc(x))$

interpolant computed by Vampire proves that the program has no feasible path from the initial state to the final state.

Symbol Elimination. Experiments with symbol elimination on array programs taken from [4,8] are summarised in Table 2. We ran Vampire in the symbol elimination mode with a time limit of 10 seconds. We recall that each conclusion of a symbol-eliminating inference is a loop invariant [9].

For all examples of Table 2 we show in the rightmost column a desired invariant that could be computed using other techniques. We were interested in the following: (i) can Vampire generate the invariant itself and if not, (ii) can Vampire generate invariants that would imply the desired invariant?

After running Vampire in the symbol-eliminating mode we sometimes obtain a large set of invariants. The number of invariants (that is, the number of symbol-eliminating inferences) is shown in the second column of the Table 2. To make them usable we did the following minimization: remove invariants that are implied by the theory axioms or by other invariants. For the task we used Vampire itself. Obviously, the problem whether an invariant is implied by other invariants, is undecidable, so we ran Vampire with a time limit of 0.3 seconds once for each invariant, trying to prove it from the remaining invariants and the theory axioms. The number of invariants that could not be proved redundant is shown in the third column of Table 2.

We tried many more examples, always with success, that is, the invariants generated by Vampire using symbol elimination always implied the desired invariant. There are many interesting issues related to symbol elimination which cannot be discussed here due to lack of space.

5 Conclusion

We described how interpolant generation and symbol elimination are implemented and can be used in the first-order theorem prover Vampire. Future work includes integrating Vampire into software verification tools for automatically generating interpolants and supporting the entire process of verification. Inferring a minimal set of invariants and improving these invariants can also be done using theorem proving and remains an interesting topic for further research.

References

1. Cimatti, A., Griggio, A., Sebastiani, R.: Efficient Interpolant Generation in Satisfiability Modulo Theories. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 397–412. Springer, Heidelberg (2008)
2. 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)
3. Gopan, D., Reps, T.W., Sagiv, M.: A Framework for Numeric Analysis of Array Operations. In: POPL, pp. 338–350 (2005)
4. Gulwani, S., Tiwari, A.: Combining Abstract Interpreters. In: Proc. of PLDI, pp. 376–386 (2006)
5. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from Proofs. In: Proc. of POPL, pp. 232–244 (2004)
6. Jhala, R., McMillan, K.L.: Interpolant-Based Transition Relation Approximation. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 39–51. Springer, Heidelberg (2005)
7. Jhala, R., McMillan, K.L.: A Practical and Complete Approach to Predicate Refinement. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 459–473. Springer, Heidelberg (2006)
8. Jhala, R., McMillan, K.L.: Array Abstractions from Proofs. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 193–206. Springer, Heidelberg (2007)
9. Kovacs, L., Voronkov, A.: Finding Loop Invariants for Programs over Arrays Using a Theorem Prover. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 470–485. Springer, Heidelberg (2009)
10. Kovacs, L., Voronkov, A.: Interpolation and Symbol Elimination. In: Schmidt, R.A. (ed.) CADE 2009. LNCS, vol. 5663, pp. 199–213. Springer, Heidelberg (2009)
11. 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)
12. McMillan, K.L.: Interpolation and SAT-Based Model Checking. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 1–13. Springer, Heidelberg (2003)

13. McMillan, K.L.: Quantified Invariant Generation Using an Interpolating Saturation Prover. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 413–427. Springer, Heidelberg (2008)
14. Riazanov, A., Voronkov, A.: The Design and Implementation of Vampire. AI Communications 15(2-3), 91–110 (2002)
15. Rybalchenko, A., Sofronie-Stokkermans, V.: Constraint Solving for Interpolation. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 346–362. Springer, Heidelberg (2007)
16. Srivastava, S., Gulwani, S.: Program Verification using Templates over Predicate Abstraction. In: PLDI, pp. 223–234 (2009)
17. Sutcliffe, G.: The TPTP Problem Library and Associated Infrastructure. The FOF and CNF Parts, v3.5.0. J. of Automated Reasoning (to appear, 2009)
18. Weidenbach, C., Schmidt, R.A., Hillenbrand, T., Rusev, R., Topic, D.: System Description: SpassVersion 3.0. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 514–520. Springer, Heidelberg (2007)
19. Yorsh, G., Musuvathi, M.: A Combination Method for Generating Interpolants. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 353–368. Springer, Heidelberg (2005)

iProver-Eq: An Instantiation-Based Theorem Prover with Equality

Konstantin Korovin and Christoph Sticksel

School of Computer Science
The University of Manchester
{korovin,sticksel}@cs.man.ac.uk

Abstract. iProver-Eq is an implementation of an instantiation-based calculus Inst-Gen-Eq which is complete for first-order logic with equality. iProver-Eq extends the iProver system with superposition-based equational reasoning and maintains the distinctive features of the Inst-Gen method. In particular, first-order reasoning is combined with efficient ground satisfiability checking where the latter is delegated in a modular way to any state-of-the-art SMT solver. The first-order reasoning employs a saturation algorithm making use of redundancy elimination in form of blocking and simplification inferences. We describe the equational reasoning as it is implemented in iProver-Eq, the main challenges and techniques that are essential for efficiency.

1 Introduction

Instantiation-based methods (IMs) are a class of calculi for first-order clausal logic. The common idea is to instantiate clauses and to employ efficient propositional or more general ground reasoning methods in order to prove unsatisfiability or to find a model. Among other important properties, IMs naturally decide the first-order logic fragment of effectively propositional logic (EPR) which has interesting applications as it has been shown recently (see, e.g., [2] for an overview). Let us remark that Inst-Gen-Eq decides the EPR fragment modulo equality.

The basic idea of the Inst-Gen method, introduced in [4], is as follows. The set of first-order clauses is abstracted to a set of ground clauses by mapping all variables to the same ground term. If this ground abstraction is unsatisfiable, then the set of first-order clauses is also unsatisfiable. Otherwise, there is a ground model for the abstraction that is used to guide an instantiation process. The ground satisfiability check and construction of a ground model is delegated to an industrial-strength satisfiability modulo theories (SMT) solver in the presence of equations or to a propositional (SAT) solver if no equational reasoning is required.

The model is represented as a set of abstracted literals and an attempt is made to extend it to a model of the first-order clauses by reasoning on the first-order literals corresponding to the abstracted literals in the model. When this fails, new (not necessarily ground) instances of clauses are generated in a way that forces the ground reasoner to refine the model in the next iteration. Inst-Gen is therefore composed of two parts: ground satisfiability solving on the abstraction of the set of clauses and first-order reasoning on literals corresponding to ground literals in the model of the abstraction.

A characteristic feature of the Inst-Gen method is the delegation of the ground reasoning to a black-boxed off-the-shelf solver. The iProver-Eq system currently uses MiniSat [3] as the SAT solver and either CVC3 [1] or Z3 [7] as the SMT solver.

The iProver system [6] has treated equations only axiomatically, the iProver-Eq system adds equational reasoning based on term rewriting. Following the approach from [5], it implements a superposition-style calculus that both finds sets of inconsistent equational literals and obtains instantiating substitutions from the proof of their inconsistency.

The implementation addresses the combination of three main components:

1. ground reasoning by an SMT solver.
2. superposition-based equational reasoning with literals in a candidate model.
3. instantiation by extracting substitutions from proofs generated in 2.

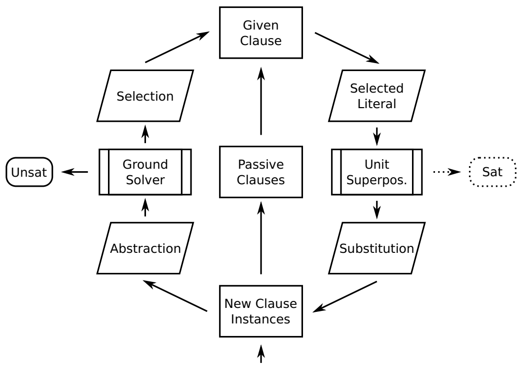
This system description first outlines the structure of the iProver-Eq system. We continue by defining the unit superposition calculus for equational reasoning and demonstrate it by way of an example. We discuss extraction of instantiating substitutions from proofs, giving an example for one of the non-trivial obstacles encountered which render the method incomplete if naively addressed. Finally, we highlight some of the main features of the implementation and conclude with an evaluation and directions for further research.

2 System Overview

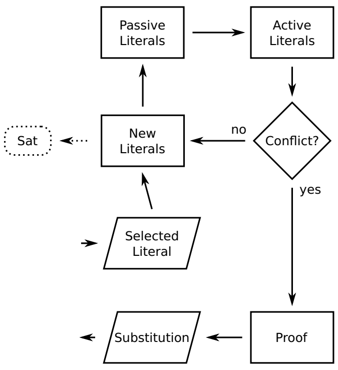
Given a set of first-order clauses S we first form its ground abstraction $S\bot$ by mapping all variables to the same ground term, conventionally denoted $\bot$. If the ground abstraction $S\bot$ is unsatisfiable, the original set S is also unsatisfiable and the procedure can terminate. Otherwise, there is a model $I_\bot$ of the ground abstraction $S\bot$ and the first-order instantiation process is guided by means of a selection function based on $I_\bot$. The selection function assigns to each first-order clause C in S exactly one literal L from C such that $I_\bot \models L\bot$. At least one such literal always exists as the ground abstraction of the clause is true in the model $I_\bot$.

If the set of selected (not necessarily ground) literals, seen as unit clauses, is consistent in first-order logic modulo equality, a model for the clause set S exists and it has thus been proved satisfiable. Otherwise, there is a subset of the selected literals which is inconsistent and the clauses these literals are selected in are instantiated such that the inconsistency can already be witnessed in the ground abstraction and thus forces the ground solver to refine it. For non-equational literals it suffices to search for unifiable complementary literal pairs. In the presence of equations, we apply the unit superposition calculus in order to find inconsistent literals and to obtain instantiating substitutions.

iProver-Eq generates instances of clauses in a saturation process outlined in Figure 1(a). Two major components in this process are unit superposition (US) for equational reasoning on literals and an SMT solver for ground reasoning. Both are non-trivial processes and while the equational reasoning will be described in the next section, the ground solver is regarded as a black box. The saturation process is based on



(a) Inst-Gen-Eq-Loop



(b) US-Loop

Fig. 1. Sketch of the iProver-Eq System

a given clause algorithm which partitions the set of clauses into two disjoint sets, in the following called the Inst-active and the Inst-passive clauses. The invariant is that the ground abstractions of the selected literals in the set of Inst-active clauses are consistent and have been passed to the US component. Initially, there are no Inst-active clauses, all clauses are considered to be new instances, their ground abstractions are input to the ground solver which is then invoked to return a model of the abstraction or to prove its unsatisfiability. The new clauses are moved to the Inst-passive set from where in each step of the process a clause, called the given clause, is chosen and put into the Inst-active set. Using the current model of the ground abstraction, one of the literals in the given clause is selected and passed to the unit superposition calculus, see Figure 1(b). If a subset of the selected literals is found to be inconsistent by the unit superposition calculus, then substitutions are extracted from the proof of the inconsistency and corresponding instances are added to the set of new clauses. The process continues by adding the abstractions of the new clauses to the SMT solver, running the solver on the extended set of ground clauses and moving the new clauses to the Inst-passive set. iProver-Eq terminates with a result of unsatisfiable if the ground solver reports an unsatisfiable abstraction. If the passive clause set is empty and the selected literals are consistent as indicated by the unit superposition component, iProver-Eq terminates with the result satisfiable.

3 The Unit Superposition Calculus

In this section we describe the inference rules of the unit superposition calculus for finding inconsistent equational literals and demonstrate it with an example (for a proof of completeness, see [5]). Subsequently we discuss the US-Loop saturation procedure.

For simplicity, we work with pure equational logic where all atoms are equations. The unit superposition calculus is similar to the standard superposition calculus, see, e.g., [8]. Different literals are assumed to be variable-disjoint and as we only work with literals, i.e. unit clauses, we can reduce the inference rules to the following ones.

Definition 1 (Unit Superposition)

$$\frac{l \simeq r \qquad s[l'] \simeq t}{(s[r] \simeq t)\sigma}\ (\sigma) \qquad \frac{l \simeq r \qquad s[l'] \not\simeq t}{(s[r] \not\simeq t)\sigma}\ (\sigma) \qquad \frac{l \not\simeq r}{\square}\ (\sigma)$$

(i) $\sigma = \mathrm{mgu}(l, l')$, *(ii)* l' *is not a variable, (iii)* $l\sigma\theta \succ r\sigma\theta$ *and (iv)* $s[l']\sigma\theta \succ t\sigma\theta$ *for some grounding substitution* θ

$\sigma = \mathrm{mgu}(l, r)$

A *proof* of the contradiction, denoted □, is a tree where the leaves are literals selected in the Inst-active set of clauses, inner nodes are obtained by applying inference rules to the parent nodes and the root is the contradiction □. In order to extract instantiating substitutions we annotate each inference with the substitution σ applied. The composition of the substitutions along the path in the proof tree from a selected literal at a leaf to the contradiction yields a substitution which we call *relevant to the selected literal.* In the Inst-Gen-Eq-Loop we take all clauses whose selected literals are leaves in the proof and instantiate each clause with the substitution relevant to its selected literal.

Example 1. Consider the clauses (1)-(4) below and let the ground abstractions of their selected literals (the first literal in each clause) be as shown to their right.

(1)	$f(f(u)) \simeq f(u)$	$f(f(\bot)) \simeq f(\bot)$
(2)	$g(f(f(x)), f(y)) \simeq h(z) \ \vee\ g(f(x), y) \not\simeq h(c)$	$g(f(f(\bot)), f(\bot)) \simeq h(\bot)$
(3)	$g(f(a), f(b)) \not\simeq h(w)$	$g(f(a), f(b)) \not\simeq h(\bot)$
(4)	$g(f(a), b) \simeq h(c)$	$g(f(a), b) \simeq h(c)$

The clause set is unsatisfiable, but its ground abstraction is satisfiable with a model containing the literals shown. Accordingly, the literals can indeed be selected and we derive the contradiction using the US calculus.

$$\cfrac{\cfrac{\cfrac{\overset{(1)}{f(f(u)) \simeq f(u)} \qquad \overset{(2)}{g(f(f(x)), f(y)) \simeq h(z)}}{g(f(x), f(y)) \simeq h(z)}\ \{x/u\} \qquad \overset{(3)}{g(f(a), f(b)) \not\simeq h(w)}}{h(z) \not\simeq h(w)}\ \{a/x, b/y\}}{\square}\ \{w/z\} \qquad (\star)$$

By tracing the branches in the proof tree, we extract a relevant substitution for each of the three literals: $\{a/u\}$, $\{a/x, b/y, w/z\}$ and $\{\}$, respectively. We instantiate the clauses with the relevant substitutions of their selected literals. Clause (3) is instantiated to itself, the instances of the first two clauses are:

(5)	$f(f(a)) \simeq f(a)$
(6)	$g(f(f(a)), f(b)) \simeq h(w) \ \vee\ g(f(a), b) \not\simeq h(c)$

The ground abstraction is now unsatisfiable due to the following four clauses.

$(3_\bot)$	$g(f(a), f(b)) \not\simeq h(\bot)$	$(5_\bot)$	$f(f(a)) \simeq f(a)$
$(4_\bot)$	$g(f(a), b) \simeq h(c)$	$(6_\bot)$	$g(f(f(a)), f(b)) \simeq h(\bot) \ \vee\ g(f(a), b) \not\simeq h(c)$

Implementation of Unit Superposition. Figure 1(b) on page 198 is a sketch of the saturation procedure in the unit superposition component in the Inst-Gen-Eq-Loop. The saturation algorithm is a given literal algorithm, a variant of the given clause algorithm described above. Literals are either US-active or US-passive and the invariant is that all inferences between two US-active literals have been drawn. The US-passive set is populated with selected literals from the Inst-Gen-Eq-Loop in Figure 1(a). In every step a given literal is chosen from the US-passive set, put into the US-active set and all conclusions from inferences between the given literal and US-active literals are drawn. The given literal is considered to be US-active in order to enable inferences with itself. If a conclusion is contradictory, substitutions are extracted from its proof as shown above and passed to the Inst-Gen-Eq-Loop. Otherwise, the conclusion is added to the US-passive set and the US-Loop continues choosing another given literal.

The US-Loop is a sub-procedure inside the Inst-Gen-Eq-Loop and interleaved with it in a fair way such that neither process has to wait for termination of the other process. All clauses from inconsistent subsets of their selected literals have to be instantiated, therefore the US-Loop continues after having found a contradiction. If the US-passive set in the US-Loop is empty, the Inst-Gen-Eq-Loop continues and only if in both processes the sets of US-passive literals and Inst-passive clauses, respectively, are empty, iProver-Eq is in a saturated state and returns satisfiability.

4 Instances from Unit Superposition Proofs

In this section we discuss one of the problems related to the extraction of substitutions from proofs. We keep all clauses variable disjoint and thus the selected literals are also variable disjoint.

It is well-known from standard implementations of paramodulation that for every literal all its variants should be considered to be identical in order to avoid duplicating inferences. However, in the Inst-Gen framework, variants of a literal can have different proofs, in turn resulting in different substitutions which may all be required by the Inst-Gen-Eq-Loop. Moreover, variants of a literal can occur in the same proof, potentially leading to cycles as the following example shows.

Example 2. We modify the clause set from Example 1 by replacing clause (2) with

(2') $$g(f(x), f(y)) \simeq h(z) \;\vee\; g(x, y) \not\simeq h(c)$$

The clause set remains unsatisfiable with a satisfiable ground abstraction and again the first literals in each clause are selected. To prove unsatisfiability, instances have to be generated in a way which might seem not immediately obvious.

$$\cfrac{\cfrac{\overset{(1)}{f(f(u)) \simeq f(u)} \qquad \overset{(2')}{g(f(x), f(y)) \simeq h(z)}}{g(f(u), f(y)) \simeq h(z)}\{f(u)/x\} \qquad \overset{(3)}{g(f(a), f(b)) \not\simeq h(w)}}{\cfrac{h(z) \not\simeq h(w)}{\Box}\{w/z\}}\{a/u, b/y\} \qquad (\diamond)$$

The literal $g(f(x), f(y)) \simeq h(z)$ is inferred to its variant which may seem redundant as the contradiction could already be derived from clauses (2') and (3). However, due to the substitution $\{f(u)/x\}$ in the inference, different substitutions are extracted from the proof. Indeed, only upon adding the instances of clauses (1) and (2') with the respective substitutions $\{a/u\}$ and $\{f(a)/x, b/y, w/z\}$ which are identical to clauses (5) and (6) from Example 1 the ground abstraction becomes unsatisfiable as shown there.

If we identify the literal variants of $g(f(x), f(y)) \simeq h(z)$ with each other, the proof tree in the example would collapse to a tree with the addition of a cycle on the literal. Bookkeeping information about cycles considerably complicates extraction of substitutions from proofs and approaches are needed that eliminate cycles.

Our main approach is to extract substitutions after each inference step and to use them to label the inferred literal. In order to combine all literal variants we introduce a new inference rule which merges different labels of the same literal. We also modify the inference rules from Definition 1 to accommodate labels which are merely annotations and do not influence the applicability of inferences. The conditions on the inference rules remain as in Definition 1.

Definition 2 (Labelled Unit Superposition)

$$\frac{\ell_1 : L \qquad \ell_2 : L}{\ell_1 \cup \ell_2 : L} \qquad\qquad \frac{\ell_1 : l \simeq r \qquad \ell_2 : L[l']}{\ell_1\sigma \cup \ell_2\sigma : L[r]\sigma}\;(\sigma) \qquad\qquad \frac{\ell : l \not\simeq r}{\ell\sigma : \Box}\;(\sigma)$$

In proof ($\Diamond$) above, the inferred variant of the literal $g(f(x), f(y)) \simeq h(z)$ has a different label from the variant used as a premise. By first merging the labels of both variants and subsequently deriving the contradiction, we obtain the instances of (1), (2') and (3) from proof ($\Diamond$) as well as the instances from (2') and (3) alone. We now do not need to trace a proof tree that potentially contains cycles, the necessary substitutions to generate instances can be read from the label of the contradiction.

For Example 2, we can choose a second approach orthogonal to labelling to tackle cycles by generating instances. We instantiate clause (2') with the substitution $\{f(u)/x\}$ from the cycle to clause (2) from Example 1 and can prove unsatisfiability as shown there. The cyclic proof ($\Diamond$) becomes redundant and the relevant substitution for clause (2) leading to its instance (6) can instead be extracted from the proof ($\star$). Although this method of unfolding cycles by eagerly instantiating clauses seems simple and yet powerful, it becomes much more involved when the substitution in the cycle is not proper, i.e. no variable is instantiated to a term, e.g., $\{y/x, y/z\}$. For such non-proper cycles the labelled approach is advantageous and we are investigating the benefits of combining both approaches.

5 Features of the Implementation

iProver-Eq is implemented in the functional language OCaml and uses MiniSat, CVC3 and Z3 as ground (SAT/SMT) solvers via their C/C++ APIs. It processes input in TPTP format and uses the E prover[1] for clausification of non-CNF problems. We briefly mention the most significant features of the implementation, some of which have already been present in the iProver system and were adapted or extended.

Passive Clauses/Literals. Both the Inst-passive set of clauses and the US-passive set of literals are maintained in the form of priority queues that allow user-configurable heuristics to prefer promising clauses and literals.

Dismatching Constraints. All clauses are annotated with dismatching constraints that make redundancy due to common ground instances between a clause and its instances explicit. Thus we can block redundant instantiations in the Inst-Gen-Eq-Loop and most crucially redundant proofs in the US-Loop.

Demodulation. In addition to superposition inferences, the US-Loop simplifies literals with demodulation inferences with orientable equations obtained from unit clauses.

[1] `http://www.eprover.org`

Indexing. Several unification indexes, implemented as non-perfect discrimination trees, make the forward and backward search for unifiable subterms for unit superposition and for matching subterms for demodulation efficient.

Global subsumption. iProver-Eq makes use of a global subsumption algorithm for simplifying both ground and non-ground clauses using the ground solver similar to the way it is done in iProver. It also integrates the resolution prover from iProver to obtain short clauses which are propagated to the ground solver and in turn enhance global subsumption.

6 Evaluation

iProver-Eq[2] is still in an early stage of development which has not been focused on efficiency issues yet. We have evaluated the current version of iProver-Eq which integrates CVC3 as its ground solver, on the standard TPTP v4.0.1 benchmark library. Running on Intel Xeon Quad Core machines with 2.33GHz and 3GB of memory, 5004 out of the 13783 problems are solved within 60 seconds. These include three problems that are not known to be solved by any other theorem prover. The success in 1621 problems is due to the equational reasoning and iProver did not succeed on them with the previous axiomatic handling of equations using CVC3 as a SAT solver.[3]

At the moment the core US component taken as a stand-alone reasoner for unit equations is not as efficient as dedicated superposition-based provers. As an obvious next step we are working on strengthening the US component, however, as we have demonstrated in Section 4, not all techniques from state-of-the-art reasoners can be straightforwardly adapted due to the requirement to generate all relevant instances.

References

1. Barrett, C.W., Tinelli, C.: CVC3. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 298–302. Springer, Heidelberg (2007)
2. Baumgartner, P.: Logical Engineering with Instance-Based Methods. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 404–409. Springer, Heidelberg (2007)
3. Eén, N., Sörensson, N.: An Extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)
4. Ganzinger, H., Korovin, K.: New Directions in Instantiation-Based Theorem Proving. In: LICS 2003, pp. 55–64. IEEE, Los Alamitos (2003)
5. Ganzinger, H., Korovin, K.: Integrating Equational Reasoning into Instantiation-Based Theorem Proving. In: Marcinkowski, J., Tarlecki, A. (eds.) CSL 2004. LNCS, vol. 3210, pp. 71–84. Springer, Heidelberg (2004)
6. Korovin, K.: iProver - An Instantiation-Based Theorem Prover for First-Order Logic (System Description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 292–298. Springer, Heidelberg (2008)
7. 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)
8. Nieuwenhuis, R., Rubio, A.: Paramodulation-based theorem proving. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, Elsevier, Amsterdam (1999)

[2] Available from `http://www.cs.man.ac.uk/~korovink/iprover`

[3] In SAT solving, CVC3 has an overhead due to its theory handing. For non-equational problems, one should return to the more efficient MiniSat which we did not do in the experiments.

Classical Logic with Partial Functions

Hans de Nivelle

Instytut Informatyki, University of Wrocław, Poland

Abstract. We introduce a semantics for classical logic with partial functions. We believe that the semantics is natural. When a formula contains a subterm in which a function is applied outside of its domain, our semantics ensures that the formula has no truth-value, so that it cannot be used for reasoning. The semantics relies on order of formulas. In this way, it is able to ensure that functions and predicates are properly declared before they are used. We define a sequent calculus for the semantics, and prove that this calculus is sound and complete for the semantics.

1 Introduction

Partial functions occur frequently in mathematics and programming. In high-school, one is taught that one 'should not divide by zero'. Similarly, one is taught that log(0) and $\tan \frac{\pi}{2}$ 'do not exist'. In programming, partial functions are even more abundant. A pointer can only be dereferenced if it is not the null-pointer. A vector has only a first element if it is non-empty. A file can only be read from if it is in a good state.

One approach to partial functions is what is called *the traditional approach to partial functions* in [5] and [6]: In this approach, **(1)** variables and constants are always defined, and **(2)** formulas are always true or false. Atoms (including equalities) containing undefined subterms are always false. Although the traditional approach takes partiality serious, it does not fit with our view that ill-typed formulas should not be propositions at all. (Because no assumptions should be made about programs containing undefined values.)

For this reason, many authors ([1,8]) have taken an approach based on three-valued Kleene logic. Three-valued logic is obtained by introducing an extra value $\mathbf{u}$, which is the truth-value for undefined propositions. It is assumed that the truth values are ordered as $\mathbf{f} < \mathbf{u} < \mathbf{t}$. Using this order, $P \vee Q$ can be defined as $\text{MAX}_<(P, Q)$. Negation can be defined from $[\mathbf{f} \Rightarrow \mathbf{t},\ \mathbf{u} \Rightarrow \mathbf{u},\ \mathbf{t} \Rightarrow \mathbf{f}]$. Other operators can be introduced through standard equivalences. The well-definedness approach of [2,9] is closely related to Kleene logic, although at first it may appear different, due to its proof-theoretic motivation. In the WD-approach, one has to prove that a formula is well-defined before it is used. It can be seen from the definitions in [9] that a formula is not well-defined iff it would take the value $\mathbf{u}$ in Kleene logic.

Kleene logic (and the WD-approach) are closer to our intuitions, but there is a difference: In our view, ill-defined formulas are not *unknown*, but *errors* after which nothing can be assumed. The justification for setting $\mathbf{t} \vee \mathbf{u} = \mathbf{t}$, is the fact that whichever value **u** will take, the **t** ensures that the disjunction will be true. In our philosophy, nothing should be assumed about error values.

In addition to this philosophy, our system has some other, teachnical features that we believe may be useful: Preconditions of partial functions and types are treated in a unified way, and they are treated inside the logic itself, not by an external type system. This ensures that the logic does not have any built-in restrictions on type systems with which it is used. Both the type and the preconditions of a partial function can be expressed by ordinary formulas, as for example in $\forall x, y \ \mathrm{Nat}(x) \wedge \mathrm{Nat}(y) \rightarrow \mathrm{Nat}(x+y)$. Subtraction can be specified as a partial function by the formula: $\forall x, y \ \mathrm{Nat}(x) \wedge \mathrm{Nat}(y) \wedge x \geq y \rightarrow \mathrm{Nat}(x-y)$.

In our setting, partial functions are functions that sometimes have results about which nothing can be assumed. If the specification of the function requires that it throws exceptions, then something is assumed about the result. We view exceptions as a form of polymorphism, which is different from partiality. Our system is flexible enough to handle both polymorphism and partiality.

In order to connect preconditions to formulas, we introduce two binary operators: The first operator is the *lazy implication* operator $[A]B$, the second operator is the *lazy conjunction* operator $\langle A \rangle B$. We call the operators 'lazy' because they do not look at the second argument when the first argument is false. (Similar to `&&` and `||` in `C`). Because of this, B needs to be a proposition only when A is true, so that truth of A can be assumed when proving that B is a proposition. In the strict operators $A \rightarrow B$ and $A \wedge B$, the second argument must be a proposition independent of A.

We now introduce the syntax and semantics of our system, which we will call PCL (*Partial Classical Logic*).

Definition 1. *The set of* terms *of PCL (*partial classical logic*) is recursively defined by the following rule: If $t_1, \ldots, t_n$ are terms (with $n \geq 0$), and f is a function symbol with arity n, then $f(t_1, \ldots, t_n)$ is also a term. We call function symbols with arity* 0 constants *or* variables*, dependant on how they are used.*

Using the set of terms, we define the set of formulas of PCL recursively as follows:

- *$\bot$ and $\top$ are formulas.*
- *Every term A is a formula. We will call formulas of this form* atoms*.*
- *If t_1, t_2 are terms, then $t_1 = t_2$ is a formula.*
- *If A is a formula, then $\neg A$ is a formula.*
- *If A and B are formulas, then $A \wedge B$, $A \vee B$, $A \rightarrow B$, and $A \leftrightarrow B$ are formulas.*
- *If A and B are formulas, then $[A]B$ and $\langle A \rangle B$ are formulas.*
- *If x is a variable, A is a formula, then $\forall x \ A$ and $\exists x \ A$ are formulas.*
- *If A is a formula, then* $\mathrm{Prop}(A)$ *is a formula.*

The intuitive meaning of $\mathrm{Prop}(F)$ is 'F is a formula'. The logic is set up in such a way, that type correctness of formulas and terms must be proven within the calculus. As a consequence, there is no syntactic distinction between formulas and terms in Definition 1.

We will now introduce a lattice on non-truth values. It will be used in Definition 3, to ensure that logical operators behave in a predictable way when their arguments are not valid propositions. This has the advantage that many meta-properties of the logic can be formulated as equivalences. For example, without the lattice on non-truth values, $I(A) = I(A \wedge A)$ would not hold as equality. This would have no effect on the provable formulas, because the equality would still hold for the truth values. The lattice is only a trick to make the meta-properties nicer. We do not intend to use the lattice for abstraction, as is proposed in [7].

Definition 2. *Let S be a set. A relation $\sqsubseteq$ is called* a partial order *if it meets the following requirements:* **(1)** *For all $s \in S$, $s \sqsubseteq s$.* **(2)** *For all s_1, s_2, s_3, $s_1 \sqsubseteq s_2 \wedge s_2 \sqsubseteq s_3 \Rightarrow s_1 \sqsubseteq s_3$.* **(3)** *For all s_1, s_2, $s_1 \sqsubseteq s_2 \wedge s_2 \sqsubseteq s_1 \Rightarrow s_1 = s_2$. Let S' be a subset of S. We call $s \in S$ a* lower bound *of S' if for all $s' \in S'$, $s \sqsubseteq s'$. We call s a greatest lower bound of S' if s is a lower bound of S', and for every lower bound $\hat{s}$ of S', we have $\hat{s} \sqsubseteq s$.*

We write $\sqcap S'$ for the greatest lower bound of S', if it exists. If S' is finite, we write $s_1 \sqcap s_2 \sqcap \cdots \sqcap s_n$ instead of $\sqcap\{s_1, s_2, \ldots, s_n\}$.

It is easily checked that the greatest lower bound is unique if it exists.

Definition 3. *An interpretation $I = (D, \mathbf{f}, \mathbf{t}, \sqsubseteq, [\ ])$ is defined by*

- *A domain D.*
- *Two distinct truth constants $\mathbf{f}$ and $\mathbf{t}$, such that both of $\mathbf{f}, \mathbf{t} \in D$,*
- *A partial order $\sqsubseteq$ on $D \backslash \{\mathbf{f}, \mathbf{t}\}$, s.t. every non-empty $D' \subseteq D \backslash \{\mathbf{f}, \mathbf{t}\}$ has a greatest lower bound $\sqcap D'$, which is in $D \backslash \{\mathbf{f}, \mathbf{t}\}$.*
- *a function $[\ ]$ that interprets function symbols as follows: If f is a function symbol with arity n, then $[f]$ is a total function from D^n to D.*

As said above, the role of the partial order $\sqsubseteq$ is to obtain predictable behaviour of the logical operators when they are applied on non-Boolean objects.

Definition 4. *Let $I = (D, \mathbf{f}, \mathbf{t}, \sqsubseteq, [\ ])$ be an interpretation. We recursively define the* interpretation *$I(F)$ of a formula F as follows:*

- *$I(\bot) = \mathbf{f}$, $I(\top) = \mathbf{t}$.*
- *If $F = [f]$, then $I(\ f(t_1, \ldots, t_n)\ ) = F(I(t_1), \ldots, I(t_n))$.*
- *If $I(t_1) = I(t_2)$, then $I(t_1 = t_2) = \mathbf{t}$. Otherwise, $I(t_1 = t_2) = \mathbf{f}$.*
- *If $I(A) = \mathbf{t}$, then $I(\neg A) = \mathbf{f}$. If $I(A) = \mathbf{f}$, then $I(\neg A) = \mathbf{t}$. Otherwise $I(\neg A) = I(A)$.*
- *We characterize the strict binary operators:*
  - *If $I(A) \in \{\mathbf{f}, \mathbf{t}\}$, and $I(B) \notin \{\mathbf{f}, \mathbf{t}\}$, then $I(A \wedge B) = I(A \vee B) = I(A \rightarrow B) = I(A \leftrightarrow B) = I(B)$.*
  - *If $I(A) \notin \{\mathbf{f}, \mathbf{t}\}$, and $I(B) \in \{\mathbf{f}, \mathbf{t}\}$, then $I(A \wedge B) = I(A \vee B) = I(A \rightarrow B) = I(A \leftrightarrow B) = I(A)$.*
  - *If both $I(A), I(B) \notin \{\mathbf{f}, \mathbf{t}\}$, then $I(A \wedge B) = I(A \vee B) = I(A \rightarrow B) = I(A \leftrightarrow B) = I(A) \sqcap I(B)$.*

- *If both of* $I(A), I(B) \in \{\mathbf{f}, \mathbf{t}\}$, *then* $\wedge, \vee, \rightarrow, \leftrightarrow$ *are characterized by the following (standard) truth table:*

$I(A)$	$I(B)$	$I(A \wedge B)$	$I(A \vee B)$	$I(A \rightarrow B)$	$I(A \leftrightarrow B)$
f	**f**	**f**	**f**	**t**	**t**
f	**t**	**f**	**t**	**t**	**f**
t	**f**	**f**	**t**	**f**	**f**
t	**t**	**t**	**t**	**t**	**t**

– *We characterize the lazy binary operators:*
  - *If* $I(A) = \mathbf{f}$, *then* $I(\ [A]B\ ) = \mathbf{t}$, *and* $I(\ \langle A \rangle B\ ) = \mathbf{f}$.
  - *If* $I(A) \not\in \{\mathbf{f}, \mathbf{t}\}$, *and* $I(B) \in \{\mathbf{f}, \mathbf{t}\}$, *then* $I(\ [A]B\ ) = I(\ \langle A \rangle B\ ) = I(A)$.
  - *If both* $I(A), I(B) \not\in \{\mathbf{f}, \mathbf{t}\}$, *then* $I(\ [A]B\ ) = I(\ \langle A \rangle B\ ) = I(A) \sqcap I(B)$.
  - *If* $I(A) = \mathbf{t}$, *then* $I(\ [A]B\ ) = I(\ \langle A \rangle B\ ) = I(B)$.

– *Next come the quantifiers: Let* x *be some variable. Let* F *be a formula. Let* $R = \{\ I_d^x(d) \mid d \in D\}$, *where* I_d^x *is defined as usual.*
  - *If* $R \not\subseteq \{\mathbf{f}, \mathbf{t}\}$, *then* $I(\ \forall x\ F\ ) = I(\ \exists x\ F\ ) = \sqcap(R \backslash \{\mathbf{f}, \mathbf{t}\})$.
  - *If* $R = \{\mathbf{f}\}$, *then* $I(\ \forall x\ F\ ) = I(\ \exists x\ F\ ) = \mathbf{f}$.
  - *If* $R = \{\mathbf{t}\}$, *then* $I(\ \forall x\ F\ ) = I(\ \exists x\ F\ ) = \mathbf{t}$.
  - *If* $R = \{\mathbf{f}, \mathbf{t}\}$, *then* $I(\ \forall x\ F\ ) = \mathbf{f}$ *and* $I(\ \exists x\ F\ ) = \mathbf{t}$.

– *It remains to characterize* Prop. *If* $I(A) \in \{\mathbf{f}, \mathbf{t}\}$, *then* $I(\ \mathrm{Prop}(A)\ ) = \mathbf{t}$. *Otherwise,* $I(\ \mathrm{Prop}(A)\ ) = \mathbf{f}$.

Valid judgments will be represented by sequents.

Definition 5. *A* context *is a finite sequence of formulas* $\Gamma_1, \ldots, \Gamma_n$. *A* sequent *is an object of form* $\Gamma \vdash A$, *in which* Γ *is a context and* A *is a formula.*

We introduce two notions of validity for sequents. The first notion is the standard notion. The second, stronger notion is the notion that we will be using.

Definition 6. *Let* $\Gamma_1, \ldots, \Gamma_n \vdash A$ *be a sequent. We call* $\Gamma_1, \ldots, \Gamma_n \vdash A$ valid *if in every interpretation* $I = (D, \mathbf{f}, \mathbf{t}, \sqsubseteq, [\ ])$, *s.t.* $I(\Gamma_1) = \cdots = I(\Gamma_n) = \mathbf{t}$, *we also have* $I(A) = \mathbf{t}$. *We call the sequent* $\Gamma_1, \ldots, \Gamma_n \vdash A$ strongly valid*, if it is valid, and in addition the context* $\Gamma_1, \ldots, \Gamma_n$ *has the following property: Either for all* i *with* $1 \leq i \leq n$, *we have* $I(\Gamma_i) = \mathbf{t}$, *or for the first* i *with* $1 \leq i \leq n$ *that has* $I(\Gamma_i) \neq \mathbf{t}$, *we have* $I(\Gamma_i) = \mathbf{f}$.

The sequent $A \vdash A$ is valid, but not strongly valid. One can take an interpretation I with $I(A) = \mathbf{e}$, for some $\mathbf{e} \not\in \{\mathbf{f}, \mathbf{t}\}$. The sequent $\mathrm{Prop}(A), A \vdash A$ is strongly valid because it is valid, and if $I(A) = \mathbf{e}$, then $I(\mathrm{Prop}(A)) = \mathbf{f}$. Similarly, the sequent $\vdash A \vee \neg A$ is valid, but not strongly valid. The sequent $\mathrm{Prop}(A) \vdash A \vee \neg A$ is strongly valid.

The notion of strong validity captures the fact that functions and predicates have to be declared before they are used. If one has a context Γ_1 and a formula A, for which $\Gamma_1 \not\models \mathrm{Prop}(A)$, then there exists an interpretation I, in which $I(\Gamma_1) = \mathbf{t}$ and $I(A) \not\in \{\mathbf{f}, \mathbf{t}\}$, so that no sequent of form $\Gamma_1, A, \Gamma_2 \vdash B$ can be strongly valid. The following example illustrates declaration of partial functions, and usage of the lazy operators $\langle\ \rangle$ and $[\ ]$:

F1 $\forall x$ Prop(Nat(x)),
F2 $\forall xy$ Nat$(x) \wedge$ Nat$(y) \rightarrow$ Prop$(x \geq y)$,
F3 $\forall xy$ [Nat$(x) \wedge$ Nat(y)] $x \geq y \rightarrow$ Nat$(x - y)$,
F4 $\forall xy$ [Nat$(x) \wedge$ Nat(y)] $x \geq y \rightarrow \exists z \langle$ Nat$(z) \rangle \langle x \geq z \rangle\ x - z = y$.

In F2, the relation $\geq$ is defined on natural numbers. This can be done with standard implication $\rightarrow$ because Prop$(x \geq y)$ is always Prop by itself. In F3, subtraction $x - y$ is declared to return Nat on the condition that $x \geq y$. Here lazy implication must be used, because without Nat(x), Nat(y), $x \geq y$ would not be Prop. In F4, $\langle\ \rangle$ must be used with $\exists$ to declare z in Nat, but also to declare $x \geq z$, because otherwise $x - z$ would not be Nat.

We aim to define a sequent calculus that is able to model strong validity. It turns out that definition of this calculus is simpler when one defines a one-sided calculus, in which sequents are refuted instead of proven. The reasons for this are the following: Validity of Prop$(A) \vdash A \vee \neg A$ and the fact that the semantics is based on truth-values, suggest that PCL is essentially classical (in contract to intuitionistic). At the same time, the notion of strong validity depends on the order of formulas in the sequent. Allowing formulas to freely move from the premise to the conclusion in a sequent, which would be needed for classical $\neg$-rules, and simultaneously keeping track of the order of the formulas in the sequent, is tedious. It can be avoided by using one-sided sequents.

Definition 7. *A* one-sided sequent *is an object of form $\Gamma_1, \ldots, \Gamma_n \vdash$, in which $\Gamma_1, \ldots, \Gamma_n$ $(n \geq 0)$ is a sequence of formulas. We say that $\Gamma_1, \ldots, \Gamma_n \vdash$* fails *in an interpretation I if there is an i, $(1 \leq i \leq n)$, s.t. $I(\Gamma_i) \neq \mathbf{t}$. We will usually write 'sequent' instead of 'one-sided sequent', since it is always clear from the form which type is meant.*

We say that $\Gamma \vdash$ fails strongly *in I if there is an i, $(1 \leq i \leq n)$, s.t. $I(\Gamma_i) = \mathbf{f}$ and for all j, $(1 \leq j < i)$, $I(\Gamma_j) = \mathbf{t}$. If we want to stress that Γ_i is the first formula in Γ with $I(\Gamma_i) \neq \mathbf{t}$ (which implies that $I(\Gamma_i) = \mathbf{f}$), then we say that Γ* fails strongly at *Γ_i in I.*

We call the one-sided sequent $\Gamma \vdash$ unsatisfiable *if it fails in every interpretation. We call Γ* strongly unsatisfiable *if it fails strongly in every interpretation.*

Theorem 1. *Let $\Gamma \vdash A$ be a sequent. $\Gamma \vdash A$ is strongly valid if and only if the one-sided sequent $\Gamma, \neg A \vdash$ is strongly unsatisfiable.*

Proof. Write $\Gamma = \Gamma_1, \ldots, \Gamma_n \vdash$ with $n \geq 0$. For convenience, define $\Gamma_{n+1} := \neg A$.

Assume that $\Gamma \vdash A$ is strongly valid. We have to show that the sequent $\Gamma_1, \Gamma_2, \ldots, \Gamma_{n+1} \vdash$ is strongly unsatisfiable. Let I be an arbitrary interpretation. We have to show that there exists an i, $(1 \leq i \leq n+1)$ with property $\Phi(i)$, where $\Phi(i)$ is the property that $I(\Gamma_i) = \mathbf{f}$, and for all j, $(1 \leq j < i)$, $I(\Gamma_j) = \mathbf{t}$. We distinguish two cases:

- If for all i, $1 \leq i \leq n$, $I(\Gamma_i) = \mathbf{t}$, then it follows from validity of $\Gamma \vdash A$ that $I(A) = \mathbf{t}$, so that $I(\neg A) = \mathbf{f}$. Since $\neg A = \Gamma_{n+1}$, we have $\Phi(n+1)$.
- If there is an i with $1 \leq i \leq n$, s.t. $I(\Gamma_i) \neq \mathbf{t}$, then by strong validity of $\Gamma \vdash A$ (See Definition 6), we have $I(\Gamma_i) = \mathbf{f}$ for the first i with $I(\Gamma_i) \neq \mathbf{t}$. It follows that we have $\Phi(i)$.

$$\begin{array}{llll}
\neg\bot & \Rightarrow \top, & \neg\top & \Rightarrow \bot \\
\neg\neg A & \Rightarrow A, & & \\
\neg(\ \langle A\rangle B\ ) & \Rightarrow [A]\neg B, & \neg(\ [A]B\ ) & \Rightarrow \langle A\rangle\neg B \\
\neg(A \wedge B) & \Rightarrow \neg A \vee \neg B, & \neg(A \vee B) & \Rightarrow \neg A \wedge \neg B \\
\neg(A \rightarrow B) & \Rightarrow A \wedge \neg B, & \neg(A \leftrightarrow B) & \Rightarrow A \leftrightarrow \neg B \\
\neg\forall x\ F, & \Rightarrow \exists x\ \neg F & \neg\exists x\ F & \Rightarrow \forall x\ \neg F
\end{array}$$

Fig. 1. Reduction Rules for $\neg$

In order to show the other direction, assume that $\Gamma_1, \ldots, \Gamma_n, \Gamma_{n+1} \vdash$ is strongly unsatisfiable. We first show that $\Gamma_1, \ldots, \Gamma_n \vdash A$ is valid. Let I be an interpretation. Assume that $I(\Gamma_1) = I(\Gamma_2) = \cdots = I(\Gamma_n) = \mathbf{t}$. It follows from the strong unsatisfiability of $\Gamma_1, \ldots, \Gamma_n, \Gamma_{n+1} \vdash$ that $I(\Gamma_{n+1}) = \mathbf{f}$, so that $I(A) = \mathbf{t}$. Next we show the additional property that makes $\Gamma_1, \ldots, \Gamma_n \vdash A$ strongly valid. If no i has $I(\Gamma_i) \neq \mathbf{t}$, then we are done. Otherwise, let i be the first position where $I(\Gamma_i) \neq \mathbf{t}$. If we would have $I(\Gamma_i) \neq \mathbf{f}$, then this would contradict strong unsatisfiability of the sequent $\Gamma_1, \ldots, \Gamma_n, \Gamma_{n+1} \vdash$, so that the proof is complete.

Using Theorem 1, the conclusion of a sequent can be moved to the left hand side, after which it can be treated in the same way as the other premises. This has the advantage that one can delete half of the rules from the sequent calculus, and it avoids the burden of keeping track of the order of formulas spread between the premises and the conclusions. In order to further simplify the calculus, we use the reduction rules in Figure 1 and Figure 2. Figure 1 contains rules for pushing negation inwards, while Figure 2 contains rules for pushing Prop inwards. Most rules in Figure 1 look familiar, but their validity still needs to be checked in the context of PCL. It can be checked (by case analysis) that for every interpretation I, for each rule $A \Rightarrow B$ in Figure 1 or Figure 2, we have $I(A) = I(B)$, so that the equivalences can be freely used in proofs. Figure 1 and Figure 2 ensure that $\neg$ or Prop never needs to be the main operator of a formula. The only cases where Prop and $\neg$ cannot be eliminated are in formulas of form $\phi_1(\phi_2(A))$, where A is an atom, ϕ_2 is either Prop or nothing, and ϕ_1 is either $\neg$ or nothing. Such formulas play the same role as literals in first-order logic. One can either simplify the sequent completely before proof search, or apply the rules 'lazily,' i.e. only when $\neg$ or Prop stands in the way of a rule application.
Figure 3 contains the rules of the sequent calculus Seq_{PCL}. Most rules probably look familiar, but there are pitfalls. For example, the rule for $\wedge$-introduction would be unsound if the second premise would be removed. It would then be possible that in some interpretation $I = (D, \mathbf{f}, \mathbf{t}, \sqsubseteq, [\,])$, one has $I(\Gamma_1) = \mathbf{t},\ I(A) = \mathbf{f}$, and $I(B) \notin \{\mathbf{f}, \mathbf{t}\}$. In that case the left premise would fail strongly, while the conclusion would fail only weakly. Similarly, the rule for $\forall$-introduction would be unsound if one would not keep a copy of $\forall x\ P(x)$ in the premise before $P(t)$. It could happen that in some interpretation I, $\ I(P(t)) = \mathbf{t}$, while at the same time $I(\forall x\ P(x)) \notin \{\mathbf{f}, \mathbf{t}\}$.

If one would remove the A from the second premise in $[\ ]$-introduction, the rule would still be sound, but become too weak for completeness. The problem would show up when $\text{Prop}(B)$ depends on A.

$$
\begin{array}{lll}
\mathrm{Prop}(\top) & \Rightarrow \top & \\
\mathrm{Prop}(\bot) & \Rightarrow \top & \\
\mathrm{Prop}(\neg A) & \Rightarrow \mathrm{Prop}(A) & \\
\mathrm{Prop}(\mathrm{Prop}(A)) & \Rightarrow \top & \\
 & & \\
\mathrm{Prop}(A \wedge B) & \Rightarrow \mathrm{Prop}(A) \wedge \mathrm{Prop}(B) & (\text{or } \langle\ \mathrm{Prop}(A)\ \rangle\ \mathrm{Prop}(B)\ ) \\
\mathrm{Prop}(A \vee B) & \Rightarrow \mathrm{Prop}(A) \wedge \mathrm{Prop}(B) & (\text{or } \langle\ \mathrm{Prop}(A)\ \rangle\ \mathrm{Prop}(B)\ ) \\
\mathrm{Prop}(A \rightarrow B) & \Rightarrow \mathrm{Prop}(A) \wedge \mathrm{Prop}(B) & (\text{or } \langle\ \mathrm{Prop}(A)\ \rangle\ \mathrm{Prop}(B)\ ) \\
\mathrm{Prop}(A \leftrightarrow B) & \Rightarrow \mathrm{Prop}(A) \wedge \mathrm{Prop}(B) & (\text{or } \langle\ \mathrm{Prop}(A)\ \rangle\ \mathrm{Prop}(B)\ ) \\
 & & \\
\mathrm{Prop}(\ \langle A\rangle B\ ) & \Rightarrow \langle\ \mathrm{Prop}(A)\ \rangle(\ A \rightarrow \mathrm{Prop}(B)\ ) & \\
\mathrm{Prop}(\ [A]B\ ) & \Rightarrow \langle\ \mathrm{Prop}(A)\ \rangle(\ A \rightarrow \mathrm{Prop}(B)\ ) & \\
 & & \\
\mathrm{Prop}(\forall x\ F) & \Rightarrow \forall x\ \mathrm{Prop}(F) & \\
\mathrm{Prop}(\exists x\ F) & \Rightarrow \forall x\ \mathrm{Prop}(F) & \\
 & & \\
\mathrm{Prop}(t_1 = t_2) & \Rightarrow \top &
\end{array}
$$

Fig. 2. Reduction Rules for Prop

In contrast to standard first-order logic, a sequent of form $\Gamma, A, \neg A \vdash$ is not automatically an axiom. It is possible that $\Gamma \vdash$ fails weakly, or $I(\Gamma) = \mathbf{t}$ and $I(A) \notin \{\mathbf{f}, \mathbf{t}\}$. Both cases are covered by requiring the additional sequent $\Gamma,\ \neg\mathrm{Prop}(A) \vdash$.

We will prove soundness of the rules for $\langle\ \rangle$, $\vee$, and $\exists$. Most of the other rules can be reduced to $\langle\ \rangle$ and $\vee$, by using the equivalences in Figure 4. The remaining rules can be checked by case analysis.

Theorem 2. *Let $\Gamma_{\langle A\rangle B}$ be a sequent of form $\Gamma_1, \ldots, \Gamma_m,\ \langle A\rangle B,\ \Gamma'_1, \ldots, \Gamma'_n \vdash$. Let $\Gamma_{A,B}$ be the sequent $\Gamma_1, \ldots, \Gamma_m, A, B, \Gamma'_1, \ldots, \Gamma'_n \vdash$. Let I be an interpretation. Then $\Gamma_{\langle A\rangle B}$ fails strongly in I iff $\Gamma_{A,B}$ fails strongly in I.*

Proof. Assume that $\Gamma_{\langle A\rangle B}$ fails strongly in I. This means that the first formula F in $\Gamma_{\langle A\rangle B}$, for which $I(F) \neq \mathbf{t}$, has $I(F) = \mathbf{f}$.

– If F is among the Γ_i, then it is immediate that $\Gamma_{A,B}$ fails strongly in I.
– If $F = \langle A\rangle B$, then either $I(A) = \mathbf{f}$, or ($I(A) = \mathbf{t}$ and $I(B) = \mathbf{f}$). Since $I(\Gamma_1) = \cdots = I(\Gamma_m) = \mathbf{t}$, in both cases $\Gamma_{A,B}$ fails strongly in I.
– If F is among the Γ'_j, then F also occurs in $\Gamma_{A,B}$. We know that $I(\Gamma_1) = \cdots = I(\Gamma_m) = \mathbf{t}$. From the fact that $I(\ \langle A\rangle B\ ) = \mathbf{t}$, follows that $I(A) = I(B) = \mathbf{t}$. Since we assumed that $I(\Gamma'_1) = \cdots = I(\Gamma'_{j-1}) = \mathbf{t}$, it follows that F is the first formula in $\Gamma_{A,B}$ for which $I(F) \neq \mathbf{t}$. Since $I(F) = \mathbf{f}$, we know that $\Gamma_{A,B}$ fails strongly in I.

For the other direction, assume that $\Gamma_{A,B}$ fails strongly in I. Let F be the first formula in $\Gamma_{A,B}$ for which $I(F) \neq \mathbf{t}$. We have $I(F) = \mathbf{f}$.

– If F is among the Γ_i, then it is immediate that $\Gamma_{\langle A\rangle B}$ fails strongly in I.
– If $F = A$, then $I(\ \langle A\rangle B\ ) = \mathbf{f}$, and $I(\Gamma_1) = \cdots = I(\Gamma_m) = \mathbf{t}$, so that $\Gamma_{\langle A\rangle B}$ fails strongly in I at formula A.

Rules for $\langle\ \rangle$ and $\vee$

$$\frac{\Gamma_1, A, B, \Gamma_2 \vdash}{\Gamma_1,\ \langle A \rangle\ B,\ \Gamma_2 \vdash} \qquad \frac{\Gamma_1, A, \Gamma_2 \vdash \qquad \Gamma_1, B, \Gamma_2 \vdash}{\Gamma_1,\ A \vee B,\ \Gamma_2 \vdash}$$

Rules for $\wedge$ and $[\ ]$

$$\frac{\Gamma_1, A, B, \Gamma_2 \vdash \qquad \Gamma_1, B, A, \Gamma_2 \vdash}{\Gamma_1,\ A \wedge B,\ \Gamma_2 \vdash} \qquad \frac{\Gamma_1,\ \neg A,\ \Gamma_2 \vdash \qquad \Gamma_1, A, B, \Gamma_2 \vdash}{\Gamma_1,\ [A]B,\ \Gamma_2 \vdash}$$

Rules for $\rightarrow$ and $\leftrightarrow$

$$\frac{\Gamma_1,\ \neg A,\ \Gamma_2 \qquad \Gamma_1, B, \Gamma_2 \vdash}{\Gamma_1,\ A \rightarrow B,\ \Gamma_2 \vdash} \qquad \frac{\Gamma_1,\ A, B,\ \Gamma_2 \vdash \qquad \Gamma_1,\ \neg B, \neg A,\ \Gamma_2 \vdash}{\Gamma_1,\ A \leftrightarrow B,\ \Gamma_2 \vdash}$$

Rules for $\forall$ and $\exists$

$$\frac{\Gamma_1,\ \forall x\ P(x),\ P(t),\ \Gamma_2 \vdash}{\Gamma_1,\ \forall x\ P(x),\ \Gamma_2 \vdash} \qquad \frac{\Gamma_1,\ P(x),\ \Gamma_2 \vdash}{\Gamma_1,\ \exists x\ P(x),\ \Gamma_2 \vdash}$$

(In the $\forall$-rule, t must be a term. In the $\exists$-rule x must be not free in Γ_1 or Γ_2.)

Equivalence If $A \Rightarrow B$ is an instance of one of the rules in figure 2 or figure 1, then the following derivation is possible:

$$\frac{\Gamma_1, B, \Gamma_2 \vdash}{\Gamma_1, A, \Gamma_2 \vdash}$$

Axioms

$$\frac{\Gamma,\ \neg\mathrm{Prop}(A) \vdash}{\Gamma, A, \neg A \vdash} \qquad \frac{\Gamma,\ \neg\mathrm{Prop}(A) \vdash}{\Gamma, \neg A, A \vdash} \qquad \frac{}{\bot \vdash}$$

Weakening

$$\frac{\Gamma_1, \neg\mathrm{Prop}(A) \vdash \qquad \Gamma_1, \Gamma_2 \vdash}{\Gamma_1, A, \Gamma_2 \vdash} \qquad \frac{\Gamma \vdash}{\Gamma, A \vdash} \qquad \frac{\Gamma_1, \Gamma_2 \vdash}{\Gamma_1, \top, \Gamma_2 \vdash}$$

(In the first two rules, A can be an arbitrary formula.)

Contraction, Cut

$$\frac{\Gamma_1, A, \Gamma_2, A, \Gamma_3 \vdash}{\Gamma_1, A, \Gamma_2, \Gamma_3 \vdash} \qquad \frac{\Gamma_1,\ \neg A \vdash \qquad \Gamma_1, A, \Gamma_2 \vdash}{\Gamma_1, \Gamma_2 \vdash}$$

(A can be an arbitrary formula.)

Equality

$$\frac{}{\mathrm{Prop}(A), A,\ t_1 = u_1, \ldots, t_n = u_n, \neg A' \vdash} \qquad \frac{}{t_1 = u_1, \ldots, t_n = u_n, t \neq u \vdash}$$

In the first axiom, it must be the case that $A, t_1 = u_1, \ldots, t_n = u_n \models A'$ in the standard theory of equality. In the second axiom, it must be the case that $t_1 = u_1, \ldots, t_n = u_n \vdash t = u$ in the standard theory of equality.

Fig. 3. Rules of Seq_{PCL} :

$\wedge$	$A \wedge B \ \Rightarrow (\ \langle A \rangle B\ ) \vee (\ \langle B \rangle A\ )$	$\top$	$\langle A \rangle \top \ \Rightarrow A$
$[\ ]$	$[A]B \ \Rightarrow \neg A \vee (\ \langle A \rangle B\ )$	$\top$	$\langle \top \rangle A \ \Rightarrow A$
$\rightarrow$	$A \rightarrow B \Rightarrow \neg A \vee B$		
$\leftrightarrow$	$A \leftrightarrow B \Rightarrow (\ \langle A \rangle B\ ) \vee (\ \langle \neg A \rangle \neg B\ )$	$\forall$	$\forall x\ P(x) \Rightarrow \langle\ \forall x\ P(x)\ \rangle\ P(t)$

Fig. 4. Reduction of remaining rules to $\langle\ \rangle$ and $\vee$

- If $F = B$, then we know that $I(A) = \mathbf{t}$, so that $I(\ \langle A \rangle B\ ) = \mathbf{f}$. Since $I(\Gamma_1) = \cdots I(\Gamma_m) = \mathbf{t}$, it follows that $\Gamma_{\langle A \rangle B}$ fails strongly in I at formula B.
- If F is among the Γ'_j, then F also occurs in $\Gamma_{\langle A \rangle B}$, so that it is sufficient to show that there is no formula F' before F in $\Gamma_{\langle A \rangle B}$, s.t. $I(F') \neq \mathbf{t}$. The only candidate is $\langle A \rangle B$, because all other formulas were copied from $\Gamma_{A,B}$. But since we know that $I(A) = I(B) = \mathbf{t}$, it follows that $I(\ \langle A \rangle B\ ) = \mathbf{t}$.

Theorem 3. *Let $\Gamma_{A \vee B}$ be a sequent of form $\Gamma_1, \ldots, \Gamma_m, A \vee B, \Gamma'_1, \ldots, \Gamma'_n \vdash$. Let $\Gamma_A = \Gamma_1, \ldots, \Gamma_m, A, \Gamma'_1, \ldots, \Gamma'_n \vdash$, and let $\Gamma_B = \Gamma_1, \ldots, \Gamma_m, B, \Gamma'_1, \ldots, \Gamma'_n \vdash$. Let I be an interpretation. The sequent $\Gamma_{A \vee B}$ fails strongly in I iff both of Γ_A and Γ_B fail strongly in I.*

Proof. Because of space restriction, we do some handwaving. Using Theorem 2 and Figure 4, we can collapse $\Gamma_1, \ldots, \Gamma_m$ and $\Gamma'_1, \ldots, \Gamma'_n$ into single formulas of form $C_1 = \langle \Gamma_1 \rangle \cdots \langle \Gamma_m \rangle \top$ and $C_2 = \langle \Gamma'_1 \rangle \cdots \langle \Gamma'_n \rangle \top$.

After the replacement, the proof reduces to showing that $C_1, A \vee B, C_2 \vdash$ fails strongly in I iff both of $C_1, A, C_2 \vdash$ and $C_1, B, C_2 \vdash$ fail strongly in I. This can be checked by case analysis. Each of C_1, A, B, C_2 can be either $\mathbf{f}, \mathbf{t}$ or $\not\in \{\mathbf{f}, \mathbf{t}\}$. This results in $3^4 = 81$ cases, which can be checked. [1]

Theorem 4. *Let $\Gamma_\exists$ be a sequent of form $\Gamma_1, \ldots, \Gamma_m,\ \exists x\ P(x),\ \Gamma'_1, \ldots, \Gamma'_n \vdash$. Let Γ_x be the sequent $\Gamma_x = \Gamma_1, \ldots, \Gamma_m,\ P(x),\ \Gamma'_1, \ldots, \Gamma'_n \vdash$. Assume that x is not free in any of the formulas $\Gamma_1, \ldots, \Gamma_m, \Gamma'_1, \ldots, \Gamma'_n$.*

Let $I = (D, \mathbf{f}, \mathbf{t}, \sqsubseteq, [\ ])$ be an arbitrary interpretation. Then $\Gamma_\exists$ fails strongly in I iff for every $d \in D$, the sequent Γ_x fails strongly in $I^x_d = (D, \mathbf{f}, \mathbf{t}, \sqsubseteq, [\ ]^x_d)$.

Proof. In the proof, we make use of the fact that $I(\Gamma_i) = I^x_d(\Gamma_i)$ and $I(\Gamma'_j) = I^x_d(\Gamma'_j)$, because x is not free in Γ_i, Γ'_j.

Assume that $\Gamma_\exists$ fails strongly in I. This means that for the first formula F in $\Gamma_\exists$ with $I(F) \neq \mathbf{t}$, one has $I(F) = \mathbf{f}$.

- If F is one of the Γ_i, then $I(\Gamma_i) = I^x_d(\Gamma_i) = \mathbf{f}$, and for all i', $(1 \leq i' \leq i)$, $I(\Gamma_{i'}) = I^x_d(\Gamma_{i'}) = \mathbf{t}$, so that Γ_x fails strongly at Γ_i in every I^x_d.
- If F is $\exists x\ P(x)$, then the fact that $I(\ \exists x\ P(x)\ ) = \mathbf{f}$, implies that for every $d \in D$, $I^x_d(\ P(x)\ ) = \mathbf{f}$. Since for every i, $I(\Gamma_i) = I^x_d(\Gamma_i) = \mathbf{t}$, it follows that Γ_x fails strongly at formula $P(x)$ in every interpretation I^x_d.
- If F is one of the Γ'_j, then we know that for every $d \in D$, $I(\Gamma_1) = I^x_d(\Gamma_1) = \cdots = I(\Gamma_m) = I^x_d(\Gamma_m) = \mathbf{t}$. It can be seen from Definition 4 that

[1] The cases have been checked by a computer program, together with all cases for the reductions in Figure 1, 2 and 4.

$I(\ \exists x\ P(x)\ ) = \mathbf{t}$ implies that for every $d \in D$, either $I_d^x(\ P(x)\ ) = \mathbf{f}$, or $I_d^x(\ P(x)\ ) = \mathbf{t}$. If $I_d^x(\ P(x)\ ) = \mathbf{f}$, then Γ_x strongly fails at $P(x)$ in I_d^x. Otherwise, we have $I_d^x(\ P(x)\ ) = \mathbf{t}$ and $I(\Gamma_1') = I_d^x(\Gamma_1') = \cdots = I(\Gamma_{j-1}') = I_d^x(\Gamma_{j-1}') = \mathbf{t},$ and $I(\Gamma_j) = I_d^x(\Gamma_j) = \mathbf{f}$, so that Γ_x fails strongly at Γ_j' in I_d^x

For the other direction, we use contraposition, so assume that $\Gamma_\exists$ does not fail strongly in I. We show that there exists a $d \in D$, s.t. Γ_x does not fail strongly in I_d^x. We distinguish the following cases:

– The first formula F with $I(F) \neq \mathbf{t}$ is among the Γ_i and $I(\Gamma_i) \neq \mathbf{f}$. Since for all $i,\ (1 \leq i \leq m),\ \ I(\Gamma_i) = I_d^x(\Gamma_i)$, the sequent Γ_x does not fail strongly in any I_d^x.
– The first formula F with $I(F) \neq \mathbf{t}$ is $\exists x\ P(x)$, and $I(\ \exists x\ P(x)\ ) \neq \mathbf{f}$. It follows from Definition 4 that there is a $d \in D$, s.t. $I_d^x(\ P(x)\ ) \notin \{\mathbf{f}, \mathbf{t}\}$. In the corresponding I_d^x, the sequent Γ_x does not fail strongly.
– The first formula F for which $I(F) \neq \mathbf{t}$ is among the Γ_j', and $I(\Gamma_j') \neq \mathbf{f}$. Since $I(\ \exists x\ P(x)\ ) = \mathbf{t}$, there exists a $d \in D$, s.t. $I_d^x(\ P(x)\ ) = \mathbf{t}$. In I_d^x, we have $I(\Gamma_1) = I_d^x(\Gamma_1) = \cdots = I(\Gamma_m) = I_d^x(\Gamma_m) = \mathbf{t},$ and $I(\Gamma_1') = I_d^x(\Gamma_1') = \cdots = I(\Gamma_{j-1}') = I_d^x(\Gamma_{j-1}') = \mathbf{t}$, so that Γ_x does not fail strongly in I_d^x.
– There is no formula F for which $I(F) \neq \mathbf{t}$ in $\Gamma_\exists$. This case is analogeous to the previous case.

2 Completeness

In the previous section we introduced Seq_{PCL} and proved its soundness. In the rest of the paper, we will give an outline of the completeness proof. If there would exist no $\forall$-quantifier, we would already have the completeness proof at this point. The calculus has sufficiently many equivalence preserving rules: For every interpretation I, the conclusion of the rule fails strongly in I iff all premises of the rule strongly fail in I. Using the equivalence preserving rules, it is possible to break down the goal sequent into a set of sequents that contain only (negations of) (Props of) atoms. These simple sequents are either axioms, or there exists a model in which they do not fail strongly. By the equivalence property, this implies that we either have a proof of the original sequent, or a counter interpretation.

In order to include $\forall$ in the completeness proof, we would like to proceed in a standard way: Allow each $\forall$-quantifier to have some fixed set of instances. If no proof can be constructed, then grant each $\forall$-quantifier one instance more. This process either results in a proof, or it leads to an increasing sequence of sets of atoms from which one can read of an interpretation in the limit.

Unfortunately, there is a problem with this approach, which is caused by the fact that in most cases the limit will be infinite. We want to show that the limit sequent does not fail strongly in the limit interpretation I (and that none of the sequents on the way fails strongly in I), but we have no concept of strong failure for infinite sequents. One possible solution would be to introduce infinite sequents. Infinite sequents can be defined by labelling a well-founded set with formulas. The sequent fails strongly if every element in the well-founded set that is labelled with a non-true formula, has an element before it, that is labelled with a false formula. Finite sequents would correspond to linearly ordered, finite sets. It turns out that there is a simpler approach, which avoids introducing special notions for infinite sequents:

Definition 8. *Let $\Gamma = \Gamma_1, \ldots, \Gamma_n \vdash$ be a sequent. We say that Γ is* in Prop normal form *(PNF) if for every Γ_i, either* **(1)** *Γ_i is of form $t_1 = t_2$, $t_1 \neq t_2$,* Prop(A) *or* $\neg$Prop(A), *or* **(2)** *there is a $j < i$, s.t. Γ_j has form* Prop(Γ_i).

Lemma 1. *Let $\Gamma \vdash$ be a sequent in PNF. Let I be an interpretation. Then $\Gamma \vdash$ does not fail strongly in I iff for every formula F in Γ, $I(F) = \mathbf{t}$.*

Theorem 5. *If* Seq_{PCL} *is complete for sequents in PNF, then it is complete for all sequents.*

Proof. Assume that Seq_{PCL} is complete for sequents in PNF. Let $\Gamma \vdash$ be an arbitrary sequent. Write $\Gamma \vdash$ in the form $\Gamma_1, \ldots, \Gamma_n \vdash$. Let $\#\Gamma \vdash$ be the number of violations of Definition 8 in $\Gamma \vdash$. (This is the number of Γ_i that are not of form $t_1 = t_2$, $t_1 \neq t_2$, Prop(A), $\neg$Prop(A), and for which there also exists no $j < i$ with $\Gamma_j = \text{Prop}(\Gamma_i)$.)

If $\#\Gamma \vdash = 0$, then $\Gamma \vdash$ is in PCL, so that we are done. Otherwise, assume that the first violation of Definition 8 occurs on position i. This implies that the sequent $S_1 = \Gamma_1, \ldots, \Gamma_{i-1}, \neg\text{Prop}(\Gamma_i) \vdash$ is in PNF. If S_1 has no proof, then by PNF-completeness, we know that there exists an interpretation I, in which S_1 does not fail strongly. By Lemma 1, $I(\Gamma_1) = \cdots = I(\Gamma_{i-1}) = I(\ \neg\text{Prop}(\Gamma_i)\ ) = \mathbf{t}$, so that $I(\ \text{Prop}(\Gamma_i)\ ) = \mathbf{f}$. This implies that the sequent $\Gamma_1, \ldots, \Gamma_{i-1}, \Gamma_i, \ldots, \Gamma_n \vdash$ fails in I, but not strongly. As a consequence, we have completeness for this case.

If S_1 does have a proof, then we consider the sequent
$S_2 = \Gamma_1, \ldots, \Gamma_{i-1},\ \text{Prop}(\Gamma_i), \Gamma_i, \ldots, \Gamma_n \vdash$. Clearly, $\#S_2 = \#(\Gamma \vdash) - 1$, so that we can assume completeness for S_2.

If S_2 has no proof, then there exists an interpretation I, in which S_2 does either not fail at all, or it fails but not strongly. If S_2 does not fail in I, then $\Gamma \vdash$ also does not fail, and we are done. Otherwise, consider the first formula F in S_2, for which $I(F) \neq \mathbf{t}$. If F were among the $\Gamma_1, \ldots, \Gamma_{i-1}$, this would imply that the sequent S_2 fails strongly, due to the fact that S_1 has a proof. From the provability of S_1 follows, that F cannot be Prop(Γ_i). If F would be Γ_i, this would imply that $I(\ \text{Prop}(\Gamma_i)\ ) = \mathbf{f}$, which contradicts the fact that S_2 is provable. So it must be the case that F is among $\Gamma_{i+1}, \ldots, \Gamma_n$. But this implies that $\Gamma \vdash$ also fails non strongly in I, so that we have completeness in this case as well.

Finally assume that S_2 has a proof. In that case, we can combine the proofs of S_1 and S_2 into a proof of $\Gamma \vdash$ as follows:

$$\frac{\Gamma_1, \ldots, \Gamma_{i-1},\ \neg\text{Prop}(\Gamma_i) \vdash \qquad\qquad \Gamma_1, \ldots, \Gamma_{i-1}, \text{Prop}(\Gamma_i), \Gamma_i, \ldots, \Gamma_n \vdash}{\Gamma_1, \ldots, \Gamma_{i-1}, \Gamma_i, \ldots, \Gamma_n \vdash}\ (cut).$$

The fact that we can restrict our attention to sequents in PNF, simplifies the completeness proof quite a lot. By Lemma 1, we know that we are looking either for a proof, or an interpretation that makes all atoms in the sequent true. Since this does not rely on order anymore, we can use standard techniques to construct the limit of the sequents in the failed proof attempt. We still have to show two things, but they turn out unproblematic: **(1)** It does not happen that, during proof search for a sequent in PNF, one needs to make use of a sequent that is not in PNF. **(2)** All formulas in the orginal sequent are true in the resulting interpretation. In order to do this, we show that a nonsucceeding proof attempt converges towards a saturated set, which is defined as follows:

Definition 9. *Let Σ be a set of formulas. We call Σ* saturated *if it has the following properties:*

- *If $A \in \Sigma$, and A is not of form $t_1 = t_2,\ t_1 \neq t_2,\ \mathrm{Prop}(B)$, or $\neg\mathrm{Prop}(B)$, then $\mathrm{Prop}(A) \in \Sigma$.*
- $\bot \notin \Sigma$.
- *There exist no terms t, u, no $n \geq 0$, no sequence of terms $t_1, u_1, \ldots, t_n, u_n$, s.t. $\{t_1 = u_1, \ldots, t_n = u_n, t \neq u\} \subseteq \Sigma$, and $t_1 = u_1, \ldots, t_n = u_n \vdash t = u$ in the standard theory of equality.*
- *There exist no atoms A, A', no $n \geq 0$, no sequence of terms $t_1, u_1, \ldots, t_n, u_n$, s.t. $\{A,\ t_1 = u_1, \ldots, t_n = u_n,\ \neg A'\} \subseteq \Sigma$, and $A,\ t_1 = u_1, \ldots, t_n = u_n \vdash A'$ in the standard theory of equality.*
- *If $\{\ \mathrm{Prop}(A \vee B),\ A \vee B\ \} \subseteq \Sigma$, then either $\{\mathrm{Prop}(A), A\} \subseteq \Sigma$, or $\{\mathrm{Prop}(B), B\} \subseteq \Sigma$.*
- *If $\{\ \mathrm{Prop}(\ \langle A \rangle B\ ),\ \langle A \rangle B\ \} \subseteq \Sigma$, then $\{\mathrm{Prop}(A), \mathrm{Prop}(B), A, B\} \subseteq \Sigma$.*
- *If $\{\ \mathrm{Prop}(\ \exists x\ P(x)\ ),\ \exists x\ P(x)\ \} \subseteq \Sigma$, then there exists a term t, s.t. $\{\ \mathrm{Prop}(\ P(t)\ ),\ P(t)\ \} \subseteq \Sigma$.*
- *If $\{\ \mathrm{Prop}(\ \forall x\ P(x)\ ),\ \forall x\ P(x)\ \} \subseteq \Sigma$, then for every term t that can be formed from the signature of Σ, we have $\{\ \mathrm{Prop}(P(t)),\ P(t)\ \} \subseteq \Sigma$.*
- *For every instance $A \Rightarrow B$ of a rule in Figure 1 or Figure 4, if $\{\mathrm{Prop}(A), A\} \subseteq \Sigma$, then $\{\mathrm{Prop}(B), B\} \subseteq \Sigma$.*
- *For every instance $\mathrm{Prop}(A) \Rightarrow B$ of a rule in Figure 2, if $\mathrm{Prop}(A) \in \Sigma$, but $A \notin \Sigma$, then $B \in \Sigma$.*

Note that, by taking $n = 0$ in the fourth case, the definition of saturated set implies that Σ does not contain a complementary pair of atoms $A, \neg A$. Since our aim is to prove completeness, we need a proof search strategy that converges towards a saturated set in the limit. In order to obtain a saturated set, it is necessary to preserve PNF during proof search. If, for example, one has a sequent of form $\Gamma_1, \mathrm{Prop}(A \wedge B), A \wedge B, \Gamma_2 \vdash$ and tries to prove it from $\Gamma_1, \mathrm{Prop}(A \wedge B), A, B, \Gamma_2 \vdash$, then the new sequent is not in PNF anymore. In this case we can continue proof search by replacing $\mathrm{Prop}(A \wedge B)$ by $\langle \mathrm{Prop}(A) \rangle \mathrm{Prop}(B)$, which in turn can be replaced by $\mathrm{Prop}(A), \mathrm{Prop}(B)$, which is in PNF again. Figures 5, 6, 7 and 8 show that, for the operators $\langle\ \rangle, \vee, \forall$ and $\exists$, it is always possible to continue proof search with sequents in PNF. All of the remaining cases can be reduced to the cases for $\langle\ \rangle$ and $\vee$, using the equivalences in Figures 1, 2 and 4.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_1,\ \mathrm{Prop}(A), \neg A, A, B, \Gamma_2 \vdash}\ \text{(provable)} \qquad \Gamma_1,\ \mathrm{Prop}(A),\ \mathrm{Prop}(B),\ A, B, \Gamma_2 \vdash}{\Gamma_1,\ \mathrm{Prop}(A),\ \neg A \vee \mathrm{Prop}(B), A, B, \Gamma_2 \vdash}\ (\vee\text{-intro})}{\Gamma_1,\ \langle\ \mathrm{Prop}(A)\ \rangle\ (\ A \rightarrow \mathrm{Prop}(B)\ ),\ \langle A \rangle B,\ \Gamma_2 \vdash}\ (\langle\ \rangle\text{-intro, Equiv Figure 4})}{\Gamma_1,\ \mathrm{Prop}(\ \langle A \rangle B\ ),\ \langle A \rangle B,\ \Gamma_2 \vdash}\ (\text{Equiv Figure 2})$$

Fig. 5. Preservation of PNF under $\langle\ \rangle$-intro

$$\dfrac{\dfrac{\Gamma_1,\ \mathrm{Prop}(A),\ \mathrm{Prop}(B),\ A,\ \Gamma_2 \vdash \qquad \Gamma_1,\ \mathrm{Prop}(A),\ \mathrm{Prop}(B), B,\ \Gamma_2 \vdash}{\Gamma_1,\ \mathrm{Prop}(A),\ \mathrm{Prop}(B), A \vee B, \Gamma_2 \vdash}\ (\vee\text{-intro})}{\Gamma_1,\ \mathrm{Prop}(\ A \vee B\ ),\ A \vee B,\ \Gamma_2 \vdash}\ (\text{Equiv Figure 2},\ \langle\ \rangle\text{-intro})$$

Fig. 6. Preservation of PNF under $\vee$-intro

In Figure 5, the leftmost sequent is provable, because it is in PNF, and it contains the complementary pair $A, \neg A$. Hence, it is sufficient to continue proof search with the rightmost sequent, which is also in PNF.
Figure 7 shows how PNF can be preserved when instantiating a $\forall$. In the middle, the proof splits at the cut application. The first formula of the left branch is provable, because it contains the complementary pair $\mathrm{Prop}(P(t)), \neg\mathrm{Prop}(P(t))$, Γ_1 is in PNF, and the remaining formulas $\forall x\ \mathrm{Prop}(\ P(x)\ )$, $\mathrm{Prop}(P(t))$, $\forall x\ P(x)$ can be easily proven Prop in their respective contexts. The right premise of the cut application is in PNF, and has the formulas $\mathrm{Prop}(P(t)), P(t)$ added. Figure 8 branches at the weakening step, and its first premise is provable.
It remains to extract an interpretation $I_\Sigma = (D_\Sigma, \mathbf{f}_\Sigma, \mathbf{t}_\Sigma, \sqsubseteq_\Sigma, [\ ]_\Sigma)$ from the saturated set Σ. This can be done as follows:

- Assume two designated objects $\mathbf{f}, \mathbf{t}$ that are not in the signature of Σ. They will represent the truth values. Let T_Σ be the set of terms that can be formed from the signature of Σ. Let $\equiv$ be the smallest congruence relation on $T_\Sigma \cup \{\mathbf{f}, \mathbf{t}\}$, s.t.
  - for all $t_1, t_2 \in T_\Sigma$, if $(t_1 = t_2) \in \Sigma$, then $t_1 \equiv t_2$,
  - for all $t \in T_\Sigma$, if both of $\mathrm{Prop}(t), t \in \Sigma$, then $t \equiv \mathbf{t}$,
  - for all $t \in T_\Sigma$, if $\mathrm{Prop}(t) \in \Sigma$, but $t \not\in \Sigma$, then $t \equiv \mathbf{f}$.

  The domain D_Σ of I_Σ is defined as $(T \cup \{\mathbf{f}, \mathbf{t}\}) / \equiv$.
- $\mathbf{f}_\Sigma$ is the element of D_Σ that contains $\mathbf{f}$.
- $\mathbf{t}_\Sigma$ is the element of D_Σ that contains $\mathbf{t}$.
- The choice of $\sqsubseteq_\Sigma$ is not important, so we simply select an arbitrary total order on $D_\Sigma \backslash \{\mathbf{f}_\Sigma, \mathbf{t}_\Sigma\}$.
- The function $[\ ]_\Sigma$ is defined in such a way that for every $t \in T_\Sigma$, the interpretation $I_\Sigma(t)$ is the equivalence class of $\equiv$ in which t falls.

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma_1, \forall x\ \text{Prop}(\ P(x)\ ), \text{Prop}(\ P(t)\ ), \forall x\ P(x),\ \neg\text{Prop}(\ P(t)\ ) \vdash}{\Gamma_1,\ \forall x\ \text{Prop}(\ P(x)\ ), \forall x\ P(x),\ \neg\text{Prop}(\ P(t)\ ) \vdash}\ (\forall\text{-intro})}{\Gamma_1,\ \text{Prop}(\ \forall x\ P(x)\ ), \forall x\ P(x),\ \neg\ \text{Prop}(\ P(t)\ ) \vdash}\ (\text{Equiv Figure 2}) \quad \Gamma_1,\ \text{Prop}(\ \forall x\ P(x)\ ), \forall x\ P(x),\ \text{Prop}(\ P(t)\ ), P(t), \Gamma_2 \vdash}{\Gamma_1,\ \text{Prop}(\ \forall x\ P(x)\ ), \forall x\ P(x),\ P(t),\ \Gamma_2 \vdash}\ (\text{cut})}{\Gamma_1,\ \text{Prop}(\ \forall x\ P(x)\ ), \forall x\ P(x), \Gamma_2 \vdash}\ \forall\text{-intro}$$

Fig. 7. Preservation of PNF under ∀-intro

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma_1,\ \neg\text{Prop}(\ \forall x\ \text{Prop}(\ P(x)\ )\ ) \vdash \qquad \Gamma_1,\ \text{Prop}(\ P(y)\ ),\ P(y),\ \Gamma_2 \vdash}{\Gamma_1,\ \forall x\ \text{Prop}(\ P(x)\ ),\ \text{Prop}(\ P(y)\ ),\ P(y),\ \Gamma_2 \vdash}\ (\text{weakening})}{\Gamma_1,\ \forall x\ \text{Prop}(\ P(x)\ ),\ P(y),\ \Gamma_2 \vdash}\ (\forall\text{-intro})}{\Gamma_1,\ \forall x\ \text{Prop}(\ P(x)\ ), \exists x\ P(x),\ \Gamma_2 \vdash}\ (\exists\text{-intro})}{\Gamma_1,\ \text{Prop}(\ \exists x\ P(x)\ ), \exists x\ P(x), \Gamma_2 \vdash}\ (\text{Equiv Figure 2})$$

Fig. 8. Preservation of PNF under ∃-intro

It remains to show that for every formula A, $A \in \Sigma \Leftrightarrow I_\Sigma(A) = \mathbf{t}$. This can be proven by structural induction on A using the properties in Definition 9.

Theorem 6. *Sequent calculus* Seq_{PCL} *is complete: If a sequent* $\Gamma \vdash$ *is not provable in* Seq_{PCL}*, then there exists an interpretation* I *in which* $\Gamma \vdash$ *does not fail strongly.*

3 Conclusions, Future Work

We have introduced a variant of first-order logic that supports partial functions and explicit type reasoning (PCL). We have introduced a semantics for PCL, which captures the intuitive meaning of partiality in a natural way. One of the motivations for introducing geometric resolution in [4] was the expectation that geometric resolution will be better at handling partial functions than standard automated theorem proving techniques. The current paper results from attempts to extend geometric resolution with partial functions. The next step is to extend geometric resolution (and its implementation Geo [3]), so that it can deal with PCL. On the theoretical side, we would like to know whether Seq_{PCL} admits cut elimination.

Part of this research was funded by the 'Programme for the Advancement of Computer Science' of the city of Wrocław.

References

1. Berezin, S., Barrett, C., Shikanian, I., Chechik, M., Gurfinkel, A., Dill, D.: A practical approach to partial functions in CVC Lite. In: Selected Papers from the Workshop on Disproving and the Second International Workshop on Pragmatics of Decision Procedures (PDPAR 04), July 2005. Electronic Notes in Theoretical Computer Science, vol. 125, pp. 13–23. Elsevier, Amsterdam (2005)
2. Darvas, Á., Mehta, F., Rudich, A.: Efficient well-definedness checking. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 100–115. Springer, Heidelberg (2008)
3. de Nivelle, H.: Theorem prover Geo 2007F. Can be obtained from the author's homepage (September 2007)
4. de Nivelle, H., Meng, J.: Geometric resolution: A proof procedure based on finite model search. In: Harrison, J., Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 303–317. Springer, Heidelberg (2006)
5. Farmer, W.M.: Mechanizing the traditional approach to partial functions. In: Farmer, W., Kerber, M., Kohlhase, M. (eds.) Proceedings of the Workshop on the Mechanization of Partial Functions (associated to CADE 13), pp. 27–32 (1996)
6. Farmer, W.M., Guttman, J.D.: A set theory with support for partial functions. Studia Logica 66, 59–78 (2000)
7. Hähnle, R.: Many-valued logic, partiality, and abstraction in formal specification languages. Logic Journal of the IGPL 13(4), 415–433 (2005)
8. Kerber, M., Kohlhase, M.: A mechanization of strong Kleene logic for partial functions. In: Bundy, A. (ed.) CADE 1994. LNCS, vol. 814, pp. 371–385. Springer, Heidelberg (1994)
9. Mehta, F.: A practical approach to partiality - a proof based approach. In: Liu, S., Maibaum, T., Araki, K. (eds.) ICFEM 2008. LNCS, vol. 5256, pp. 238–257. Springer, Heidelberg (2008)

Automated Reasoning for Relational Probabilistic Knowledge Representation[⋆]

Christoph Beierle[1], Marc Finthammer[1], Gabriele Kern-Isberner[2], and Matthias Thimm[2]

[1] Dept. of Computer Science, FernUniversität in Hagen, 58084 Hagen, Germany
[2] Dept. of Computer Science, TU Dortmund, 44221 Dortmund, Germany

Abstract. KReator is a toolbox for representing, learning, and automated reasoning with various approaches combining relational first-order logic with probabilities. We give a brief overview of the KReator system and its automated reasoning facilities.

1 Introduction

Approaches combining logic with probabilities for representing uncertain information are typically based upon propositional logic (see, e.g., [8]). Various extensions to a first-order setting like Bayesian logic programs [3, Ch. 10] or Markov logic networks [3, Ch. 12] have been proposed. In order to promote the use as well as the evaluation and comparison of such proposals, KReator provides a common and easy-to-use interface for representing, learning, and automated reasoning with different relational probabilistic approaches. In this paper, we give a brief description of the KReator system and its background and usage. In Sec. 2, we start with recalling the concept of Bayesian logic programs and Markov logic networks and sketch the relational maximum entropy framework RME. Section 3 gives an overview on the KReator system and presents a system walk-through with several examples, while Sec. 4 outlines its system architecture and implementation and points out further work.

2 Background

Bayesian logic programming combines logic programming and Bayesian networks [3, Ch. 10]. The basic structure for knowledge representation in Bayesian logic programs are *Bayesian clauses* like $(\mathit{alarm}(\mathsf{X}) \mid \mathit{lives\_in}(\mathsf{X},\mathsf{Y}), \mathit{tornado}(\mathsf{Y}))$ which model probabilistic dependencies between Bayesian atoms. A function cpd_c for a Bayesian clause c expresses the conditional probability distribution $P(\mathsf{head}(c) \mid \mathsf{body}(c))$ and thus partially describes an underlying probability distribution P. In order to aggregate probabilities that arise from applications of different Bayesian clauses with the same head, BLPs make use of *combining*

[⋆] The research reported here was partially supported by the Deutsche Forschungsgemeinschaft (grants BE 1700/7-1 and KE 1413/2-1).

rules. Semantics are given to Bayesian logic programs via transformation into propositional forms, i. e. into Bayesian networks [8] (see [3, Ch. 10] for details).

Markov logic [3, Ch. 12] establishes a framework which combines Markov networks [8] with first-order logic to handle a broad area of statistical relational learning tasks. The Markov logic syntax complies with first-order logic where each formula is quantified by an additional weight value, e.g., $(lives\_in(x, y) \wedge tornado(y) \Rightarrow alarm(x), 2.2)$. Semantics are given to sets of Markov logic formulas by a probability distribution over propositional possible worlds that is calculated as a log-linear model over weighted ground formulas. The fundamental idea in Markov logic is that first-order formulas are not handled as hard constraints but each formula is more or less softened depending on its weight. A *Markov logic network (MLN) L* is a set of weighted first-order logic formulas F_i together with a set of constants C. The semantics of L is given by a ground Markov network $M_{L,C}$ constructed from F_i and C [3, Ch. 12]. The standard semantics of Markov networks [8] is used for reasoning, e.g. to determine the consequences of L (see [3, Ch. 12] for details).

The basic idea of the *relational maximum entropy* framework (RME) [2,6,11]. is to make use of propositional maximum entropy techniques [7,4,9] after grounding the knowledge base appropriately. The *entropy H* is an information-theoretic measure on probability distributions and is defined as a weighted sum on the information encoded in every possible world $\omega \in \Omega$: $H(P) = -\sum_{\omega\in\Omega} P(\omega)\log P(\omega)$. By employing the *principle of maximum entropy* one can determine the single probability distribution that is the optimal model for a consistent knowledge base $\mathcal{R}$ in an information-theoretic sense: $P^{ME}_{\mathcal{R}} = \arg\max_{P\models\mathcal{R}} \mathcal{H}(P)$. However, this depends crucially on $\mathcal{R}$ being consistent, for otherwise no model of $\mathcal{R}$ exists, let alone models with maximum entropy. Since groundings may introduce non-trivial conflicts, this problem is all the more difficult in a first-order context with free variables where one has probabilistic first-order clauses like $(alarm(\mathsf{X}) \mid lives\_in(\mathsf{X}, \mathsf{Y}), tornado(\mathsf{Y}))[0.9]$, specifying that the conditional probability of $alarm(\mathsf{X})$ given $lives\_in(\mathsf{X}, \mathsf{Y})$ and $tornado(\mathsf{Y})$ ought to be 0.9. The RME inference process can be divided into three steps: (1) ground the knowledge base $\mathcal{R}$ with a grounding operator $\mathcal{G}$, (2) calculate the probability distribution $P^{ME}_{\mathcal{G}(\mathcal{R})}$ with maximum entropy for the grounded instance $\mathcal{G}(\mathcal{R})$, and (3) determine the probabilistic implications of $P^{ME}_{\mathcal{G}(\mathcal{R})}$ [2,6,11].

3 Examples and System Overview

The KReator system provides automated reasoning facilities for all three probabilistic relational frameworks sketched in Sec. 2. As an illustration, we consider the well-known burglary example given in [8] where we have some (uncertain) beliefs about the relationships between burglaries, types of neighborhoods, natural disasters, and alarms. This example could be represented by a BLP containig the three Bayesian clauses

$c_1: (alarm(\mathsf{X}) \mid burglary(\mathsf{X}))$ $\qquad$ $c_3: (burglary(\mathsf{X}) \mid nhood(\mathsf{X}))$

$c_2: (alarm(\mathsf{X}) \mid lives\_in(\mathsf{X}, \mathsf{Y}), tornado(\mathsf{Y}))$

together with corresponding conditional probability distributions cpd_{c_i}. For instance, $\mathsf{cpd}_{c_2}(\mathsf{true}, \mathsf{true}, \mathsf{true}) = 0.9$ would express our subjective belief that *alarm* is true with probability 0.9 if $\mathit{lives\_in}(\mathsf{X}, \mathsf{Y})$ and $\mathit{tornado}(\mathsf{Y})$ are true. In this BLP modelling, *nhood* is a multi-valued unary predicate, while in the modellings using MLNs and RME, *nhood* will be a binary-valued two-place predicate.

Using the Alchemy syntax [5] for MLN files, a MLN knowledge base for the burglary example is given by the following five weighted clauses:

$$\begin{array}{llll}
2.2 & \mathit{burglary}(x) \Rightarrow \mathit{alarm}(x) & -0.8 & \mathit{nhood}(x, \mathit{Good}) \Rightarrow \mathit{burglary}(x) \\
2.2 & \mathit{lives\_in}(x,y) \wedge \mathit{tornado}(y) \Rightarrow \mathit{alarm}(x) & -0.4 & \mathit{nhood}(x, \mathit{Average}) \Rightarrow \mathit{burglary}(x) \\
 & & 0.4 & \mathit{nhood}(x, \mathit{Bad}) \Rightarrow \mathit{burglary}(x)
\end{array}$$

Note that, in contrast to BLPs and RMEs, MLNs do not support conditional probabilities, so the rule-like knowledge has to be modeled as material implications. Modeling our running example in RME can be done by

$$\begin{array}{ll}
c_1\colon (\mathit{alarm}(\mathsf{X}) \mid \mathit{burglary}(\mathsf{X}))[0.9] & c_4\colon (\mathit{burglary}(\mathsf{X}) \mid \mathit{nhood}(\mathsf{X}, \mathit{average}))[0.4] \\
c_2\colon (\mathit{alarm}(\mathsf{X}) \mid \mathit{lives\_in}(\mathsf{X}, \mathsf{Y}), & c_5\colon (\mathit{burglary}(\mathsf{X}) \mid \mathit{nhood}(\mathsf{X}, \mathit{good}))[0.3] \\
\qquad \mathit{tornado}(\mathsf{Y}))[0.9] & c_6\colon (\mathit{nhood}(\mathsf{X}, \mathsf{Z}) \mid \mathit{nhood}(\mathsf{X}, \mathsf{Y}))[0.0]\ [\mathsf{Y} \neq \mathsf{Z}] \\
c_3\colon (\mathit{burglary}(\mathsf{X}) \mid \mathit{nhood}(\mathsf{X}, \mathit{bad}))[0.6] & c_7\colon (\mathit{lives\_in}(\mathsf{X}, \mathsf{Z}) \mid \mathit{lives\_in}(\mathsf{X}, \mathsf{Y}))[0.0]\ [\mathsf{Y} \neq \mathsf{Z}]
\end{array}$$

where in this knowledge base, the conditionals c_6 and c_7 ensure mutual exclusion of the states for literals of "*nhood*" and "*lives_in*".

A query in so-called *unified syntax* can be answered by KReator with respect to a BLP, MLN, or RME knowledge base. This query syntax abstracts from the respective syntax which is necessary to address a "native" query to a BLP, MLN, or RME knowledge base. The idea behind this functionality is that while some knowledge can be modeled in different knowledge representation approaches, the user is able to compare the reasoning facilities of these approaches in a direct way by formulating appropriate queries in unified syntax, passing them to the

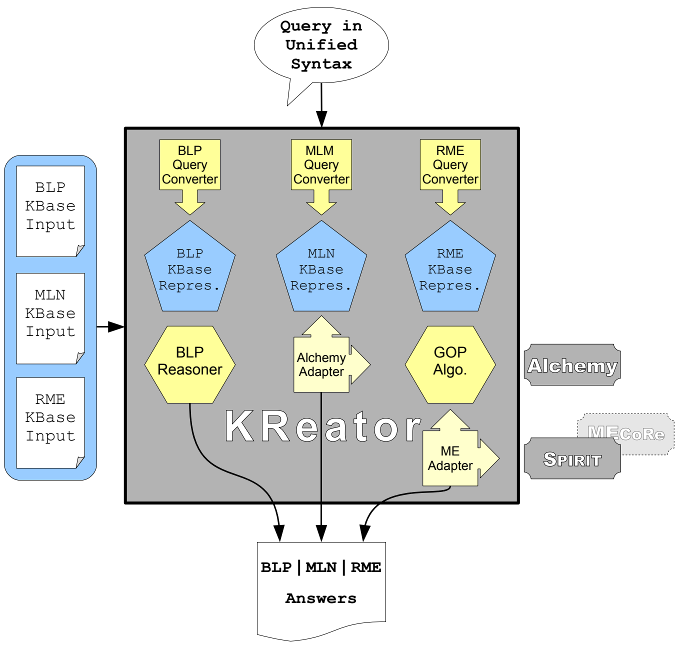


Fig. 1. Processing query in unified syntax

different knowledge bases, and analyzing the different answers. A KReator query in unified syntax consists of two parts: In the *head* of the query there are one or more ground atoms whose probabilities shall be determined. The *body* of the query is composed of several evidence atoms. For each supported knowledge representation formalism, KReator must convert a query in unified syntax in the exact syntax required by the respective inference engine. Among other things, this includes e. g. the conversion from lower case constants to upper case ones (and variables, vice versa), as required by the Alchemy tool for processing MLNs. KReator also converts the respective output results to present them in a standardized format to the user (cf. Fig. 1).

In addition to a knowledge base, that typically contains only general generic knowledge, also evidential knowledge like

$$lives\_in(james, yorkshire), lives\_in(carl, austin), burglary(james),$$
$$tornado(austin), nhood(james) = \mathsf{average}, nhood(carl) = \mathsf{good}$$

can be taken into account when reasoning with KReator. The following table shows three queries and their respective probabilities inferred by KReator from each of the example knowledge bases and the evidence given above:

	BLP	MLN	RME
alarm(*james*)	0.900	0.896	0.918
alarm(*carl*)	0.550	0.900	0.880
burglary(*carl*)	0.300	0.254	0.362

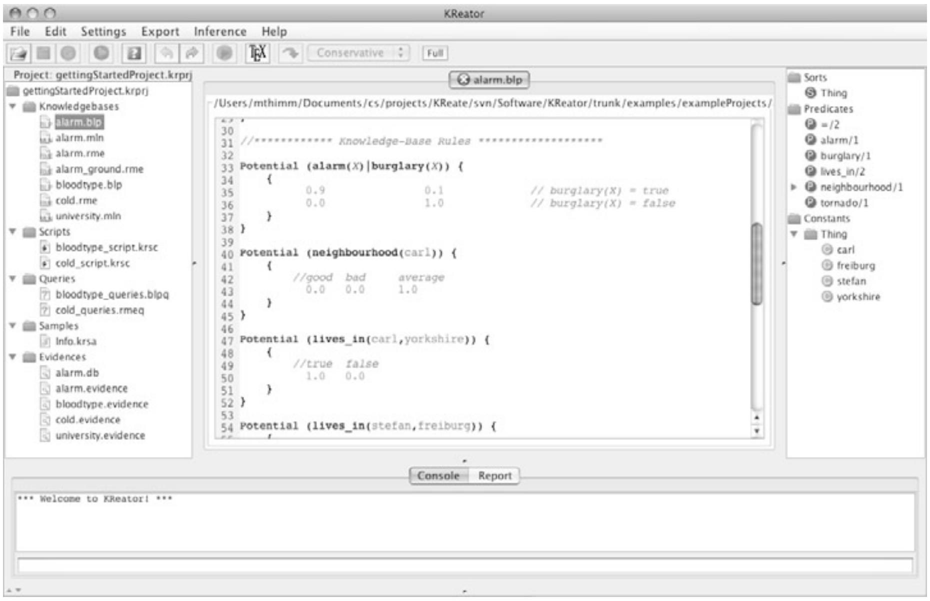
The inferred probabilities are are not identical since the same generic knowledge is modelled slightly differently in the three formalisms. For instance in BLPs, specific information resides in the combinig rules (in this example, noisy-or was used) that aggregate probabilities of different clauses with the same head.

Doing further computations in the different formalisms is conveniently supported by KReator. For example, dropping *tornado*(*austin*) from the evidence yields, as expected, the values for the query *alarm*(*james*) as given in the table above; whereas the values for *alarm*(*carl*) drop dramatically. Replacing *burglary*(*james*) by *alarm*(*james*) in the evidence and asking for *burglary*(*james*) yields 0.400 (BLP), 0.411 (MLN), and 0.507 (RME).

All specification and reasoning steps involved in these examples are conveniently supported by the KReator system. KReator comes with a graphical user interface and an integrated console-based interface. The main view of KReator (see Fig. 2) is divided into the menu, a toolbar and four main panels: the project, editor, outline, and console panel.

KReator structures its data into projects which may contain knowledge bases, scripts written in KReatorScript (see below), query collections for knowledge bases, and sample/evidence files. Although all types of files can be opened independently in KReator, projects can help the knowledge engineer to organize his work. The *project panel* (upper left in Fig. 2) gives a complete overview on the project the user is currently working on.

All files supported by KReator can be viewed and edited in the *editor panel* (upper middle in Fig. 2). Multiple files can be opened at the same time and the



```
//************ Knowledge-Base Rules ******************

Potential (alarm(X)|burglary(X)) {
    {
        0.9              0.1              // burglary(X) = true
        0.0              1.0              // burglary(X) = false
    }
}

Potential (neighbourhood(carl)) {
    {
      //good  bad     average
        0.0   0.0     1.0
    }
}

Potential (lives_in(carl,yorkshire)) {
    {
      //true  false
        1.0   0.0
    }
}

Potential (lives_in(stefan,freiburg)) {
```

```
*** Welcome to KReator! ***
```

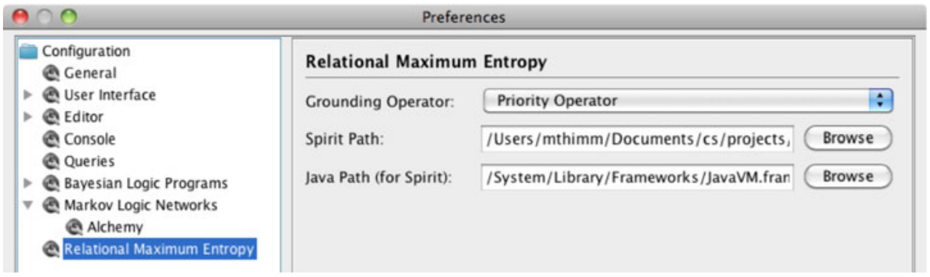
Fig. 2. KReator – Main window

editor supports editing knowledge bases and the like with syntax-highlighting, syntax check, and other features normally known from development environments for programming languages.

The *outline panel* (upper right in Fig. 2) gives an overview on the currently viewed file in the editor panel. If the file is a knowledge base the outline shows information on the logical components of the knowledge base, such as used predicates (and, in case of BLPs, their states), constants, and sorts (if the knowledge base uses a typed language).

The *console panel* (bottom in Fig. 2) contains two tabs, one with the actual console interface and one with the *report*. The console can be used to access all KReator functionality just using textual commands, e. g. querying knowledge bases, open and saving file, and so on. The console is a live interpreter for KReatorScript which can also be used for writing scripts that allows the knowledge engineer to save recurring tasks. By doing so results can be verified and sophisticated queries can be addressed to different knowledge bases with little adaptations. As a further ease, every action executed in KReator, e. g. opening a file in the graphical user interface or querying a knowledge base from the console, is recorded as a KReatorScript command in the report. The whole report or parts of it can easily be saved as script file and executed again when experiments have to be repeated and results have to be reproduced.

KReator is highly configurable and extensible. Figure 3 shows the preferences dialog which enables the configuration of nearly every property of the graphical user interface as well as special features of the different knowledge



Relational Maximum Entropy

Grounding Operator: Priority Operator

Spirit Path: /Users/mthimm/Documents/cs/projects,

Java Path (for Spirit): /System/Library/Frameworks/JavaVM.fran

Fig. 3. Preferences in KReator

representation formalism. Furthermore, every property can also be modified using KReatorScript so that different configurations can be tested easily in script files. Due to the open architecture, KReator can be extended effortlessly by further formalisms and most of the features like querying are automatically enabled.

4 System Architecture and Implementation

The implementation of KReator is done in Java and mirrors its objective to support different approaches to relational probabilistic knowledge representation and reasoning. It strictly separates the internal logic and the user interface, employing an abstract command structure allowing easy modifications on both sides. In order to support the implementation of other approaches, KReator features a large library on first-order logic and basic probabilistic methods. Among others this library contains classes for formulæ, rules, conditionals and various methods to operate on these. There is also a rudimentary implementation of Prolog available that can be used for specifying background knowledge as e. g. in BLPs. This integrated library is designed to support a fast implementation of specific approaches to statistical relational learning. The task of integrating a new approach into the KReator system is supported by a small set of interfaces that have to be implemented in order to be able to access the new approach from the user interface. There are interfaces for knowledge bases (which demands e. g. support for querying), file writers and parsers (for reading and writing the specific syntax of an approach), and learner. One thing to note is that both file writers and parsers have to work on strings only, all the cumbersome overhead of file operations and I/O is handled by KReator. With the help of a plugin-like architecture the developer of a new approach only has to be concerned with connecting her approach to KReator using these interfaces. Then all the benefits of an integrated development environment as provided by KReator are immediately accessible. Currently, KReator supports knowledge representation using BLPs, MLNs, and the relational maximum entropy approach RME; other formalisms will be integrated in the near future.

Performing inference on MLNs is done using the Alchemy software package [5], a console-based tool for processing Markov logic networks. For BLPs, a reasoning component was implemented within KReator. To process ground RME knowledge bases, KReator uses a so-called ME-adapter to communicate with a MaxEnt-reasoner. Currently, such adapters are supplied for the SPIRIT reasoner [10] and for MEcore [1] which are tools for processing (propositional) conditional probabilistic knowledge bases using maximum entropy methods.

Ongoing work includes integration of different learning algorithms. Due to the availability of several formalisms in KReator these algorithms can be implemented in a very general manner and employed by the formalisms in an easy way. Future work consists of extending the support for other relational probabilistic formalisms, such as Probabilistic Relational Models [3, Ch. 5].

References

1. Finthammer, M., Beierle, C., Berger, B., Kern-Isberner, G.: Probabilistic reasoning at optimum entropy with the MEcore system. In: Lane, H.C., Guesgen, H.W. (eds.) Proceedings 22nd International FLAIRS Conference, FLAIRS'09. AAAI Press, Menlo Park (2009)
2. Finthammer, M., Loh, S., Thimm, M.: Towards a toolbox for relational probabilistic knowledge representation, reasoning, and learning. In: Relational Approaches to Knowledge Representation and Learning. Workshop at KI-2009, Paderborn, Germany, Informatik-Bericht, vol. 354, pp. 34–48. FernUniv. in Hagen (2009)
3. Getoor, L., Taskar, B. (eds.): Introduction to Statistical Relational Learning. MIT Press, Cambridge (2007)
4. Kern-Isberner, G.: Characterizing the principle of minimum cross-entropy within a conditional-logical framework. Artificial Intelligence 98, 169–208 (1998)
5. Kok, S., Singla, P., Richardson, M., Domingos, P., Sumner, M., Poon, H., Lowd, D., Wang, J.: The Alchemy System for Statistical Relational AI: User Manual. Department of Computer Science and Engineering. University of Washington (2008)
6. Loh, S., Thimm, M., Kern-Isberner, G.: On the problem of grounding a relational probabilistic conditional knowledge base. In: Proceedings of the 14th International Workshop on Non-Monotonic Reasoning (NMR'10), Toronto, Canada (May 2010)
7. Paris, J.B.: The uncertain reasoner's companion – A mathematical perspective. Cambridge University Press, Cambridge (1994)
8. Pearl, J.: Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference. Morgan Kaufmann, San Francisco (1998)
9. Rödder, W., Meyer, C.-H.: Coherent knowledge processing at maximum entropy by SPIRIT. In: Horvitz, E., Jensen, F. (eds.) Proceedings 12th Conference on Uncertainty in Artificial Intelligence, pp. 470–476. Morgan Kaufmann, San Francisco (1996)
10. Rödder, W., Reucher, E., Kulmann, F.: Features of the expert-system-shell SPIRIT. Logic Journal of the IGPL 14(3), 483–500 (2006)
11. Thimm, M., Finthammer, M., Loh, S., Kern-Isberner, G., Beierle, C.: A system for relational probabilistic reasoning on maximum entropy. In: Proceedings 23rd International FLAIRS Conference, FLAIRS'10. AAAI Press, Menlo Park (to appear, 2010)

Optimal and Cut-Free Tableaux for Propositional Dynamic Logic with Converse

Rajeev Goré[1] and Florian Widmann[1,2]

[1]Logic and Computation Group
[2,⋆] NICTA The Australian National University
Canberra, ACT 0200, Australia,
{Rajeev.Gore,Florian.Widmann}@anu.edu.au

Abstract. We give an optimal (EXPTIME), sound and complete tableau-based algorithm for deciding satisfiability for propositional dynamic logic with converse (CPDL) which does not require the use of analytic cut. Our main contribution is a sound method to combine our previous optimal method for tracking least fix-points in PDL with our previous optimal method for handling converse in the description logic *ALCI*. The extension is non-trivial as the two methods cannot be combined naively. We give sufficient details to enable an implementation by others. Our OCaml implementation seems to be the first theorem prover for CPDL.

1 Introduction

Propositional dynamic logic (PDL) is an important logic for reasoning about programs. Its formulae consist of traditional Boolean formulae plus "action modalities" built from a finite set of atomic programs using sequential composition ($;$), non-deterministic choice ($\cup$), repetition ($*$), and test ($?$). The logic CPDL is obtained by adding converse ($^-$), which allows us to reason about previous actions. The satisfiability problem for CPDL is EXPTIME-complete [1].

De Giacomo and Massacci [2] give an NEXPTIME tableau algorithm for deciding CPDL-satisfiability, and discuss ways to obtain optimality, but do not give an actual EXPTIME algorithm. The tableau method of Nguyen and Szałas [3] is optimal. Neither method has been implemented, and since both require an explicit analytic cut rule, it is not at all obvious that they can be implemented efficiently. Optimal game-theoretic methods for fix-point logics [4] can be adapted to handle CPDL [5] but involve significant non-determinism. Optimal automata-based methods [6] for fix-point logics are still in their infancy because good optimisations are not known. We know of no resolution methods for CPDL.

We give an optimal tableau method for deciding CPDL-satisfiability which does not rely on a cut rule. Our main contribution is a sound method to combine our method for tracking and detecting unfulfilled eventualities as early as possible

⋆ NICTA is funded by the Australian Government's Department of Communications, Information Technology and the Arts and the Australian Research Council through Backing Australia's Ability and the ICT Centre of Excellence program.

Table 1. Smullyan's α- and β-notation to classify formulae

α	$\varphi \wedge \psi$	$[\gamma \cup \delta]\varphi$	$[\gamma *]\varphi$	$\langle \psi ? \rangle \varphi$	$\langle \gamma ; \delta \rangle \varphi$	$[\gamma ; \delta]\varphi$
α_1	φ	$[\gamma]\varphi$	φ	φ	$\langle \gamma \rangle \langle \delta \rangle \varphi$	$[\gamma][\delta]\varphi$
α_2	ψ	$[\delta]\varphi$	$[\gamma][\gamma *]\varphi$	ψ		

β	$\varphi \vee \psi$	$\langle \gamma \cup \delta \rangle \varphi$	$\langle \gamma * \rangle \varphi$	$[\psi ?]\varphi$
β_1	φ	$\langle \gamma \rangle \varphi$	φ	φ
β_2	ψ	$\langle \delta \rangle \varphi$	$\langle \gamma \rangle \langle \gamma * \rangle \varphi$	$\sim \psi$

in PDL [7] with our method for handling converse for *ALCI* [8]. The extension is non-trivial as the two methods cannot be combined naively.

We present a mixture of pseudo code and tableau rules rather than a set of traditional tableau rules to enable easy implementation by others. Our unoptimised OCaml implementation appears to be the first automated theorem prover for CPDL (http://rsise.anu.edu.au/~rpg/CPDLTabProver/). A longer version with full proofs is available at http://arxiv.org/abs/1002.0172.

2 Syntactic Preliminaries

Definition 1. *Let* AFml *and* APrg *be two disjoint and countably infinite sets of propositional variables and* atomic programs*, respectively. The set* LPrg *of* literal programs *is defined as* LPrg := APrg $\cup \{a^- \mid a \in$ APrg$\}$*. The set* Fml *of all formulae and the set* Prg *of all* programs *are defined mutually inductively as follows where* $p \in$ AFml *and* $l \in$ LPrg*:*

$$\begin{array}{ll} \text{Fml} & \varphi ::= p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \langle\gamma\rangle\varphi \mid [\gamma]\varphi \\ \text{Prg} & \gamma ::= l \mid \gamma;\gamma \mid \gamma \cup \gamma \mid \quad \gamma * \quad \mid \varphi? \ . \end{array}$$

A $\langle$lp$\rangle$*-formula is a formula* $\langle\gamma\rangle\varphi$ *where* $\gamma \in$ LPrg *is a* $\underline{\text{l}}$*iteral* $\underline{\text{p}}$*rogram.*

Implication ($\rightarrow$) and equivalence ($\leftrightarrow$) are not part of the core language but can be defined as usual. In the rest of the paper, let $p \in$ AFml and $l \in$ LPrg.

We omit the semantics as it is a straightforward extension of PDL [7] and write $M, w \Vdash \varphi$ if $\varphi \in$ Fml holds in the world $w \in W$ of the model M.

Definition 2. *For a literal program* $l \in$ LPrg*, we define* $l^{\smile}$ *as* a *if* l *is of the form* a^-*, and as* l^- *otherwise. A formula* $\varphi \in$ Fml *is in* negation normal form *if the symbol* $\neg$ *appears only directly before propositional variables. For every* $\varphi \in$ Fml*, we can obtain a formula* nnf(φ) *in negation normal form by pushing negations inward such that* $\varphi \leftrightarrow$ nnf φ *is valid. We define* $\sim\varphi :=$ nnf$(\neg\varphi)$*.*

We categorise formulae as α- or β-formulae as shown in Table 1 so that the formulae of the form $\alpha \leftrightarrow \alpha_1 \wedge \alpha_2$ and $\beta \leftrightarrow \beta_1 \vee \beta_2$ are valid. An *eventuality* is a formula of the form $\langle\gamma_1\rangle \dots \langle\gamma_k\rangle\langle\gamma*\rangle\varphi$, and Ev is the set of all eventualities. Using Table 1, the binary relation "$\rightsquigarrow$" relates a $\langle\rangle$-formulae α (respectively β), to its reduction α_1 (respectively β_1 and β_2). See [7, Def. 7] for their formal definitions.

3 An Overview of our Algorithm

Our algorithm builds an and-or graph G by repeatedly applying four rules (see Table 2) to try to build a model for a given ϕ in negation normal form. Each node x carries a formula set Γ_x, a status sts_x, and other fields to be described shortly. Rule 1 applies the usual expansion rules to a node to create its children. These expansion rules capture the semantics of CPDL. We use Smullyan's α/β-rule notation for classifying rules and nodes. As usual, a node x is a ("saturated") *state* if no α/β-rule can be applied to it. If x is a state then for each $\langle l\rangle\xi$ in Γ_x, we create a node y with $\Gamma_y = \{\xi\} \cup \Delta$, where $\Delta = \{\psi \mid [l]\psi \in \Gamma_x\}$, and add an edge from x to y labelled with $\langle l\rangle\xi$ to record that y is an l-successor of x.

If Γ_x contains an obvious contradiction during expansion, its status becomes "closed", which is irrevocable. Else, at some later stage, Rule 2 determines its status as either "closed" or "open". "Open" nodes contain additional information which depends on the status of other nodes. Hence, if a node changes its status, it might affect the status of another ("open") node. If the stored status of a node does not match its current status, the node is no longer *up-to-date*. Rule 3, which may be applied multiple times to the same node, ensures that "open" nodes are kept up-to-date by recomputing their status if necessary. Finally, Rule 4 detects eventualities which are impossible to fulfil and closes nodes which contain them. We first describe the various important components of our algorithm separately.

Global State Caching. For optimality, the graph G never contains two state nodes which carry the same set of formulae [8]. However, there may be multiple non-states which carry the same set of formulae. That is, a non-state node x carrying Γ which appears while saturating a child y of a state z is unique to y. If a node carrying Γ is required in some other saturation phase, a new node carrying Γ is created. Hence the nodes of two saturation phases are distinct.

Converse. Suppose state y is a descendant of an l-successor of a state x, with no intervening states. Call x the parent state of y since all intervening nodes are not states. We require that $\{\psi \mid [l^-]\psi \in \Gamma_y\} \subseteq \Gamma_x$, since y is then compatible with being a l-successor of x in the putative model under construction. If some $[l^-]\psi \in \Gamma_y$ has $\psi \notin \Gamma_x$ then x is "too small", and must be "restarted" as an alternative node x^+ containing all such ψ. If any such ψ is a complex formula to which an α/β-rule is applicable then x^+ is not a state and may have to be "saturated" further. The job of creating these alternatives is done by *special nodes* [8]. Each special node monitors a state and creates the alternatives when needed.

Detecting Fulfilled and Unfulfilled Eventualities. Suppose the current node x contains an eventuality e_x. There are three possibilities. The first is that e_x can be fulfilled in the part of the graph which is "older" than x. Else, it may be possible to reach a node z in the parts of the graph "newer" than x such that z contains a reduction e_z of e_x. Since this "newer" part of the graph is not fully explored yet, future expansions may enable us to fulfil e_x via z, so the pair (z, e_z) is a "potential rescuer" of e_x. The only remaining case is that e_x cannot be fulfilled in the "older" part of the graph, and has no potential rescuers. Thus future expansions of the graph cannot possibly help to fulfil e_x since it cannot reach these "newer" parts of the future graph. In this case x can be "closed". The technical machinery to maintain this information for PDL is from [7]. However, the presence of "converse" and the resulting need for alternative nodes requires a more elaborate scheme for CPDL.

4 The Algorithm

Our algorithm builds a directed graph G consisting of nodes and directed edges. We first explain the structure of G in more detail.

Definition 3. *Let X and Y be sets. We define $X^\perp := X \uplus \{\perp\}$ where $\perp$ indicates the undefined value and $\uplus$ is the disjoint union. If $f : X \to Y$ is a function and $x \in X$ and $y \in Y$ then the function $f[x \mapsto y] : X \to Y$ is defined as $f[x \mapsto y](x') := y$ if $x' = x$ and $f[x \mapsto y](x') := f(x')$ if $x' \neq x$.*

Definition 4. *Let $G = (V, E)$ be a graph where V is a set of nodes and E is a set of directed edges. Each node $x \in V$ has six attributes: $\Gamma_x \subseteq \mathrm{Fml}$, $\mathrm{ann}_x : \mathrm{Ev} \to \mathrm{Fml}^\perp$, $\mathrm{pst}_x \in V^\perp$, $\mathrm{ppr}_x \in \mathrm{LPrg}^\perp$, $\mathrm{idx}_x \in \mathit{Nat}^\perp$, and $\mathrm{sts}_x \in \mathfrak{S}$ where $\mathfrak{S} := \{\mathtt{unexp}, \mathtt{undef}\} \cup \{\mathtt{closed}(\mathrm{alt}) \mid \mathrm{alt} \subseteq \mathscr{P}(\mathrm{Fml})\} \cup \{\mathtt{open}(\mathrm{prs}, \mathrm{alt}) \mid \mathrm{prs} : \mathrm{Ev} \to (\mathscr{P}(V \times \mathrm{Ev}))^\perp \;\&\; \mathrm{alt} \subseteq \mathscr{P}(\mathrm{Fml})\}$. Each directed edge $e \in E$ is labelled with a label $l_e \in (\mathrm{Fml} \cup \mathscr{P}(\mathrm{Fml}) \cup \{\mathrm{cs}\})^\perp$ where cs is just a constant.*

All attributes of a node $x \in V$ are initially set at the creation of x, possibly with the value $\perp$ (if allowed). Only the attributes idx_x and sts_x are changed at a later time. We use the function create-new-node$(\Gamma, \mathrm{ann}, \mathrm{pst}, \mathrm{ppr}, \mathrm{idx}, \mathrm{sts})$ to create a new node and initialise its attributes in the obvious way.

The finite set Γ_x contains the formulae which are assigned to x. The attribute ann_x is defined for the eventualities in Γ_x at most. If $\mathrm{ann}_x(\varphi) = \varphi'$ then $\varphi' \in \Gamma_x$ and $\varphi \rightsquigarrow \varphi'$. The intuitive meaning is that φ has already been "reduced" to φ' in x. For a state (as defined below) we always have that ann_x is undefined everywhere since we do not need the attribute for states.

The node x is called a *state* iff both attributes pst_x and ppr_x are undefined. For all other nodes, the attribute pst_x identifies the, as we will ensure, unique ancestor $p \in V$ of x such that p is a state and there is no other state between p and x in G. We call p the *$\underline{p}$arent $\underline{st}$ate* of x. The creation of the child of p which lies on the path from p to x (it could be x) was caused by a $\langle \mathrm{lp} \rangle$-formula $\langle l \rangle \varphi$ in Γ_p. The literal program l which we call the *$\underline{p}$arent $\underline{pr}$ogram* of x is stored in ppr_x. Hence, for nodes which are not states, both pst_x and ppr_x are defined.

The attribute sts_x describes the *status* of x. Unlike the attributes described so far, its value may be modified several times. The value `unexp`, which is the initial value of each node, indicates that the node has not yet been expanded. When a node is expanded, its status becomes either $\mathtt{closed}(\cdot)$ if it contains an immediate contradiction, or `undef` to indicate that the node has been expanded but that its "real" status is to be determined. Eventually, the status of each node is set to either $\mathtt{closed}(\cdot)$ or $\mathtt{open}(\cdot, \cdot)$. If the status is $\mathtt{open}(\cdot, \cdot)$, it might be modified several times later on, either to $\mathtt{closed}(\cdot)$ or to $\mathtt{open}(\cdot, \cdot)$ (with different arguments), but once it becomes $\mathtt{closed}(\cdot)$, it will never change again.

We call a node *undefined* if its status is `unexp` or `undef` and *defined* otherwise. Hence a node is undefined initially, becomes defined eventually, and then never becomes undefined again. Furthermore, we call x *closed* iff its status is $\texttt{closed}(\mathrm{alt})$ for some $\mathrm{alt} \subseteq \mathscr{P}(\mathrm{Fml})$. In this case, we define $\mathrm{alt}_x := \mathrm{alt}$. We call x *open* iff its status is $\texttt{open}(\mathrm{prs}, \mathrm{alt})$ for some $\mathrm{prs} : \mathrm{Ev} \to (\mathscr{P}(V \times \mathrm{Ev}))^{\perp}$ and some $\mathrm{alt} \subseteq \mathscr{P}(\mathrm{Fml})$. In this case, we define $\mathrm{prs}_x := \mathrm{prs}$ and $\mathrm{alt}_x := \mathrm{alt}$. To avoid some clumsy case distinctions, we define $\mathrm{alt}_x := \emptyset$ if x is undefined.

The value $\texttt{closed}(\mathrm{alt})$ indicates that the node is "useless" for building an interpretation because it is either unsatisfiable or "too small". In the latter case, the set alt of *alternative sets* contains information about missing formulae. Finally, the value $\texttt{open}(\mathrm{prs}, \mathrm{alt})$ indicates that there is still hope that x is "useful" and the function prs_x contains information about each eventuality $e_x \in \Gamma_x$ as explained in the overview. Although x itself may be useful, we need its alternative sets in case it becomes closed later on. Hence it also has a set of alternative sets.

The attribute idx_x serves as a time stamp. It is set to $\perp$ at creation time of x and becomes defined when x becomes defined. When this happens, the value of idx_x is set such that $\mathrm{idx}_x > \mathrm{idx}_y$ for all nodes y which became defined earlier than x. We define $y \sqsubset x$ iff $\mathrm{idx}_y \neq \perp$ and either $\mathrm{idx}_x = \perp$ or $\mathrm{idx}_y < \mathrm{idx}_x$. Note that $y \sqsubset x$ depends on the current state of the graph. However, once $y \sqsubset x$ holds, it will do so for the rest of the time.

To track eventualities, we label an edge between a state and one of its children by the $\langle \mathrm{lp} \rangle$-formula $\langle l \rangle \varphi$ which creates this child. Additionally, we label edges from special nodes (see overview) to their <u>c</u>orresponding <u>s</u>tates with the marker cs. We also label edges from special nodes to its alternative nodes with the corresponding alternative set.

Definition 5. *Let* $\mathrm{ann}^{\perp} : \mathrm{Ev} \to \mathrm{Fml}^{\perp}$ *and* $\mathrm{prs}^{\perp} : \mathrm{Ev} \to (\mathscr{P}(V \times \mathrm{Ev}))^{\perp}$ *be the functions which are undefined everywhere. For a node* $x \in V$ *and a label* $l \in \mathrm{Fml} \cup \mathscr{P}(\mathrm{Fml}) \cup \{\mathrm{cs}\}$*, let* $\mathrm{getChild}(x, l)$ *be the node* $y \in V$ *such that there exists an edge* $e \in E$ *from* x *to* y *with* $l_e = l$*. If* y *does not exists or is not unique, let the result be* $\perp$*. For a function* $\mathrm{prs} : \mathrm{Ev} \to (\mathscr{P}(V \times \mathrm{Ev}))^{\perp}$*, a node* $x \in V$*, and an eventuality* $\varphi \in \mathrm{Ev}$*, we define the set* $\mathrm{reach}(\mathrm{prs}, x, \varphi)$ *of eventualities as follows:*

$$\mathrm{reach}(\mathrm{prs}, x, \varphi) := \Big\{ \psi \in \mathrm{Ev} \mid \exists k \in \mathbb{N}_0.\, \exists \varphi_0, \ldots, \varphi_k \in \mathrm{Ev}.\, \Big(\psi = \varphi_k \;\&\; (x, \varphi_0) \in \mathrm{prs}(\varphi) \;\&\; \forall i \in \{0, \ldots, k-1\}.\, (x, \varphi_{i+1}) \in \mathrm{prs}(\varphi_i) \Big) \Big\} \;.$$

The function $\mathrm{defer} : V \times \mathrm{Ev} \to \mathrm{Fml}^{\perp}$ *is defined as follows:*

$$\mathrm{defer}(x, \varphi) := \begin{cases} \psi & \text{if } \exists k \in \mathbb{N}_0.\, \exists \varphi_0, \ldots, \varphi_k \in \mathrm{Fml}.\, \Big(\varphi_0 = \varphi \;\&\; \varphi_k = \psi \;\&\; \\ & \quad \forall i \in \{0, \ldots, k-1\}.\, \big(\varphi_i \in \mathrm{Ev} \;\&\; \mathrm{ann}_x(\varphi_i) = \varphi_{i+1}\big) \;\&\; \\ & \quad \big(\varphi_k \notin \mathrm{Ev} \text{ or } \mathrm{ann}_x(\varphi_k) = \perp\big)\Big) \\ \perp & \text{otherwise.} \end{cases}$$

The function $\mathrm{getChild}(x, l)$ retrieves a particular child of x. It is easy to see that, during the algorithm, the child is always unique if it exists.

Intuitively, the function $\mathrm{reach}(\mathrm{prs}, x, \varphi)$ computes all eventualities which can be "reached" from φ inside x according to prs. If a potential rescuer (x, ψ) is

Procedure `is-sat`(ϕ) for testing whether a formula ϕ is satisfiable

```
Input: a formula $\phi \in \mathrm{Fml}$ in negation normal form
Output: true iff $\phi$ is satisfiable

$G :=$ a new empty graph;    $\mathrm{idx} := 1$
let $d \in \mathrm{APrg}$ be a dummy atomic program which does not occur in $\phi$
$\mathrm{rt} := \text{create-new-node}(\{\langle d\rangle\phi\}, \mathrm{ann}^{\perp}, \perp, \perp, \perp, \mathtt{unexp})$
insert rt in $G$
while one of the rules in Table 2 is applicable do
    apply any one of the applicable rules in Table 2
if $\mathrm{sts}_{\mathrm{rt}} = \mathtt{open}(\cdot,\cdot)$ then return true else return false
```

Table 2. Rules used in the procedure `is-sat`

Rule 1:	Some node x has not been expanded yet.
Condition:	$\exists x \in V.\ \mathrm{sts}_x = \mathtt{unexp}$
Action:	$\mathtt{expand}(x)$
Rule 2:	The status of some node x is still undefined.
Condition:	$\exists x \in V.\ \mathrm{sts}_x = \mathtt{undef}$
Action:	$\mathrm{sts}_x := \mathtt{det\text{-}status}(x)$ & $\mathrm{idx}_x := \mathrm{idx}$ & $\mathrm{idx} := \mathrm{idx} + 1$
Rule 3:	Some open node x is not up-to-date.
Condition:	$\exists x \in V.\ \mathtt{open}(\cdot,\cdot) = \mathrm{sts}_x \neq \mathtt{det\text{-}status}(x)$
Action:	$\mathrm{sts}_x := \mathtt{det\text{-}status}(x)$
Rule 4:	All nodes are up-to-date, and some x has an unfulfilled eventuality φ.
Condition:	Rule 3 is not applicable and
	$\exists x \in V.\ \mathrm{sts}_x = \mathtt{open}(\mathrm{prs}_x, \mathrm{alt}_x)$ & $\exists \varphi \in \mathrm{Ev} \cap \Gamma_x.\ \mathrm{prs}_x(\varphi) = \emptyset$
Action:	$\mathrm{sts}_x := \mathtt{closed}(\mathrm{alt}_x)$

contained in $\mathrm{prs}(\varphi)$, the potential rescuers of ψ are somehow relevant for φ at x. Therefore ψ itself is relevant for φ at x. The function $\mathrm{reach}(\mathrm{prs}, x, \varphi)$ computes exactly the transitive closure of this relevance relation.

Intuitively, the function $\mathrm{defer}(x, \varphi)$ follows the "ann_x-chain". That is, it computes $\varphi_1 := \mathrm{ann}_x(\varphi)$, $\varphi_2 := \mathrm{ann}_x(\varphi_1)$, and so on. There are two possible outcomes. The first outcome is that we eventually encounter a φ_k which is either not an eventuality or has $\mathrm{ann}_x(\varphi_k) = \perp$. Consequently, we cannot follow the "ann_x-chain" any more. In this case we stop and return $\mathrm{defer}(x, \varphi) := \varphi_k$. The second outcome is that we can follow the "ann_x-chain" indefinitely. Then, as Γ_x is finite, there must exist a cycle $\varphi_0, \ldots, \varphi_n, \varphi_0$ of eventualities such that $\mathrm{ann}_x(\varphi_i) = \varphi_{i+1}$ for all $0 \le i < n$, and $\mathrm{ann}_x(\varphi_n) = \varphi_0$. In this case we say that x (or Γ_x) contains an *"at a world" cycle* and return $\mathrm{defer}(x, \varphi) := \perp$.

Next we comment on all procedures given in pseudocode.

Procedure `is-sat`(ϕ) is invoked to determine whether a formula $\phi \in \mathrm{Fml}$ in negation normal form is satisfiable. It creates a root node rt and initialises the graph G to contain only rt. The dummy program d is used to make rt a state so that each node in G which is not a state has a parent state. The global variable idx is used to set the time stamps of the nodes accordingly.

Procedure `expand`(x) for expanding a node x

```
Input: a node $x \in V$ with $\mathrm{sts}_x = \texttt{unexp}$
if $\exists\varphi \in \Gamma_x.\ \sim\varphi \in \Gamma_x$ or $(\varphi \in \mathrm{Ev}$ & $\mathrm{defer}(x,\varphi) = \bot)$ then
    $\mathrm{idx}_x := \mathrm{idx}$;    $\mathrm{idx} := \mathrm{idx} + 1$;    $\mathrm{sts}_x := \texttt{closed}(\emptyset)$
else (* $x$ does not contain a contradiction *)
    $\mathrm{sts}_x := \texttt{undef}$
    if $\mathrm{pst}_x = \bot$ then (* $x$ is a state *)
        let $\langle l_1\rangle\varphi_1, \dots, \langle l_k\rangle\varphi_k$ be all of the $\langle\mathrm{lp}\rangle$-formulae in $\Gamma_x$
        for $i \longleftarrow 1$ to $k$ do
            $\Gamma_i := \{\varphi_i\} \cup \{\psi \mid [l_i]\psi \in \Gamma_x\}$
            $y_i := \text{create-new-node}(\Gamma_i, \mathrm{ann}^\bot, x, l_i, \bot, \texttt{unexp})$
            insert $y_i$, and an edge from $x$ to $y_i$ labelled with $\langle l_i\rangle\varphi_i$, into $G$
    else if $\exists\alpha \in \Gamma_x.\ \{\alpha_1, \dots, \alpha_k\} \not\subseteq \Gamma_x$ or $(\alpha \in \mathrm{Ev}$ & $\mathrm{ann}_x(\alpha) = \bot)$ then
        $\Gamma := \Gamma_x \cup \{\alpha_1, \dots, \alpha_k\}$
        ann := if $\alpha \in \mathrm{Ev}$ then $\mathrm{ann}_x[\alpha \mapsto \alpha_1]$ else $\mathrm{ann}_x$
        $y := \text{create-new-node}(\Gamma, \mathrm{ann}, \mathrm{pst}_x, \mathrm{ppr}_x, \bot, \texttt{unexp})$
        insert $y$, and an edge from $x$ to $y$, into $G$
    else if $\exists\beta \in \Gamma_x.\ \{\beta_1, \beta_2\} \cap \Gamma_x = \emptyset$ or $(\beta \in \mathrm{Ev}$ & $\mathrm{ann}_x(\beta) = \bot)$ then
        for $i \longleftarrow 1$ to $2$ do
            $\Gamma_i := \Gamma_x \cup \{\beta_i\}$
            $\mathrm{ann}_i :=$ if $\beta \in \mathrm{Ev}$ then $\mathrm{ann}_x[\beta \mapsto \beta_i]$ else $\mathrm{ann}_x$
            $y_i := \text{create-new-node}(\Gamma_i, \mathrm{ann}_i, \mathrm{pst}_x, \mathrm{ppr}_x, \bot, \texttt{unexp})$
            insert $y_i$, and an edge from $x$ to $y_i$, into $G$
    else (* $x$ is a special node *)
        if $\exists y \in V.\ \Gamma_y = \Gamma_x$ & $\mathrm{pst}_y = \bot$ then (* state already exists in $G$ *)
            insert an edge from $x$ to $y$ labelled with cs into $G$
        else (* state does not exist in $G$ yet *)
            $y := \text{create-new-node}(\Gamma_x, \mathrm{ann}^\bot, \bot, \bot, \bot, \texttt{unexp})$
            insert $y$, and an edge from $x$ to $y$ labelled with cs, into $G$
```

While at least one of the rules in Table 2 is applicable, that is its condition is true, the algorithm applies any applicable rule. If no rules are applicable, the algorithm returns satisfiable iff rt is open.

Rule 1 picks an unexpanded node and expands it. Rule 2 picks an expanded but undefined node and computes its (initial) status. It also sets the correct time stamp. Rule 3 picks an open node whose status has changed and recomputes its status. Its meaning is, that if we compute `det-status`(x) on the current graph then its result is different from the value in sts_x, and consequently, we update sts_x accordingly. Rule 4 is only applicable if all nodes are up-to-date. It picks an open node containing an eventuality φ which is currently not fulfilled in the graph and which does not have any potential rescuers either. As this indicates that φ can never be fulfilled, the node is closed.

This description leaves several questions open, most notably: "How do we check efficiently whether Rule 3 is applicable?" and "Which rule should be taken if several rules are applicable?". We address these issues in Section 5.

Procedure `det-status`(x) for determining the status of a node x

```
Input: a node $x \in V$ with $\texttt{unexp} \neq \mathrm{sts}_x \neq \texttt{closed}(\cdot)$
if $x$ is an $\alpha$-or a $\beta$-node then $\mathrm{sts}_x := \texttt{det-sts-}\beta(x)$
else if $x$ is a state then $\mathrm{sts}_x := \texttt{det-sts-state}(x)$
else (* $x$ is a special node, in particular $\mathrm{pst}_x \neq \bot \neq \mathrm{ppr}_x$ *)
    $\Gamma_{\mathrm{alt}} := \{\varphi \mid [\mathrm{ppr}_x^{\smile}]\varphi \in \Gamma_x\} \setminus \Gamma_{\mathrm{pst}_x}$
    if $\Gamma_{\mathrm{alt}} = \emptyset$ then $\mathrm{sts}_x := \texttt{det-sts-spl}(x)$ else $sts_x := \texttt{closed}(\{\Gamma_{\mathrm{alt}}\})$
```

Procedure `expand`(x) expands a node x. If Γ_x contains an immediate contradiction or an "at a world" cycle then we close x and set the time stamp accordingly. For the other cases, we assume implicitly that Γ_x does not contain either of these.

If x is a state, that is $\mathrm{pst}_x = \bot$, then we do the following for each $\langle\mathrm{lp}\rangle$-formula $\langle l_i\rangle\varphi_i$. We create a new node y_i whose associated set contains φ_i and all ψ such that $[l_i]\psi \in \Gamma_x$. As none of the eventualities in Γ_{y_i} is reduced yet, there are no annotations. The parent state of y_i is obviously x and its parent program is l_i. In order to relate y_i to $\langle l_i\rangle\varphi_i$, we label the edge from x to y_i with $\langle l_i\rangle\varphi_i$. We call y_i the *successor* of $\langle l_i\rangle\varphi_i$.

If x is not a state and Γ_x contains an α-formula α whose decompositions are not in Γ_x, or which is an unannotated eventuality, we call x an *α-node*. In this case, we create a new node y whose associated set is the result of adding all decompositions of α to Γ_x. If α is an eventuality then ann_y extends ann_x by mapping α to α_1. The parent state and the parent program of y are inherited from x. Note that pst_x and ppr_x are defined as x is not a state. Also note that $\Gamma_y \supsetneq \Gamma_x$ or α is an eventuality which is annotated in ann_y but not in ann_x.

If x is neither a state nor an α-node and Γ_x contains a β-formula β such that neither of its immediate subformulae is in Γ_x, or such that β is an unannotated eventuality, we call x a *β-node*. For each decomposition β_i we do the following. We create a new node y_i whose associated set is the result of adding β_i to Γ_x. If β is an eventuality then ann_{y_i} extends ann_x by mapping α to β_i. The parent state and the parent program of y are inherited from x. Note that pst_x and ppr_x are defined as x is not a state. Also note that $\Gamma_{y_i} \supsetneq \Gamma_x$ or β is an eventuality which is annotated in ann_{y_i} but not in ann_x.

If x is neither a state nor an α-node nor a β-node, it must be fully saturated and we call it a *special node*. Intuitively, a special node sits between a saturation phase and a state and is needed to handle the "special" issue arising from converse programs, as explained in the overview. Like α- and β-nodes, special nodes have a unique parent state and a unique parent program. In this case we check whether there already exists a state y in G which has the same set of formulae as the special node. If such a state y exists, we link x to y; else we create such a state and link x to it. In both cases we label the edge with the marker cs since a special node can have several children (see below) and we want to uniquely identify the cs-child y of x. Note that there is only at most one state for each set of formulae and that states are always fully saturated since special nodes are.

Procedure `det-sts-`$\beta(x)$ for determining the status of an α- or a β-node

```
Input: an $\alpha$- or a $\beta$-node $x \in V$ with $\texttt{unexp} \neq \mathrm{sts}_x \neq \texttt{closed}(\cdot)$
Output: the new status of $x$

let $y_1, \dots, y_k \in V$ be all children of $x$
alt := $\bigcup_{i=1}^{k} \mathrm{alt}_{y_i}$
if $\forall i \in \{1, \dots, k\}.\ \mathrm{sts}_{y_i} = \texttt{closed}(\cdot)$ then return $\texttt{closed}(\mathrm{alt})$
else (* at least one child is not closed *)
    prs := $\mathrm{prs}^{\perp}$
    foreach $\varphi \in \Gamma_x \cap \mathrm{Ev}$ do
        for $i \longleftarrow 1$ to $k$ do $\Lambda_i := \texttt{det-prs-child}(x, y_i, \varphi)$
        $\Lambda :=$ if $\exists i \in \{1, \dots, k\}.\ \Lambda_i = \perp$ then $\perp$ else $\bigcup_{i=1}^{k} \Lambda_i$
        prs := prs[$\varphi \mapsto \Lambda$]
    prs' := $\texttt{filter}(x, \mathrm{prs})$
    return $\texttt{open}(\mathrm{prs}', \mathrm{alt})$
```

Procedure `det-status`(x) determines the current status of a node x. Its result will always be `closed`$(\cdot)$ or `open`$(\cdot, \cdot)$. If x is an α/β-node or a state, the procedure just calls the corresponding sub-procedure. If x is a special node, we determine the set Γ_{alt} of all formulae φ such that $[\mathrm{ppr}_x^{\smile}]\varphi$ is in Γ_x but φ is not in the set of the parent state of x. If there is no such formula, that is Γ_{alt} is the empty set, we say that x is *compatible* with its parent state pst_x. Note that incompatibilities can only arise because of converse programs.

If x is compatible with pst_x, all is well, so we determine its status via the corresponding sub-procedure. Else we cannot connect pst_x to a state with Γ_x assigned to it in the putative model as explained in the overview, and, thus, we can close x. That does not, however, mean that pst_x is unsatisfiable; maybe it is just missing some formulae. We cannot extend pst_x directly as this may have side-effects elsewhere; but to tell pst_x what went wrong, we remember Γ_{alt}. The meaning is that if we create an alternative node for pst_x by adding the formulae in Γ_{alt}, we might be more successful in building an interpretation.

Procedure `det-sts-`$\beta(x)$ computes the status of an α- or a β-node $x \in V$. For this task, an α-node can be seen as a β-node with exactly one child. The set of alternative sets of x is the union of the sets of alternative sets of all children. If all children of x are closed then x must also be closed. Otherwise we compute the set of potential rescuers for each eventuality φ in Γ_x as follows. For each child y_i of x we determine the potential rescuers of φ which result from following y_i by invoking `det-prs-child`. If the set of potential rescuers corresponding to some y_i is $\perp$ then φ can currently be fulfilled via y_i and $\mathrm{prs}_x(\varphi)$ is set to $\perp$. Else φ cannot currently be fulfilled in G, but each child returned a set of potential rescuers, and the set of potential rescuers for φ is their union. Finally, we deal with potential rescuers in prs of the form (x, χ) for some $\chi \in \mathrm{Ev}$ by calling `filter`.

Procedure `det-sts-state`(x) computes the status of a state $x \in V$. We obtain the successors for all $\langle\mathrm{lp}\rangle$-formulae in Γ_x. If any successor is closed then x is closed with the same set of alternative sets. Else the set of alternative sets of x is the union of the sets of alternative sets of all children and we compute the potential

Procedure `det-sts-state`(x) for determining the status of a state

```
Input: a state $x \in V$ with unexp $\neq \mathrm{sts}_x \neq$ closed$(\cdot)$
Output: the new status of $x$

let $\langle l_1\rangle\varphi_1, \dots, \langle l_k\rangle\varphi_k$ be all of the $\langle \mathrm{lp}\rangle$-formulae in $\Gamma_x$
for $i \longleftarrow 1$ to $k$ do $y_i :=$ getChild$(x, \langle l_i\rangle\varphi_i)$
if $\exists i \in \{1, \dots, k\}.\ \mathrm{sts}_{y_i} =$ closed(alt) then return closed(alt)
else (* no child is closed *)
    alt $:= \bigcup_{i=1}^{k} \mathrm{alt}_{y_i}$
    prs $:= \mathrm{prs}^{\perp}$
    for $i \longleftarrow 1$ to $k$ do
        if $\varphi_i \in \mathrm{Ev}$ then
            $\Lambda :=$ det-prs-child$(x, y_i, \varphi_i)$
            prs $:= \mathrm{prs}[\langle l_i\rangle\varphi_i \mapsto \Lambda]$
    prs$'$ $:=$ filter$(x, \mathrm{prs})$
    return open$(\mathrm{prs}', \mathrm{alt})$
```

rescuers for each eventuality $\langle l_i\rangle\varphi_i$ in Γ_x by invoking `det-prs-child`. Finally, we deal with potential rescuers in prs of the form (x, χ) for some $\chi \in \mathrm{Ev}$ by calling `filter`. Note that we do not consider eventualities which are not $\langle \mathrm{lp}\rangle$-formulae. The intuitive reason is that the potential rescuers of such eventualities are determined by following the annotation chain (see below). However, different special nodes which have the same set, and hence all link to x, might have different annotations. Hence we cannot (and do not need to) fix the potential rescuer sets for eventualities in x which are not $\langle \mathrm{lp}\rangle$-formulae.

Procedure `det-sts-spl`(x) computes the status of a special node $x \in V$. First, we retrieve the state y_0 corresponding to x, namely the unique cs-child of x. For all alternative sets Γ_i of y_0 we do the following. If there does not exist a child of x such that the corresponding edge is labelled with Γ_i, we create a new node y_i whose associated set is the result of adding the formulae in Γ_i to Γ_x. The annotations, the parent state, and the parent program of y_i are inherited from x. We label the new edge from x to y_i with Γ_i. In other words we unpack the information stored in the alternative sets in alt_{y_0} into actual nodes which are all children of x. Note that each $\Gamma_i \neq \emptyset$ by construction in `det-status`. Some children of x may not be referenced from alt_{y_0}, but we consider them anyway.

The set of alternative sets of x is the union of the sets of alternative sets of all children; with the exception of y_0 since the alternative sets of y_0 are not related to pst_x but affect x directly as we have seen. If all children of x are closed then x must also be closed. Otherwise we compute the set of potential rescuers for each eventuality φ in Γ_x as follows.

First, we determine $\varphi' := \mathrm{defer}(x, \varphi)$. Note that φ' is defined because the special node x cannot contain an "at a world" cycle by definition. If φ' is not an eventuality then φ' is fulfilled in x and $\mathrm{prs}(\varphi)$ remains $\perp$. If φ' is an eventuality, it must be a $\langle \mathrm{lp}\rangle$-formula as x is a special node. We use φ' instead of φ since only $\langle \mathrm{lp}\rangle$-formula have a meaningful interpretation in prs_{y_0} (see above). For each child y_i of x we determine the potential rescuers of φ' by invoking

Procedure `det-sts-spl(x)` for determining the status of a special node

```
**Input**: a special node $x \in V$ with $\texttt{unexp} \neq \mathrm{sts}_x \neq \texttt{closed}(\cdot)$
**Output**: the new status of $x$

$y_0 := \mathrm{getChild}(x, \mathrm{cs})$
let $\Gamma_1, \dots, \Gamma_j$ be all the sets in the set $\mathrm{alt}_{y_0}$
for $i \longleftarrow 1$ to $j$ do
    $y_i := \mathrm{getChild}(x, \Gamma_i)$
    if $y_i = \bot$ then (* child does not exist *)
        $y_i :=$ create-new-node$(\Gamma_x \cup \Gamma_i, \mathrm{ann}_x, \mathrm{pst}_x, \mathrm{ppr}_x, \bot, \texttt{unexp})$
        insert $y_i$, and an edge from $x$ to $y_i$ labelled with $\Gamma_i$, into $G$

let $y_{j+1}, \dots, y_k$ be all the remaining children of $x$
$\mathrm{alt} := \bigcup_{i=1}^{k} \mathrm{alt}_{y_i}$
if $\forall i \in \{0, \dots, k\}.\ \mathrm{sts}_{y_i} = \texttt{closed}(\cdot)$ then return $\texttt{closed}(\mathrm{alt})$
else (* at least one child is not closed *)
    $\mathrm{prs} := \mathrm{prs}^{\bot}$
    foreach $\varphi \in \Gamma_x \cap \mathrm{Ev}$ do
        $\varphi' := \mathrm{defer}(x, \varphi)$
        if $\varphi' \in \mathrm{Ev}$ then
            for $i \longleftarrow 0$ to $k$ do $\Lambda_i := \texttt{det-prs-child}(x, y_i, \varphi')$
            $\Lambda :=$ if $\exists i \in \{0, \dots, k\}.\ \Lambda_i = \bot$ then $\bot$ else $\bigcup_{i=0}^{k} \Lambda_i$
            $\mathrm{prs} := \mathrm{prs}[\varphi \mapsto \Lambda]$
    $\mathrm{prs}' := \texttt{filter}(x, \mathrm{prs})$
    return $\texttt{open}(\mathrm{prs}', \mathrm{alt})$
```

`det-prs-child`. If the set of potential rescuers corresponding to some y_i is $\bot$ then φ' can currently be fulfilled via y_i and so $\mathrm{prs}_x(\varphi)$ is set to $\bot$. Otherwise φ' cannot currently be fulfilled in G, but each child returned a set of potential rescuers, and the set of potential rescuers for φ is their union. Finally, we deal with potential rescuers in prs of the form (x, χ) for some $\chi \in \mathrm{Ev}$ by calling `filter`.

Procedure `det-prs-child`(x, y, φ) determines whether an eventuality $\psi \in \Gamma_x$, which is not passed as an argument, can be fulfilled via y such that φ is part of the corresponding fulfilling path; or else which potential rescuers ψ can reach via y and φ. If y is closed, it cannot help to fulfil ψ as indicated by the empty set. If y is undefined or did not become defined before x then (y, φ) itself is a potential rescuer of x. Else, if φ can be fulfilled, i.e. $\mathrm{prs}_y(\varphi) = \bot$, then ψ can be fulfilled too, so we return $\bot$. Otherwise we invoke the procedure recursively on all potential rescuers in $\mathrm{prs}_y(\varphi)$. If at least one of these invocations returns $\bot$ then ψ can be fulfilled via y and φ and the corresponding rescuer in $\mathrm{prs}_y(\varphi)$. If all invocations return a set of potential rescuers, the set of potential rescuers for ψ is their union. The recursion is well-defined because if $(z_i, \varphi_i) \in \mathrm{prs}_y(\varphi)$ then either z_i is still undefined or z_i became defined later than y.

Each invocation of `det-prs-child` can be uniquely assigned to the invocation of `det-sts-`β, `det-sts-state`, or `det-sts-spl` which (possibly indirectly) invoked it. To meet our complexity bound, we require that under the same invocation of `det-sts-`β, `det-sts-state`, or `det-sts-spl`, the procedure `det-prs-child` is only executed at most once for each argument triple. Instead of executing it a second time with the same arguments, it uses the cached result of the first invocation. Since `det-prs-child` does not modify the graph, the second invocation would return the same result as the first one. An easy implementation of the cache is to store the result of `det-prs-child`(x, y, φ) in the node y together with φ and a unique id number for each invocation of `det-sts-`β, `det-sts-state`, or `det-sts-spl`.

Procedure `filter`(x, prs) deals with the potential rescuers for each eventuality of a node x which are of the form (x, ψ) for some $\psi \in \text{Ev}$. The second argument of `filter` is a provisional prs for x. If an eventuality $\varphi \in \Gamma_x$ is currently fulfillable in G there is nothing to be done, so let $(x, \psi) \in \text{prs}(\varphi)$. If $\psi = \varphi$ then (x, φ) cannot be a potential rescuer for φ in x and should not appear in $\text{prs}(\varphi)$. But what about potential rescuers of the form (x, ψ) with $\psi \neq \varphi$? Since we want the nodes in the potential rescuers to become defined later than x, we cannot keep (x, ψ) in $\text{prs}(\varphi)$; but we cannot just ignore the pair either.

Intuitively $(x, \psi) \in \text{prs}(\varphi)$ means that $\varphi \in \Gamma_x$ can "reach" $\psi \in \Gamma_x$ by following a loop in G which starts at x and returns to x itself. Thus if ψ can be fulfilled in G, so can φ; and all potential rescuers of ψ are also potential rescuers of φ. The function $\text{reach}(\text{prs}, x, \varphi)$ computes all eventualities in x which are "reachable" from φ in the sense above, where transitivity is taken into account. That is, it detects all self-loops from x to itself which are relevant for fulfilling φ. We add φ as it is not in $\text{reach}(\text{prs}, x, \varphi)$. If any of these eventualities is fulfilled in G then φ can be fulfilled and is consequently undefined in the resulting prs'. Otherwise we take all their potential rescuers whose nodes are not x.

Theorem 6 (Soundness, Completeness and Complexity). *Let $\phi \in \text{Fml}$ be a formula in negation normal form of size n. The procedure* `is-sat`(ϕ) *terminates, runs in* EXPTIME *in n, and ϕ is satisfiable iff* `is-sat`(ϕ) *returns true.*

5 Implementation, Optimisations, and Strategy

It should be fairly straightforward to implement our algorithm. It remains to show an efficient way to find nodes which are not up-to-date. It is not too hard to see that the status of a node x can become outdated only if its children change their status or `det-prs-child`$(x, y, \cdot)$ was invoked when x's previous status was determined and y now changes its status. If we keep track of nodes of the second kind by inserting additional "update"-edges as described in [7], we can use a queue for all nodes that might need updating. When the status of a node is modified, we queue all parents and all nodes linked by "update"-edges.

We have omitted several refinements from our description for clarity. The most important is that if a state s is closed, all non-states which have s as a parent state are ignorable since their status cannot influence any other node t unless t also has s as a parent state. Moreover, if every special node parent x of a state s' is incompatible or itself has a closed parent state, then s' and the nodes having s' as parent state are ignorable. This applies transitively, but if s' gets a new parent whose parent state is not closed then s' becomes "active" again.

Another issue is which rule to choose if several are applicable. As we have seen, it is advantageous to close nodes as early as possible. Apart from immediate contradictions, we have Rule 4 which closes a node because it contains an unfulfillable eventuality. If we can apply Rule 4 early while the graph is still small, we might prevent big parts of the graph being built needlessly later. Trying to apply Rule 4 has several consequences on the strategy of how to apply rules.

First, it is important to keep all nodes up-to-date since Rule 4 is not applicable otherwise. Second, it is preferable that a node x cannot reach open nodes which became defined (or will be defined) after x did. Hence, we should try to use Rule 2 on a node only if all children are already defined.

6 An Example

To demonstrate how the algorithm works, we invoke it on the satisfiable toy formula $\langle a \rangle \phi$ where $\phi := \langle a* \rangle [a^-]p$. To save space, Fig. 1 only shows the core subgraph of the tableau. Remember that the order of rule applications is not fixed but the example will follow some of the guidelines given in Section 5.

The nodes in Fig. 1 are numbered in order of creation. The annotation ann is given using "$\rightsquigarrow$" in Γ. For example, in node (3), we have $\Gamma_3 = \{\ \phi,\ [a^-]p\ \}$, and ann_3 maps the eventuality ϕ to $[a^-]p$ and is undefined elsewhere. The bottom line of a node contains the parent state and the parent program on the left, and the time stamp on the right. We do not show the status of a node since it changes during the algorithm, but explain it in the text. If we write $\mathrm{sts}_x = \mathtt{open}(\Lambda, \cdot)$ where $\Lambda \subseteq V \times \mathrm{Ev}$, we mean that prs_x maps all eventualities in Γ_x, with the exception of non-$\langle \mathrm{lp} \rangle$-formulae if x is a state, to Λ and is undefined elsewhere.

We only consider the core subgraph of ϕ and start by expanding node (1) which creates (2). Then we expand (2) and create (3) and (4) which are both special nodes. Next we expand (3) and create the state (5). Expanding (5) creates no new nodes since Γ_5 contains no $\langle \mathrm{lp} \rangle$-formula. Now we define (5) and then (3). This results in setting $\mathrm{sts}_5 := \mathtt{open}(\mathrm{prs}^{\perp}, \emptyset)$ according to `det-sts-state`, and $\mathrm{sts}_3 := \mathtt{closed}(\{p\})$ since (3) is not compatible with its parent state (1). Expanding (4) inserts the edge from (4) to (1) and defining (4) sets $\mathrm{sts}_4 := \mathtt{open}(\{(1, \langle a \rangle \phi)\}, \emptyset)$ according to `det-sts-spl`. Note that (6) does not exist yet. Next we define (2) and then (1) which results in setting $\mathrm{sts}_2 := \mathtt{open}(\{(1, \langle a \rangle \phi)\}, \{p\})$ according to $\mathtt{det\text{-}sts\text{-}}\beta$ and $\mathrm{sts}_1 := \mathtt{open}(\emptyset, \{p\})$ thanks to `filter`.

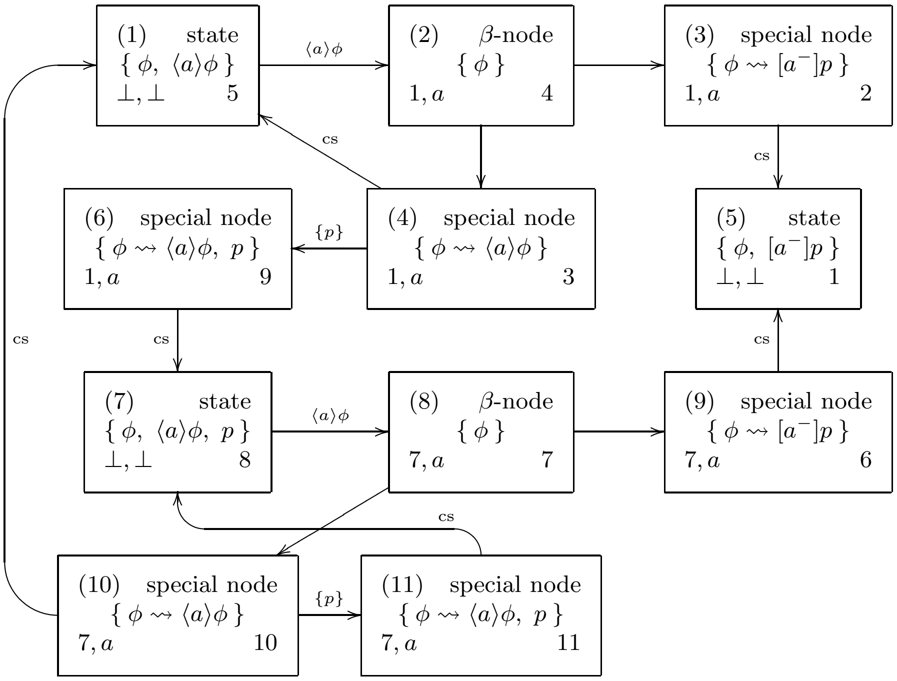
Note that $\langle a \rangle \phi \in \Gamma_1$ has an empty set of potential rescuers. In PDL, we could thus close (1), but converse programs complicate matters for CPDL as reflected by the fact that Rule 4 is not applicable for (1) because (4) is not up-to-date. Updating (4) creates (6) and sets $\mathrm{sts}_4 := \mathtt{open}(\{(1, \langle a \rangle \phi), (6, \langle a \rangle \phi)\}, \emptyset)$. Updating (2) and then (1) sets $\mathrm{sts}_2 := \mathtt{open}(\{(1, \langle a \rangle \phi), (6, \langle a \rangle \phi)\}, \{p\})$ and $\mathrm{sts}_1 := \mathtt{open}(\{(6, \langle a \rangle \phi)\}, \{p\})$. Now all nodes are up-to-date, but Rule 4 is not applicable for (1) because the set of potential rescuers for ϕ is no longer empty.

Next we expand (6), which creates (7), then (7), which creates (8), then (8), which creates (9) and (10), and finally (9), which creates no new nodes. Node (9)

Procedure `det-prs-child`(x, y, φ) for passing a prs-entry of a child to a parent

Input: two nodes $x, y \in V$ and a formula $\varphi \in \Gamma_y \cap \mathrm{Ev}$
Output: $\bot$ or a set of node-formula pairs
Remark: if `det-prs-child`(x, y, φ) has been invoked before with exactly the same arguments and *under the same invocation of* `det-sts-`β, `det-sts-state` *or* `det-sts-spl`, the procedure is not executed a second time but returns the cached result of the first invocation. We do not model this behaviour explicitly in the pseudocode.

```
if $\mathrm{sts}_y = \mathtt{closed}(\cdot)$ then return $\emptyset$
else if $\mathrm{sts}_y = \mathtt{unexp}$ or $\mathrm{sts}_y = \mathtt{undef}$ or not $y \sqsubset x$ then return $\{(y, \varphi)\}$
else (* $\mathrm{sts}_y = \mathtt{open}(\cdot, \cdot)$ & $y \sqsubset x$ *)
    if $\mathrm{prs}_y(\varphi) = \bot$ then return $\bot$
    else (* $\mathrm{prs}_y(\varphi)$ is defined *)
        let $(z_1, \varphi_1), \ldots, (z_k, \varphi_k)$ be all of the pairs in $\mathrm{prs}_y(\varphi)$
        for $i \longleftarrow 1$ to $k$ do $\Lambda_i :=$ det-prs-child$(x, z_i, \varphi_i)$
        if $\exists j \in \{1, \ldots, k\}.\ \Lambda_j = \bot$ then return $\bot$ else return $\bigcup_{i=1}^{k} \Lambda_i$
```



(1) state $\{\phi,\ \langle a\rangle\phi\}$ $\bot,\bot$ 5
(2) β-node $\{\phi\}$ $1,a$ 4
(3) special node $\{\phi \rightsquigarrow [a^-]p\}$ $1,a$ 2
(4) special node $\{\phi \rightsquigarrow \langle a\rangle\phi\}$ $1,a$ 3
(5) state $\{\phi,\ [a^-]p\}$ $\bot,\bot$ 1
(6) special node $\{\phi \rightsquigarrow \langle a\rangle\phi,\ p\}$ $1,a$ 9
(7) state $\{\phi,\ \langle a\rangle\phi,\ p\}$ $\bot,\bot$ 8
(8) β-node $\{\phi\}$ $7,a$ 7
(9) special node $\{\phi \rightsquigarrow [a^-]p\}$ $7,a$ 6
(10) special node $\{\phi \rightsquigarrow \langle a\rangle\phi\}$ $7,a$ 10
(11) special node $\{\phi \rightsquigarrow \langle a\rangle\phi,\ p\}$ $7,a$ 11
$\langle a\rangle\phi$
$\{p\}$
cs

Fig. 1. An example: The graph G just before setting the status of node (2)

Procedure `filter(`x, prs`)` for handling self-loops in prs chains in G

```
Input: a node $x \in V$ and a function $\mathrm{prs} : \mathrm{Ev} \to (\mathscr{P}(V \times \mathrm{Ev}))^{\perp}$
Output: prs where self-loops have been handled
$\mathrm{prs}' := \mathrm{prs}^{\perp}$
foreach $\varphi \in \Gamma_x \cap \mathrm{Ev}$ such that $\mathrm{prs}(\varphi) \neq \perp$ do
    $\Delta := \{\varphi\} \cup \mathrm{reach}(\mathrm{prs}, x, \varphi)$
    if not $\exists \chi \in \Delta.\ \mathrm{prs}(\chi) = \perp$ then
        $\Lambda := \bigcup_{\chi \in \Delta} \{(z, \psi) \in \mathrm{prs}(\chi) \mid z \neq x\}$
        $\mathrm{prs}' := \mathrm{prs}'[\varphi \mapsto \Lambda]$
return $\mathrm{prs}'$
```

is similar to (3), but unlike (3), it is compatible with its parent state (7) which results in $\mathrm{sts}_9 :=$ `open`$(\perp, \emptyset)$. Using our strategy from the last section, we would now expand (10) so that (8) can become defined after both its children became defined. Since (9) fulfils all its eventualities, we choose to define (8) instead and set $\mathrm{sts}_8 :=$ `open`$(\perp, \emptyset)$. Next we define (7) and then (6) which sets $\mathrm{sts}_7 :=$ `open`$(\perp, \emptyset)$ and $\mathrm{sts}_6 :=$ `open`$(\perp, \emptyset)$. The status of (4) is not affected since (6) was defined after (4), giving "(6) $\not\sqsubseteq$ (4)" in `det-prs-child`(4, 6, $\langle a \rangle \phi$).

We expand (10) which inserts the edge from (10) to (1). Then we define (10) which creates (11) and sets $\mathrm{sts}_{10} :=$ `open`$(\perp, \emptyset)$. Note that the invocation of `det-prs-child`(10, 1, $\langle a \rangle \phi$) in the invocation `det-sts-spl`(10) leads to the recursive invocation `det-prs-child`(10, 6, $\langle a \rangle \phi$). Expanding and defining (11) yields $\mathrm{sts}_{11} :=$ `open`$(\perp, \emptyset)$. Finally, no rule is applicable in the shown subgraph.

References

1. Vardi, M.Y.: The taming of converse: Reasoning about two-way computations. In: Parikh, R. (ed.) Logic of Programs 1985. LNCS, vol. 193, pp. 413–424. Springer, Heidelberg (1985)
2. De Giacomo, G., Massacci, F.: Combining deduction and model checking into tableaux and algorithms for Converse-PDL. Inf. and Comp. 162, 117–137 (2000)
3. Nguyen, L.A., Szałas, A.: An optimal tableau decision procedure for Converse-PDL. In: Proc. KSE-09, pp. 207–214. IEEE Computer Society, Los Alamitos (2009)
4. Lange, M., Stirling, C.: Focus games for satisfiability and completeness of temporal logic. In: Proc. LICS-01, pp. 357–365. IEEE Computer Society, Los Alamitos (2001)
5. Lange, M.: Satisfiability and completeness of Converse-PDL replayed. In: Günter, A., Kruse, R., Neumann, B. (eds.) KI 2003. LNCS (LNAI), vol. 2821, pp. 79–92. Springer, Heidelberg (2003)
6. Vardi, M., Wolper, P.: Automata theoretic techniques for modal logics of programs. Journal of Computer and System Sciences 32(2), 183–221 (1986)
7. Goré, R., Widmann, F.: An optimal on-the-fly tableau-based decision procedure for PDL-satisfiability. In: Schmidt, R.A. (ed.) CADE 2009. LNCS, vol. 5663, pp. 437–452. Springer, Heidelberg (2009)
8. Goré, R., Widmann, F.: Sound global state caching for ALC with inverse roles. In: Giese, M., Waaler, A. (eds.) TABLEAUX 2009. LNCS, vol. 5607, pp. 205–219. Springer, Heidelberg (2009)

Terminating Tableaux for Hybrid Logic with Eventualities

Mark Kaminski and Gert Smolka

Saarland University, Saarbrücken, Germany

Abstract. We present the first terminating tableau system for hybrid logic with eventualities. The system is designed as a basis for gracefully degrading reasoners. Eventualities are formulas of the form $\Diamond^* s$ that hold for a state if it can reach in $n \geq 0$ steps a state satisfying the formula s. The system is prefix-free and employs a novel clausal form that abstracts away from propositional reasoning. It comes with an elegant correctness proof. We discuss some optimizations for decision procedures.

1 Introduction

We consider basic modal logic extended with nominals and eventualities. We call this logic H^*. Nominals are formulas of the form x that hold exactly for the state x. Eventualities are formulas of the form $\Diamond^* s$ that hold for a state if it can reach in $n \geq 0$ steps a state satisfying the formula s. Nominals equip modal logic with equality and are the characteristic feature of hybrid logic [4,2]. Eventualities extend modal logic with reflexive transitive closure and are an essential feature of PDL [10,12] and temporal logics [16,8,9]. One can see H^* either as hybrid logic extended with eventualities or as stripped-down PDL extended with nominals. Due to the inductive nature of eventualities, H^* is not compact (consider $\Diamond^* \neg p$, p, $\Box p$, $\Box\Box p$, $\ldots$). On the other hand, the satisfiability problem for H^* is decidable and EXPTIME-complete. Decidability can easily be shown with filtration [4]. Decidability in deterministic exponential time follows from a corresponding result for the hybrid μ-calculus [18], a logic that subsumes H^*. EXPTIME-hardness follows from Fischer and Ladner's [10] proof for PDL, which also applies to modal logic with eventualities. See Blackburn et al. [4] for a discussion and an elegant proof (Theorem 6.52).

We are interested in a terminating tableau system for H^* that can serve as a basis for gracefully degrading reasoners. Given that there are terminating tableau systems for both PDL [17,3,7,1,11] and hybrid logic [6,5,13,14,15], one would hope that coming up with a terminating system for H^* is not difficult. Once we attacked the problem we found it rather difficult. First of all, the approaches taken by the two families of systems are very different. Hybrid systems rely on fine-grained prefix-based propagation of equational information and lack the structure needed for checking eventualities. PDL systems do not provide the propagation needed for nominals and this propagation is in conflict with the and-or graph representation [17,11] and the existing techniques for proving correctness.

After some trial an error, we decided to first construct a terminating tableau system for modal logic with eventualities and nothing else. The goal was to obtain a simple system with a simple correctness proof that would scale to the extension with nominals. In the end we found a convincing solution that uses some new ideas. In contrast to existing systems for hybrid logic, our system is prefix-free and does not rely on fine-grained propagation of equational constraints. Following the PDL systems of Baader [3] and De Giacomo and Massacci [7], our system avoids a posteriori eventuality checking by disallowing bad loops. The novel feature of our system is the use of a clausal form that abstracts away from propositional reasoning and puts the focus on modal reasoning. This way termination and bad loop checking become obvious. The crucial part of the correctness proof, which shows that branches with bad loops can be safely ignored, employs the notion of a straight model. A straight model requires that the links on the branch make maximal progress towards the fulfillment of the eventuality they serve. The notion of a straight model evolved in work with Sigurd Schneider [19] and builds on an idea in Baader's [3] correctness proof (Proposition 4.7).

Due to the clausal form, the extension of our system to nominals is straightforward. When we add a new clause to a branch, we add to the new clause all literals that occur in clauses of the branch that have a nominal in common with the new clause. This takes care of nominal propagation. Clauses and links that are already on the branch remain unchanged. Our approach yields a novel and particularly simple tableau system for hybrid logic.

The paper is organized as follows. First, we introduce formulas, interpretations, and the clausal form. We then present the tableau system and its correctness proof in three steps, first for modal logic, then for hybrid logic, and finally for hybrid logic with eventualities. Finally, we discuss some optimizations for decision procedures.

2 The Logic

We assume that two kind of names, called *nominals* and *predicates*, are given. Nominals (x, y) denote states and predicates (p, q) denote sets of states. *Formulas* are defined as follows:

$$s ::= p \mid \neg p \mid s \wedge s \mid s \vee s \mid \Diamond s \mid \Box s \mid \Diamond^* s \mid \Box^* s \mid x \mid \neg x \mid @_x s$$

For simplicity we employ only a single transition relation and consider only formulas in negation normal form. Generalization of our results to multiple transition relations is straightforward. We use the notations $\Diamond^+ s := \Diamond\Diamond^* s$ and $\Box^+ s := \Box\Box^* s$. An *eventuality* is a formula of the form $\Diamond^* s$ or $\Diamond^+ s$. All other *diamond formulas* $\Diamond s$ are called *simple*. An *interpretation* $\mathcal{I}$ consists of the following components:

– A set $|\mathcal{I}|$ of *states*.
– A *transition relation* $\rightarrow_{\mathcal{I}} \subseteq |\mathcal{I}| \times |\mathcal{I}|$.

– A set $\mathcal{I}p \subseteq |\mathcal{I}|$ for every predicate p.
– A state $\mathcal{I}x \in |\mathcal{I}|$ for every nominal x.

We write $\rightarrow^*_{\mathcal{I}}$ for the reflexive transitive closure of $\rightarrow_{\mathcal{I}}$. The *satisfaction relation* $\mathcal{I}, a \models s$ between interpretations $\mathcal{I}$, states $a \in |\mathcal{I}|$, and formulas s is defined by induction on s:

$$\begin{array}{rcl@{\qquad}rcl}
\mathcal{I}, a \models p & \iff & a \in \mathcal{I}p & \mathcal{I}, a \models s \wedge t & \iff & \mathcal{I}, a \models s \text{ and } \mathcal{I}, a \models t \\
\mathcal{I}, a \models \neg p & \iff & a \notin \mathcal{I}p & \mathcal{I}, a \models s \vee t & \iff & \mathcal{I}, a \models s \text{ or } \mathcal{I}, a \models t \\
\mathcal{I}, a \models x & \iff & a = \mathcal{I}x & \mathcal{I}, a \models \Diamond s & \iff & \exists b:\ a \rightarrow_{\mathcal{I}} b \text{ and } \mathcal{I}, b \models s \\
\mathcal{I}, a \models \neg x & \iff & a \neq \mathcal{I}x & \mathcal{I}, a \models \Box s & \iff & \forall b:\ a \rightarrow_{\mathcal{I}} b \text{ implies } \mathcal{I}, b \models s \\
\mathcal{I}, a \models @_x s & \iff & \mathcal{I}, \mathcal{I}x \models s & \mathcal{I}, a \models \Box^* s & \iff & \forall b:\ a \rightarrow^*_{\mathcal{I}} b \text{ implies } \mathcal{I}, b \models s \\
 & & & \mathcal{I}, a \models \Diamond^* s & \iff & \exists b:\ a \rightarrow^*_{\mathcal{I}} b \text{ and } \mathcal{I}, b \models s
\end{array}$$

We interpret sets of formulas conjunctively. Given a set A of formulas, we write $\mathcal{I}, a \models A$ if $\mathcal{I}, a \models s$ for all formulas $s \in A$. An interpretation $\mathcal{I}$ *satisfies* (or is a *model* of) a formula s or a set A of formulas if there is a state $a \in |\mathcal{I}|$ such that $\mathcal{I}, a \models s$ or, respectively, $\mathcal{I}, a \models A$. A formula s (a set A) is *satisfiable* if s (A) has a model.

We say that two formulas s and t are *equivalent* and write $s \cong t$ if the equivalence $\mathcal{I}, a \models s \iff \mathcal{I}, a \models t$ holds for all interpretations $\mathcal{I}$ and all states $a \in |\mathcal{I}|$. Two important equivalences are $\Diamond^* s \cong s \vee \Diamond^+ s$ and $\Box^* s \cong s \wedge \Box^+ s$.

We write $\mathrm{H}^*_@$ for the full logic and define several sublogics:

K	$p \mid \neg p \mid s \wedge s \mid s \vee s \mid \Diamond s \mid \Box s \mid \Box^* s$
K^*	K extended with $\Diamond^* s$
H	K extended with x, $\neg x$
H^*	H extended with $\Diamond^* s$
$\mathrm{H}^*_@$	H^* extended with $@_x s$

Note that K is basic modal logic plus positive occurrences of $\Box^* s$, and H is basic hybrid logic plus positive occurrences of $\Box^* s$.

3 Clausal Form

We define a clausal form for our logic. The clausal form allows us to abstract from propositional reasoning and to focus on modal reasoning.

A *literal* is a formula of the form p, $\neg p$, $\Diamond s$, $\Box s$, x, $\neg x$, or $@_x s$. A *clause* (C, D) is a finite set of literals that contains no complementary pair (p and $\neg p$ or x and $\neg x$). Clauses are interpreted conjunctively. *Satisfaction of clauses* (i.e., $\mathcal{I}, X \models C$) is a special case of satisfaction of sets of formulas (i.e., $\mathcal{I}, X \models A$), which was defined in §2. For instance, the clause $\{p, \neg p\}$ is unsatisfiable. Note that every clause not containing literals of the forms $\Diamond s$ and $@_x s$ is satisfiable. We will show that for every formula one can compute $n \geq 0$ clauses such that the disjunction of the clauses is equivalent to the formula.

The *syntactic closure* $\mathcal{S}A$ of a set A of formulas is the least set of formulas that contains A and is closed under the rules

$$\frac{\neg s}{s} \qquad \frac{s \wedge t}{s\,,\, t} \qquad \frac{s \vee t}{s\,,\, t} \qquad \frac{\Diamond s}{s} \qquad \frac{\Box s}{s} \qquad \frac{\Box^* s}{s\,,\, \Box^+ s} \qquad \frac{\Diamond^* s}{s\,,\, \Diamond^+ s} \qquad \frac{@_x s}{x\,,\, s}$$

Note that $\mathcal{S}A$ is finite if A is finite, and that the size of $\mathcal{S}A$ is linear in the size of A (sum of the sizes of the formulas appearing as elements of A).

The *support relation* $C \rhd s$ between clauses C and formulas s is defined by induction on s:

$$\begin{aligned} C \rhd s &\iff s \in C \quad \text{if } s \text{ is a literal} \\ C \rhd s \wedge t &\iff C \rhd s \text{ and } C \rhd t \\ C \rhd s \vee t &\iff C \rhd s \text{ or } C \rhd t \\ C \rhd \Box^* s &\iff C \rhd s \text{ and } C \rhd \Box^+ s \\ C \rhd \Diamond^* s &\iff C \rhd s \text{ or } C \rhd \Diamond^+ s \end{aligned}$$

We say C *supports* s if $C \rhd s$. We write $C \rhd A$ and say C *supports* A if $C \rhd s$ for every $s \in A$. Note that $C \rhd D \iff D \subseteq C$ if C and D are clauses.

Proposition 3.1. *If $\mathcal{I}, a \models C$ and $C \rhd A$, then $\mathcal{I}, a \models A$.*

Proposition 3.2. *If $C \rhd A$ and $C \subseteq D$ and $B \subseteq A$, then $D \rhd B$.*

We define a function $\mathcal{D}$ that yields for every set A of formulas the set of all minimal clauses supporting A:

$$\mathcal{D}A \;:=\; \{\, C \mid C \rhd A \text{ and } \forall D \subseteq C\colon\;\; D \rhd A \text{ implies } D = C \,\}$$

We call $\mathcal{D}A$ the *DNF* of A.

Example 3.1. Consider $s = p \wedge q \vee \neg p \wedge q$. Then $\mathcal{D}\{s\} = \{\{p, q\}, \{\neg p, q\}\}$. Hence $\{q\} \not\rhd \{s\}$ even though q and s are equivalent.

If X is a set, we use the notation $X\,; x := X \cup \{x\}$.

Proposition 3.3.

1. $\mathcal{I}, a \models A \iff \exists C \in \mathcal{D}A\colon\;\; \mathcal{I}, a \models C$.
2. *If $C \in \mathcal{D}A$, then $C \subseteq \mathcal{S}A$.*
3. $C \rhd A \iff \exists D \in \mathcal{D}A\colon\;\; D \subseteq C$.
4. $\mathcal{D}(A\,; s) \subseteq \mathcal{D}(A\,; \Diamond^* s)$.

Proposition 3.4. *If A is a finite set of formulas, then $\mathcal{D}A$ is finite.*

Proof. The claim follows with Proposition 3.3 (2) since $\mathcal{S}A$ is finite. $\square$

The DNF of a finite set of formulas can be computed with the following tableau rules:

$$\frac{s \wedge t}{s\,,\,t} \qquad \frac{s \vee t}{s \mid t} \qquad \frac{\Box^* s}{s\,,\,\Box^+ s} \qquad \frac{\Diamond^* s}{s \mid \Diamond^+ s}$$

To obtain $\mathcal{D}A$, one develops A into a complete tableau. The literals of each open branch yield a clause. The minimal clauses obtained this way constitute $\mathcal{D}A$.

Let C and D be clauses. The *request of* C is $\mathcal{R}C := \{\, t \mid \Box t \in C \,\}$. We say D *realizes* $\Diamond s$ *in* C if $D \triangleright \mathcal{R}C\,; s$.

Proposition 3.5. *If $\Diamond s \in C$ and $\mathcal{I}$ satisfies C, then $\mathcal{I}$ satisfies some clause $D \in \mathcal{D}(\mathcal{R}C\,; s)$.*

Proof. Follows with Proposition 3.3(1). $\Box$

4 Tableaux for K

We start with a terminating tableau system for the sublogic K to demonstrate the basic ideas of our approach. A *branch* of the system is a finite and nonempty set of clauses. A *model of a branch* is an interpretation that satisfies all clauses of the branch. A branch is *satisfiable* if it has a model. Let Γ be a branch, C be a clause, and $\Diamond s$ be a literal. We say that

– Γ *realizes* $\Diamond s$ *in* C if $D \triangleright \mathcal{R}C\,; s$ for some clause $D \in \Gamma$.
– Γ *is evident* if Γ realizes $\Diamond s$ in C for all $\Diamond s \in C \in \Gamma$.

The *syntactic closure* $\mathcal{S}\Gamma$ of a branch Γ is the union of the syntactic closures of the clauses $C \in \Gamma$. Note that the syntactic closure of a branch is finite. Moreover, $C \subseteq \mathcal{S}\Gamma$ for all clauses $C \in \Gamma$.

Every evident branch describes a finite interpretation that satisfies all its clauses. The states of the interpretation are the clauses of the branch, and the transitions of the interpretation are the pairs (C, D) such that $D \triangleright \mathcal{R}C$.

Theorem 4.1 (Model Existence). *Every evident branch has a finite model.*

Proof. Let Γ be an evident branch and $\mathcal{I}$ be an interpretation as follows:

– $|\mathcal{I}| = \Gamma$
– $C \to_{\mathcal{I}} D \iff D \triangleright \mathcal{R}C$
– $C \in \mathcal{I}p \iff p \in C$

We show $\forall s \in \mathcal{S}\Gamma\ \forall C \in \Gamma\colon\ C \triangleright s \Longrightarrow \mathcal{I}, C \models s$ by induction on s. Let $s \in \mathcal{S}\Gamma$, $C \in \Gamma$, and $C \triangleright s$. We show $\mathcal{I}, C \models s$ by case analysis. The cases are all straightforward except possibly for $s = \Box^* t$. So let $s = \Box^* t$. Let $C = C_1 \to_{\mathcal{I}} \dots \to_{\mathcal{I}} C_n$. We show $\mathcal{I}, C_n \models t$ by induction on n. If $n = 1$, we have $C_n \triangleright s$ by assumption. Hence $C_n \triangleright t$ and the claim follows by the outer inductive hypothesis. If $n > 1$, we have $s \in \mathcal{R}C_1$ since $\Box s \in C_1$ since $C_1 \triangleright \Box s$ since $C_1 \triangleright s$. Thus $C_2 \triangleright s$ and the claim follows by the inner inductive hypothesis. $\Box$

The tableau system for K is obtained with a single rule.

> *Expansion Rule for K*
> If $\Diamond s \in C \in \Gamma$ and Γ does not realize $\Diamond s$ in C,
> then expand Γ to all branches $\Gamma\,; D$ such that $D \in \mathcal{D}(\mathcal{R}C; s)$.

The expansion rule for K has the obvious property that it applies to a branch if and only if the branch is not evident. It is possible that the expansion rule applies to a branch but does not produce an extended branch. We call a branch *closed* if this is the case. Note that a branch is closed if and only if it contains a clause C that contains a literal $\Diamond s$ such that the DNF of $\mathcal{R}C; s$ is empty.

Example 4.1. Consider the clause $C = \{\Diamond p,\ \Box\neg p\}$. Since $\Diamond p$ is not realized in C in $\Gamma = \{C\}$, the expansion rule applies to the branch Γ. Since $\mathcal{D}(\mathcal{R}C\,; p) = \emptyset$, the expansion rule fails to produce an extension of Γ. Thus Γ is closed.

Example 4.2. Here is a complete tableau for a clause C_1.

$$\begin{array}{r|l}
\multicolumn{2}{c}{C_1 = \{\Diamond\Diamond p,\ \Diamond(q \wedge \Diamond p),\ \Box(q \vee \Box\neg p)\}} \\
\hline
C_2 = \{\Diamond p, q\} & C_3 = \{\Diamond p, \Box\neg p\} \\
C_4 = \{p\} &
\end{array}$$

The development of the tableau starts with the branch $\{C_1\}$. Application of the expansion rule to $\Diamond\Diamond p \in C_1$ yields the branches $\{C_1, C_2\}$ and $\{C_1, C_3\}$. The branch $\{C_1, C_3\}$ is closed. Expansion of $\Diamond p \in C_2 \in \{C_1, C_2\}$ yields the evident branch $\{C_1, C_2, C_4\}$. Note that C_2 realizes $\Diamond(q \wedge \Diamond p)$ in C_1.

Theorem 4.2 (Termination). *The tableau system for K terminates.*

Proof. Let a branch Γ' be obtained from a branch Γ by the expansion rule such that $\Gamma \subsetneq \Gamma'$. By Proposition 3.3(2) we have $\mathcal{S}\Gamma' = \mathcal{S}\Gamma$ and $\Gamma \subsetneq \Gamma' \subseteq 2^{\mathcal{S}\Gamma'} = 2^{\mathcal{S}\Gamma}$. This suffices for termination since $\mathcal{S}\Gamma$ is finite. $\Box$

Theorem 4.3 (Soundness). *Let $\mathcal{I}$ be a model of a branch Γ and $\Diamond s \in C \in \Gamma$. Then there exists a clause $D \in \mathcal{D}(\mathcal{R}C\,; s)$ such that $\Gamma\,; D$ is a branch and $\mathcal{I}$ is a model of $\Gamma\,; D$.*

Proof. Follows with Proposition 3.5. $\Box$

We now have a tableau-based decision procedure that decides the satisfiability of branches. Given a branch Γ, the procedure either extends Γ to an evident branch that describes a model of Γ, or it constructs a closed tableau that proves that Γ is unsatisfiable. The procedure is recursive. It checks whether the current branch contains a diamond formula that is not yet realized. If no such formula exists, the branch is evident and the procedure terminates. Otherwise, the DNF of the body of such a formula and the request of the clause containing it are computed. If the DNF is empty, the branch is closed and thus unsatisfiable. Otherwise, the branch is expanded into as many branches as the DNF has clauses and the procedure continues recursively. The correctness of the procedure follows from the theorems and propositions stated above.

5 Tableaux for H

We now develop a terminating tableau system for the sublogic H, which extends K with nominals. The system is very different from existing systems for hybrid logic [6,5,13,14,15] since it does not employ prefixes. The key observation is that two clauses that contain a common nominal must be satisfied by the same state in every model of the branch. Thus if two clauses on a branch contain a common nominal, we can always add the union of the two clauses to the branch.

Proposition 5.1. *Suppose an interpretation $\mathcal{I}$ satisfies two clauses C and D that contain a common nominal $x \in C \cap D$. Then $\mathcal{I}$ satisfies $C \cup D$ and the set $C \cup D$ is a clause.*

We call a clause *nominal* if it contains a nominal. Let Γ be a set of clauses and A be a set of formulas. We define two notations to realize what we call *nominal propagation*:

$$\begin{aligned} A^\Gamma &:= A \cup \{\, s \mid \exists x \in A\ \exists C \in \Gamma\colon\ x \in C \wedge s \in C \,\} \\ \mathcal{D}^\Gamma A &:= \{\, C^\Gamma \mid C \in \mathcal{D}A \text{ and } C^\Gamma \text{ is a clause} \,\} \end{aligned}$$

Note that A^Γ is the least set of formulas that contains A and all clauses $C \in \Gamma$ that have a nominal in common with A. Thus $(A^\Gamma)^\Gamma = A^\Gamma$. Moreover, $A^\Gamma = A$ if A contains no nominal.

Proposition 5.2. *Let A be a set of formulas, $\mathcal{I}$ be a model of a branch Γ, and a be a state of $\mathcal{I}$. Then $\mathcal{I}, a \models A \iff \mathcal{I}, a \models A^\Gamma$.*

Proof. Follows with Proposition 5.1. □

The *branches* of the tableau system for H are finite and nonempty sets Γ of clauses that satisfy the following condition:

– *Nominal coherence:* If $C \in \Gamma$, then $C^\Gamma \in \Gamma$.

Satisfaction, *realization*, *evidence*, and the *syntactic closure* of branches are defined as for K. The *core* $\mathcal{C}\Gamma := \{\, C \in \Gamma \mid C^\Gamma = C \,\}$ of a branch Γ is the set of all clauses of Γ that are either maximal or not nominal.

Proposition 5.3. *Let Γ be a branch. Then:*

1. *$\mathcal{C}\Gamma$ is a branch.*
2. *An interpretation satisfies Γ iff it satisfies $\mathcal{C}\Gamma$.*
3. *Γ is evident iff $\mathcal{C}\Gamma$ is evident.*

Proof. Claims (1) and (2) are obvious, and (3) follows with Proposition 3.2. □

Theorem 5.1 (Model Existence). *Every evident branch has a finite model.*

Proof. Let Γ be an evident branch. Without loss of generality we can assume that for every nominal $x \in \mathcal{S}\Gamma$ there is a unique clause $C \in \mathcal{C}\Gamma$ such that $x \in C$ (add clauses $\{x\}$ as necessary). We choose an interpretation $\mathcal{I}$ that satisfies the conditions

– $|\mathcal{I}| = \mathcal{C}\Gamma$
– $C \rightarrow_{\mathcal{I}} D \iff D \triangleright \mathcal{R}C$
– $C \in \mathcal{I}p \iff p \in C$
– $\mathcal{I}x = C \iff x \in C$ for all $x \in \mathcal{S}\Gamma$

The last condition can be satisfied since Γ is nominally coherent. We show $\forall s \in \mathcal{S}\Gamma\ \forall C \in \mathcal{C}\Gamma\colon\ C \triangleright s \Longrightarrow \mathcal{I}, C \models s$ by induction on s. Let $s \in \mathcal{S}\Gamma$, $C \in \mathcal{C}\Gamma$, and $C \triangleright s$. We show $\mathcal{I}, C \models s$ by case analysis. The proof now proceeds as the proof of Theorem 4.1. The additional cases for nominals can be argued as follows.

Let $s = x$. Then $x \in C$ and thus $\mathcal{I}x = C$. Hence $\mathcal{I}, C \models s$.

Let $s = \neg x$. Then $\neg x \in C$ and thus $x \notin C$ and $\mathcal{I}x \neq C$. Hence $\mathcal{I}, C \models s$. □

Nominal coherence acts as an invariant for the tableau system for H. The expansion rule for H refines the expansion rule for K such that the invariant is maintained. The tableau system for H is obtained with the following rule.

Expansion Rule for H
If $\Diamond s \in C \in \mathcal{C}\Gamma$ and Γ does not realize $\Diamond s$ in C,
then expand Γ to all branches $\Gamma ; D$ such that $D \in \mathcal{D}^{\Gamma}(\mathcal{R}C; s)$.

As in the system for K, the expansion rule for H has the property that it applies to a branch if and only if the branch is not evident. Moreover, termination follows as for K. The adaption of the soundness theorem is also straightforward.

Theorem 5.2 (Soundness). *Let $\mathcal{I}$ be a model of a branch Γ and $\Diamond s \in C \in \Gamma$. Then there exists a clause $D \in \mathcal{D}^{\Gamma}(\mathcal{R}C\,; s)$ such that $\Gamma\,; D$ is a branch and $\mathcal{I}$ is a model of $\Gamma\,; D$.*

Proof. Since $\mathcal{I}$ satisfies C, we know by Proposition 3.5 that $\mathcal{I}$ satisfies some clause $D \in \mathcal{D}(\mathcal{R}C\,; s)$. The claim follows with Proposition 5.2. □

We have now arrived at a decision procedure for the sublogic H.

Example 5.1. Consider the following closed tableau.

$$\begin{tikzcd}
& \boxed{\Diamond(\Diamond\Diamond(x \wedge \neg p)\ \wedge\ (\Diamond(x \wedge p) \vee x \wedge p))}^{0} \arrow[dl] \arrow[dr] & \\
\boxed{\Diamond\Diamond(x \wedge \neg p),\ \Diamond(x \wedge p)}^{1} \arrow[d] & & \boxed{x,\ \Diamond\Diamond(x \wedge \neg p),\ p}^{2} \arrow[d] \\
\boxed{\Diamond(x \wedge \neg p)}^{3} \arrow[d] & & \boxed{\Diamond(x \wedge \neg p)}^{5} \\
\boxed{x,\ \neg p}^{4} & &
\end{tikzcd}$$

The numbers identifying the clauses indicate the order in which they are introduced. Once clause 4 is introduced, $\Diamond(x \wedge p)$ in clause 1 cannot be realized due to nominal propagation from clause 4. Hence the left branch is closed. The right branch is also closed since the diamond formula in clause 5 cannot be realized due to nominal propagation from clause 2.

The example shows that nominals have a severe impact on modal reasoning. The impact of nominals is also witnessed by the fact that in K the union of two branches is a branch while this is not the case in H (e.g., $\{\{x, p\}\}$ and $\{\{x, \neg p\}\}$).

Remark 5.1. To obtain the optimal worst-case run time for tableau provers, one must avoid recomputation. In the absence of nominals this can be accomplished with a minimal and-or graph representation [17,11]. Unfortunately, the minimal and-or graph representation is not compatible with nominal propagation. To see this, consider the minimal and-or graph representing the tableau of Example 5.1. This graph represents clauses 3 and 5 with a single node. This is not correct since the nominal context of the clauses is different. While clause 3 can be expanded, clause 5 cannot.

6 Evidence for H*

Next, we consider the sublogic H*, which extends H with eventualities $\Diamond^* s$. We define branches and evidence for H* and prove the corresponding model existence theorem. As one would expect, realization of eventualities $\Diamond^+ s$ is more involved than realization of simple diamond formulas.

Example 6.1. Consider the clause $C = \{\Diamond^+ \neg p,\ \Box^+ p,\ p\}$. We have $\mathcal{R}C = \{\Box^* p\}$. Hence $C \triangleright \mathcal{R}C\,; \Diamond^* \neg p$. If we extend our definitions for H to H*, the branch $\{C\}$ is evident. However, the clause C is not satisfiable.

To obtain an adequate notion of realization for eventualities, branches for H* will contain links in addition to clauses. A *link* is a triple $C(\Diamond s)D$ such that $\Diamond s \in C$ and $D \triangleright \mathcal{R}C; s$. A *quasi-branch* Γ is a finite and nonempty set of clauses and links such that Γ contains the clauses C and D if it contains a link $C(\Diamond s)D$. A *model of a quasi-branch* is an interpretation that satisfies all of its clauses. Note that a model is not required to satisfy the links of a quasi-branch. A quasi-branch is *satisfiable* if it has a model. A quasi-branch Γ *realizes* $\Diamond s$ *in* C if Γ contains some link $C(\Diamond s)D$. Quasi-branches can be drawn as graphs with links pointing from diamond formulas to clauses. Figure 1 shows three examples. The first two graphs describe models of the clauses. This is not the case for the rightmost graph where both clauses are unsatisfiable. Still, all diamond formulas are realized with links. The problem lies in the "bad loop" that leads from the lower clause to itself.

The notations A^Γ and $\mathcal{D}^\Gamma A$ are defined for quasi-branches in the same way as they are defined for the branches of H. Let Γ be a quasi-branch. A *path for* $\Diamond^+ s$ *in* Γ is a sequence $C_1 \dots C_n$ of clauses such that $n \geq 2$ and:

1. $\forall i \in [1, n]\colon \quad C_i^\Gamma = C_i$.
2. $\forall i \in [1, n-1]\ \exists D\colon \quad C_i(\Diamond^+ s)D \in \Gamma$ and $D^\Gamma = C_{i+1}$.
3. $\forall i \in [2, n-1]\colon \quad C_i \not\triangleright s$.

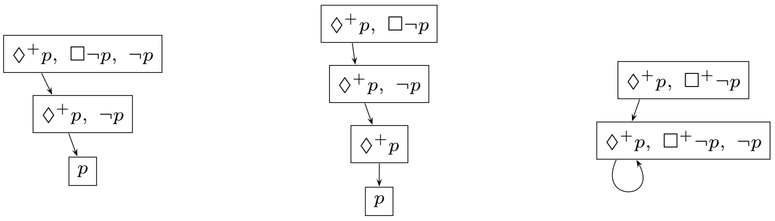

$\Diamond^+ p,\ \Box\neg p,\ \neg p$
$\Diamond^+ p,\ \neg p$
p
$\Diamond^+ p,\ \Box\neg p$
$\Diamond^+ p,\ \neg p$
$\Diamond^+ p$
p
$\Diamond^+ p,\ \Box^+\neg p$
$\Diamond^+ p,\ \Box^+\neg p,\ \neg p$

Fig. 1. Three quasi-branches drawn as graphs

A *run for* $\Diamond^+ s$ *in* C *in* Γ is a path $C \dots D$ for $\Diamond^+ s$ in Γ such that $D \rhd s$. A *bad loop for* $\Diamond^+ s$ *in* Γ is a path $C \dots C$ for $\Diamond^+ s$ in Γ such that $C \not\rhd s$. A *branch* is a quasi-branch Γ that satisfies the following conditions:

– *Nominal coherence:* If $C \in \Gamma$, then $C^\Gamma \in \Gamma$.
– *Functionality:* If $C(\Diamond s)D \in \Gamma$ and $C(\Diamond s)E \in \Gamma$, then $D = E$.
– *Bad-loop-freeness:* There is no bad loop in Γ.

The first two quasi-branches in Fig. 1 are branches. The third quasi-branch in Fig. 1 is not a branch since it contains a bad loop.

The *core* $\mathcal{C}\Gamma$ of a branch Γ is $\mathcal{C}\Gamma := \{\, C \in \Gamma \mid C^\Gamma = C \,\}$. A branch Γ is *evident* if Γ realizes $\Diamond s$ in C for all $\Diamond s \in C \in \mathcal{C}\Gamma$. The *syntactic closure* $\mathcal{S}\Gamma$ of a branch Γ is the union of the syntactic closures of the clauses $C \in \Gamma$.

Proposition 6.1. *Let Γ be an evident branch and $\Diamond^+ s \in C \in \mathcal{C}\Gamma$. Then there is a unique run for $\Diamond^+ s$ in C in Γ.*

Proof. Since Γ realizes every eventuality in every clause in $\mathcal{C}\Gamma$ and Γ is finite, functional and bad-loop-free, there is a unique run for $\Diamond^+ s$ in C in Γ. □

Theorem 6.1 (Model Existence). *Every evident branch has a finite model.*

Proof. Let Γ be an evident branch. Without loss of generality we can assume that for every nominal $x \in \mathcal{S}\Gamma$ there is a unique clause $C \in \mathcal{C}\Gamma$ such that $x \in C$ (add clauses $\{x\}$ as necessary). We choose an interpretation $\mathcal{I}$ that satisfies the following conditions:

– $|\mathcal{I}| = \mathcal{C}\Gamma$
– $C \rightarrow_{\mathcal{I}} D \iff \exists s, E\colon\ C(\Diamond s)E \in \Gamma$ and $D = E^\Gamma$
– $C \in \mathcal{I}p \iff p \in C$
– $\mathcal{I}x = C \iff x \in C$ for all $x \in \mathcal{S}\Gamma$

We show $\forall s \in \mathcal{S}\Gamma\ \forall C \in \mathcal{C}\Gamma\colon\ C \rhd s \implies \mathcal{I}, C \models s$ by induction on s. Let $s \in \mathcal{S}\Gamma$, $C \in \mathcal{C}\Gamma$, and $C \rhd s$. We show $\mathcal{I}, C \models s$ by case analysis. Except for $s = \Diamond^* t$ the claim follows as in the proofs of Theorems 4.1 and 5.1.

Let $s = \Diamond^* t$. Since $C \rhd s$, we have either $C \rhd t$ or $C \rhd \Diamond^+ t$. If $C \rhd t$, then $\mathcal{I}, C \models t$ by the inductive hypothesis and the claim follows. Otherwise, let $C \rhd \Diamond^+ t$. Then $\Diamond^+ t \in C \in \mathcal{C}\Gamma$. By Proposition 6.1 we know that there is a run for $\Diamond^+ t$ in C in Γ. Thus $C \rightarrow_{\mathcal{I}}^* D$ and $D \rhd t$ for some clause $D \in \mathcal{C}\Gamma$. Hence $\mathcal{I}, D \models t$ by the inductive hypothesis. The claim follows. □

7 Tableaux for H*

The tableau system for H* is obtained with the following rule.

*Expansion Rule for H**
If $\Diamond s \in C \in \mathcal{C}\Gamma$ and Γ does not realize $\Diamond s$ in C,
then expand Γ to all branches $\Gamma\,;D\,;C(\Diamond s)D$ such that $D \in \mathcal{D}^{\Gamma}(\mathcal{R}C; s)$.

Example 7.1. Here is a tableau derivation of an evident branch from an initial clause with eventualities.

$$\begin{tikzcd}
& \boxed{\Diamond^+ p,\ \neg p,\ \Box(x \wedge \Diamond^+ p \wedge \neg p),\ \Diamond\Box\neg p}^{0} \arrow[dl, "1"'] \arrow[dr, "3"] \arrow[dr, dashed, bend left] & \\
\boxed{x,\ \Diamond^+ p,\ \neg p}^{1} \arrow[d, "2"'] & & \boxed{x,\ \Diamond^+ p,\ \neg p,\ \Box\neg p}^{3} \arrow[d, "4"] \\
\boxed{p}^{2} & & \boxed{\Diamond^+ p,\ \neg p}^{4} \arrow[ll, "5"]
\end{tikzcd}$$

The numbers indicate the order in which the links and clauses are introduced. In the final branch, the clauses 0, 3, 4, 2 constitute a run for $\Diamond^+ p$ in clause 0.

The dashed link is not on the branch. We use it to indicate the implicit redirection of link 1 that occurs when clause 3 is added. The implicit redirection is due to nominal propagation and is realized in the definition of paths. Note that before link 3 is added, the clauses 0, 1, 2 constitute a run for $\Diamond^+ p$ in clause 0. When clause 3 is added, this run disappears since clause 1 is no longer in the core.

As in the system for H, the expansion rule for H* has the property that it applies to a branch if and only if the branch is not evident. Moreover, termination follows essentially as for H. There is, however, an essential difference as it comes to soundness. Due to our definition of branches a candidate extension $\Gamma\,;D\,;C(\Diamond s)D$ is only admissible if it is a bad-loop-free quasi-branch. We now encounter the difficulty that the analogue of the soundness property for H (Theorem 5.2) does not hold since there are satisfiable branches to which the expansion rule applies but fails to produce extended branches since all candidate branches contain bad loops. This is shown by the next example.

Example 7.2. Consider the literal $s := \Box^+(q \vee \Box\neg p)$ and the following branch:

$$\begin{tikzcd}
\boxed{\Diamond^+ p,\ \neg p,\ q,\ s} \arrow[d] \\
\boxed{\Diamond^+ p,\ \Box\neg p,\ s} \arrow[d] \\
\boxed{\Diamond^+ p,\ \neg p,\ \Box\neg p,\ s}
\end{tikzcd}$$

Note that the branch is satisfiable. Since the eventuality in the third clause is not realized, the branch is not evident. The expansion rule applies to the eventuality in the third clause but does not produce an extended branch since both candidate extensions contain bad loops (one extension adds a link from the third clause to the first clause, and the other adds a link from the third clause to itself).

Note that the branch can be obtained by starting with the first clause. The link for the literal $\Diamond^+ p$ in the first clause does not make progress and leads to the bad loop situation. There are two alternative links for this literal that point to the clauses $\{p,\ q,\ s\}$ and $\{p,\ \Box\neg p,\ s\}$. Both yield evident branches.

We solve the problem with the notion of a *straight link*. The idea is that a straight link for an eventuality $\Diamond^+ s$ reduces the distance to a clause satisfying s as much as possible. We will define straightness with respect to a model.

Let $\mathcal{I}$ be an interpretation, A be a set of formulas, and s be a formula. The *distance from A to s in $\mathcal{I}$* is defined as follows:

$$\delta_{\mathcal{I}} A s \ := \ \min\{\, n \in \mathbb{N} \mid \exists a, b\colon\ a \rightarrow^n_{\mathcal{I}} b \text{ and } \mathcal{I}, a \models A \text{ and } \mathcal{I}, b \models s \,\}$$

where $\min \emptyset = \infty$ and $n < \infty$ for all $n \in \mathbb{N}$. The relations $\rightarrow^n_{\mathcal{I}}$ are defined as usual: $a \rightarrow^0_{\mathcal{I}} b$ iff $a = b$ and $a \in |\mathcal{I}|$, and $a \rightarrow^{n+1}_{\mathcal{I}} b$ iff $a \rightarrow_{\mathcal{I}} a'$ and $a' \rightarrow^n_{\mathcal{I}} b$ for some a'.

Proposition 7.1. *$\delta_{\mathcal{I}} A s < \infty$ iff $\mathcal{I}$ satisfies $A; \Diamond^* s$.*

Proposition 7.2. *Let $\mathcal{I}$ be a model of a quasi-branch Γ. Then $\delta_{\mathcal{I}} A s = \delta_{\mathcal{I}} A^{\Gamma} s$.*

A link $C(\Diamond^+ s)D$ is *straight* for an interpretation $\mathcal{I}$ if the following conditions are satisfied:

1. $\delta_{\mathcal{I}} D s \leq \delta_{\mathcal{I}} E s$ for every $E \in \mathcal{D}(\mathcal{R}C; \Diamond^* s)$.
2. If $\delta_{\mathcal{I}} D s = 0$, then $D \rhd s$.

A *straight model* of a quasi-branch Γ is a model $\mathcal{I}$ of Γ such that every link $C(\Diamond^+ s)D \in \Gamma$ is straight for $\mathcal{I}$.

Lemma 7.1 (Straightness). *A quasi-branch that has a straight model does not have a bad loop.*

Proof. By contradiction. Let $\mathcal{I}$ be a straight model of a quasi-branch Γ and let $C_1 \ldots C_n$ be a bad loop for $\Diamond^+ s$ in Γ. Then $C_n = C_1$ and $n \geq 2$. To obtain a contradiction, it suffices to show that $\delta_{\mathcal{I}} C_i s > \delta_{\mathcal{I}} C_{i+1} s$ for all $i \in [1, n-1]$. Let $i \in [1, n-1]$.

1. We have $C_i \in \mathcal{C}\Gamma$, $C_i \not\rhd s$, $C_i(\Diamond^+ s)D \in \Gamma$, and $D^{\Gamma} = C_{i+1}$ for some $D \in \Gamma$.
2. We show $\delta_{\mathcal{I}} C_i s < \infty$. By (1) we have $\Diamond^+ s \in C_i \in \Gamma$. The claim follows by Proposition 7.1 since $\mathcal{I}$ satisfies C_i.
3. We show $0 < \delta_{\mathcal{I}} C_i s$. Case analysis.
   (a) $i > 1$. Then $C_{i-1}(\Diamond^+ s)E \in \Gamma$ and $E^{\Gamma} = C_i$ for some E. By Proposition 7.2 and the second condition for straight links it suffices to show $E \not\rhd s$. This holds by Proposition 3.2 since $C_i \not\rhd s$ by (1).
   (b) $i = 1$. Then $C_{n-1}(\Diamond^+ s)E \in \Gamma$ and $E^{\Gamma} = C_1$ for some E. By Proposition 7.2 and the second condition for straight links it suffices to show $E \not\rhd s$. This holds by Proposition 3.2 since $C_1 \not\rhd s$ by (1).

4. By (2) and (3) there are states a, b, c such that $\mathcal{I}, a \models C_i$, $a \to_{\mathcal{I}} b \to_{\mathcal{I}}^{\delta_{\mathcal{I}} C_i s - 1} c$ and $\mathcal{I}, c \models s$. We have $\mathcal{I}, b \models \mathcal{R}C_i \,; \Diamond^* s$. By Proposition 3.3(1) there is a clause $E \in \mathcal{D}(\mathcal{R}C_i \,; \Diamond^* s)$ such that $\mathcal{I}, b \models E$. Thus $\delta_{\mathcal{I}} E s \leq \delta_{\mathcal{I}} C_i s - 1$. By Proposition 7.2 and the first condition for straight links we have $\delta_{\mathcal{I}} C_{i+1} s = \delta_{\mathcal{I}} D s \leq \delta_{\mathcal{I}} E s < \delta_{\mathcal{I}} C_i s$, which yields the claim. □

Theorem 7.1 (Soundness). *Let $\mathcal{I}$ be a straight model of a branch Γ and let $\Diamond s \in C \in \Gamma$. Moreover, let Γ not realize $\Diamond s$ in C. Then there exists a clause $D \in \mathcal{D}^{\Gamma}(\mathcal{R}C \,; s)$ such that $\Gamma \,; D \,; C(\Diamond s) D$ is a branch and $\mathcal{I}$ is a straight model of $\Gamma \,; D \,; C(\Diamond s) D$.*

Proof. Since $\mathcal{I}$ is a model of C and $\Diamond s \in C$, there is a clause $D \in \mathcal{D}^{\Gamma}(\mathcal{R}C; s)$ that is satisfied by $\mathcal{I}$ (Propositions 3.5 and 5.2). For every such clause, $\Gamma \,; D \,; C(\Diamond s) D$ is a quasi-branch that has $\mathcal{I}$ as a model and satisfies the nominal coherence and functionality conditions. By Lemma 7.1 it suffices to show that we can choose D such that $\mathcal{I}$ is straight for $C(\Diamond s) D$. If $\Diamond s$ is not an eventuality, this is trivially the case. Otherwise, let $\Diamond s = \Diamond^+ t$ and $s = \Diamond^* t$. We proceed by case analysis.

1. $\mathcal{I}$ satisfies $\mathcal{R}C; t$. We pick a clause $D \in \mathcal{D}^{\Gamma}(\mathcal{R}C; t)$ that is satisfied by $\mathcal{I}$. By Proposition 3.3(4), we have $\mathcal{D}(\mathcal{R}C; t) \subseteq \mathcal{D}(\mathcal{R}C; \Diamond^* t)$, and consequently $\mathcal{D}^{\Gamma}(\mathcal{R}C; t) \subseteq \mathcal{D}^{\Gamma}(\mathcal{R}C; \Diamond^* t)$. Thus $D \in \mathcal{D}^{\Gamma}(\mathcal{R}C; \Diamond^* t)$ as required. It remains to show that $\mathcal{I}$ is straight for $C(\Diamond^+ t) D$. This is the case since $\delta_{\mathcal{I}} D t = 0$ since $D \rhd t$ and $\mathcal{I}$ satisfies D (Proposition 3.1).
2. $\mathcal{I}$ does not satisfy $\mathcal{R}C; t$. This time we choose $D \in \mathcal{D}^{\Gamma}(\mathcal{R}C; \Diamond^* t)$ such that $\mathcal{I}$ satisfies D and $\delta_{\mathcal{I}} D t$ is minimal. We show that $\mathcal{I}$ is straight for $C(\Diamond^+ t) D$.

   Let $E \in \mathcal{D}(\mathcal{R}C; \Diamond^* t)$. We show $\delta_{\mathcal{I}} D t \leq \delta_{\mathcal{I}} E t$. If $\mathcal{I}$ does not satisfy E, the claim holds by Proposition 7.1. If $\mathcal{I}$ satisfies E, $\mathcal{I}$ satisfies E^{Γ} and $E^{\Gamma} \in \mathcal{D}^{\Gamma}(\mathcal{R}C; \Diamond^* t)$. Hence $\delta_{\mathcal{I}} D t \leq \delta_{\mathcal{I}} E^{\Gamma} t$. The claim follows by Proposition 7.2.

   We show $\delta_{\mathcal{I}} D t > 0$. For contradiction, let $\delta_{\mathcal{I}} D t = 0$. Then $\mathcal{I}, a \models D \,; t$ for some a. Thus $\mathcal{I}, a \models \mathcal{R}C \,; t$ by Proposition 3.3(1). Contradiction. □

We have now arrived at a decision procedure for the sublogic H^*.

8 Tableaux for H* with @

It is straightforward to extend our results to the full logic $\mathrm{H}^*_@$, which adds formulas of the form $@_x s$ to H^*. A quasi-branch Γ *realizes a literal* $@_x s$ if it contains a clause D such that $D \rhd \{x, s\}$. A branch Γ is *evident* if it is evident as defined for H^* and in addition realizes every literal $@_x s$ such that $@_x s \in C$ for some clause $C \in \Gamma$. It is easy to verify that the model existence theorem for H^* extends to the full logic $\mathrm{H}^*_@$. The realization condition for @ leads to an additional expansion rule.

Expansion Rule for @
If $@_x s \in C \in \mathcal{C}\Gamma$ and Γ does not realize $@_x s$,
then expand Γ to all branches $\Gamma \,; D$ such that $D \in \mathcal{D}^{\Gamma}\{x, s\}$.

Since the new rule respects the subterm closure, termination is preserved. The soundness of the new rule is easy to show.

Proposition 8.1 (Soundness of Rule for @). *Let $\mathcal{I}$ be a straight model of a branch Γ and let $@_x s \in C \in \Gamma$. Then there exists a clause $D \in \mathcal{D}^{\Gamma}\{x, s\}$ such that $\Gamma ; D$ is a branch and $\mathcal{I}$ is a straight model of $\Gamma ; D$.*

9 Optimizations

We give two additional rules that realize certain diamond literals with links to already present clauses, thus avoiding the introduction of unnecessary clauses and unnecessary branchings. This way the size of the tableaux the decision procedure has to explore can be reduced.

Additional Expansion Rule for Simple Diamonds
If $\Diamond s \in C \in \mathcal{C}\Gamma$ and Γ does not realize $\Diamond s$ in C
and $\Diamond s$ is simple and $D \in \Gamma$ and $D \triangleright \mathcal{R}C ; s$,
then expand Γ to $\Gamma ; C(\Diamond s)D$.

Additional Expansion Rule for Eventualities
If $\Diamond^+ s \in C \in \mathcal{C}\Gamma$ and Γ does not realize $\Diamond^+ s$ in C
and $D \in \Gamma$ and $D \triangleright \mathcal{R}C ; s$,
then expand Γ to $\Gamma ; C(\Diamond^+ s)D$.

Both rules preserve straight models and yield extensions that are branches. This suffices for their correctness.

A branch Γ is *quasi-evident* if there is some set Δ of links such that $\Gamma \cup \Delta$ is an evident branch. It suffices if the decision procedure stops with quasi-evident branches rather than evident branches. This provides for optimizations since it allows the decision procedure not to introduce new clauses and branchings for diamond formulas that can be realized with links to existing clauses. The two expansion rules given above are subsumed by this optimization.

Let Γ be a branch. A clause C is *subsumed in Γ* if C contains no eventualities and there is a clause $D \in \Gamma$ such that $C \subsetneq D$. There is no need to realize literals in subsumed clauses.

Proposition 9.1. *A branch that realizes all literals of the form $\Diamond s$ or $@_x s$ in non-subsumed clauses is quasi-evident and hence has a finite model.*

The left branch in Fig. 1 explains why subsumed clauses must not contain eventualities.

Finally, we remark that bad-loop checking can be done in quasi-constant time when a branch is extended with a new link. For this, one maintains a data structure that for every clause C and every unrealized eventuality $\Diamond^+ s \in C$ provides all clauses D such that a link $C(\Diamond^+ s)D$ would result in a bad loop. Such a data structure can be maintained in quasi-constant time. History variables as used in De Giacomo and Massacci [7] are one possibility to realize such a data structure.

References

1. Abate, P., Goré, R., Widmann, F.: An on-the-fly tableau-based decision procedure for PDL-satisfiability. In: Areces, C., Demri, S. (eds.) M4M-5. ENTCS, vol. 231, pp. 191–209. Elsevier, Amsterdam (2009)
2. Areces, C., ten Cate, B.: Hybrid logics. In: Blackburn, P., van Benthem, J., Wolter, F. (eds.) Handbook of Modal Logic, pp. 821–868. Elsevier, Amsterdam (2007)
3. Baader, F.: Augmenting concept languages by transitive closure of roles: An alternative to terminological cycles. DFKI Research Report RR-90-13, Deutsches Forschungszentrum für Künstliche Intelligenz, Kaiserslautern, Germany (1990)
4. Blackburn, P., de Rijke, M., Venema, Y.: Modal Logic. Cambridge University Press, Cambridge (2001)
5. Bolander, T., Blackburn, P.: Termination for hybrid tableaus. J. Log. Comput. 17(3), 517–554 (2007)
6. Bolander, T., Braüner, T.: Tableau-based decision procedures for hybrid logic. J. Log. Comput. 16(6), 737–763 (2006)
7. De Giacomo, G., Massacci, F.: Combining deduction and model checking into tableaux and algorithms for converse-PDL. Inf. Comput. 162(1-2), 117–137 (2000)
8. Emerson, E.A., Clarke, E.M.: Using branching time temporal logic to synthesize synchronization skeletons. Sci. Comput. Programming 2(3), 241–266 (1982)
9. Emerson, E.A., Halpern, J.Y.: "Sometimes" and "not never" revisited: On branching versus linear time temporal logic. J. ACM 33(1), 151–178 (1986)
10. Fischer, M.J., Ladner, R.E.: Propositional dynamic logic of regular programs. J. Comput. System Sci., 194–211 (1979)
11. Goré, R., Widmann, F.: An optimal on-the-fly tableau-based decision procedure for PDL-satisfiability. In: Schmidt, R.A. (ed.) CADE 2009. LNCS, vol. 5663, pp. 437–452. Springer, Heidelberg (2009)
12. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. The MIT Press, Cambridge (2000)
13. Horrocks, I., Sattler, U.: A tableau decision procedure for $\mathcal{SHOIQ}$. J. Autom. Reasoning 39(3), 249–276 (2007)
14. Kaminski, M., Smolka, G.: Terminating tableaux for hybrid logic with the difference modality and converse. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 210–225. Springer, Heidelberg (2008)
15. Kaminski, M., Smolka, G.: Terminating tableau systems for hybrid logic with difference and converse. J. Log. Lang. Inf. 18(4), 437–464 (2009)
16. Pnueli, A.: The temporal logic of programs. In: FOCS'77, pp. 46–57. IEEE Computer Society Press, Los Alamitos (1977)
17. Pratt, V.R.: A near-optimal method for reasoning about action. J. Comput. System Sci. 20(2), 231–254 (1980)
18. Sattler, U., Vardi, M.Y.: The hybrid μ-calculus. In: Goré, R., Leitsch, A., Nipkow, T. (eds.) IJCAR 2001. LNCS (LNAI), vol. 2083, pp. 76–91. Springer, Heidelberg (2001)
19. Schneider, S.: Terminating Tableaux for Modal Logic with Transitive Closure. Bachelor's thesis, Saarland University (2009)

Herod and Pilate: Two Tableau Provers for Basic Hybrid Logic

Marta Cialdea Mayer[1] and Serenella Cerrito[2]

[1] Università di Roma Tre
[2] Ibisc, Université d'Evry Val d'Essonne

Abstract. This work presents two provers for basic hybrid logic $HL(@)$, which have been implemented with the aim of comparing the internalised tableau calculi independently proposed, respectively, by Bolander and Blackburn [3] and Cerrito and Cialdea Mayer [5]. Experimental results are reported, evaluating, from the practical point of view, the different treatment of nominal equalities of the two calculi.

1 A Brief Presentation of the Calculi P and H

The treatment of nominal equalities in proof systems for hybrid logics may easily induce many redundancies. In fact, when processing a statement of the form $@_a b$, any known property of a can potentially be copied to b (and vice-versa). This work compares, from a practical point of view, two different approaches to nominal equalities, represented by two internalised tableau calculi for $HL(@)$, both terminating without loop-checking and with no restriction on rule application strategies. The two calculi were independently proposed, respectively, in [3] and [5] (a revised and extended version of the latter is [4]). The main difference between them is the treatment of nominal equalities, that is essentially carried out by means of an elegant and simple rule in the first calculus ([3], here named P), which copies formulae labelled by a to the equal nominal b (the Id rule), while the second ([4,5], the calculus H) uses a more technically involved rule, requiring explicit substitution (the Sub rule).

An analysis and comparison of P and H, highlighting their respective similarities and differences from a theoretical point of view can be found in [7], where also an application-oriented, though abstract, reformulation of the expansion rules can be found. This presentation describes the implementations and some experimental results, run on sets of formulae randomly generated by hGen [1]. The experiments show that the approach to nominal equalities in H, although theoretically less elegant than P's, has important practical advantages. The systems are also briefly compared to other implemented tableau provers. We omit here the definition of the syntax and semantics of $HL(@)$, which can be found in any work on hybrid logic.

Since the provers only deal with the uni-modal version of $HL(@)$, the (necessarily brief and informal) description of the tableau systems that follows is restricted accordingly. Differently from [4,5,7], for the sake of simplicity, tableaux are presented here in the "nodes as formulae" style. Assuming that input formulae are in negation normal form, tableau nodes in both P and H are labelled by *satisfaction statements*, i.e. formulae of the form $@_aF$, where F is in negation normal form. The initial tableau for a set S of formulae is a single branch tableau whose nodes are labelled by $@_aF$, for each $F \in S$, where a is a new nominal. The set $\{@_aF \mid F \in S\}$ is called the *initial set.*

The two systems share the following expansion rules:

$$\frac{@_a(F \wedge G)}{@_aF, @_aG}\ (\wedge) \qquad \frac{@_a(F \vee G)}{@_aF \quad \mid \quad @_aG}\ (\vee) \qquad \frac{@_a@_bF}{@_bF}\ (@)$$

$$\frac{@_a\Box F, @_a\Diamond b}{@_bF}\ (\Box) \qquad \frac{@_a\Diamond F}{@_a\Diamond b, @_bF}\ (\Diamond) \text{ where } b \text{ is new in the branch}$$

The $\Diamond$-rule is subject to some restrictions that will be given later on.

It is assumed that a formula is never added to a branch where it already occurs and that the $\Diamond$-rule, which generates new nominals, is never applied twice to the same premise on the same branch. A tableau branch is closed if it contains either $@_ap$ and $@_a\neg p$ for some nominal a and atom p, or $@_a\neg a$ for some nominal a.

A formula of the form $@_a\Diamond b$, where b is a nominal, is a *relational formula.* A relational formula generated by application of the $\Diamond$-rule is called an *accessibility formula.* The system P restricts the applicability of the $\Diamond$-rule to cases where its premise $@_a\Diamond F$ is not an accessibility formula, while in H it is restricted to cases where $@_a\Diamond F$ is not a relational formula.

If a nominal b is introduced in a branch Θ by application of the $\Diamond$-rule to a premise of the form $@_a\Diamond F$, then we say that b is a child of a, and use the notation $a \prec_\Theta b$. The relation $\prec_\Theta{}^+$ is the transitive closure of $\prec_\Theta$. If $a \prec_\Theta^+ b$ we say that b is a descendant of a and a an ancestor of b in the branch Θ.

The essential difference between the two calculi consists in the treatment of nominal equalities. In P, such formulae are expanded by means of the two premises rule Id, which is applicable only if $@_aF$ is not an accessibility formula:

$$\frac{@_aF, @_ab}{@_bF}\ (Id)$$

The system H treats equalities by means of a more complex rule, with side effects, the Substitution rule (Sub). When expanding a formula of the form $@_ab$ (where $a \neq b$) by means of Sub in a branch Θ, the whole branch is modified as follows: every occurrence of a is replaced by b, and every formula containing a descendant of a is deleted.

The two rules for the treatment of equalities are apparently very different. However, they bear strong similarities, which are highlighted in [7]. We only observe here that the restriction on the Id rule, and nominal deletion in the Sub rule, are crucial to ensure strong termination of the respective systems. And their role is similar: avoiding the "adoption" of a replaced nominal's children by the replacing one. In simple words, we can say that the main difference between the two systems is that P is more tolerant than H: even when the descendants of a nominal are of no use any longer, they are left alive, since they are not harmful, either. H, on the contrary, is radical and bloody: when a nominal becomes useless, it is killed with all its descent.

Nominal deletion in H has a practical advantage, avoiding the employment of resources to expand formulae labelled by "useless" nominals. It can therefore be conjectured that the treatment of nominal equalities in H might be more efficient than in P. In order to verify such an hypothesis, the calculi P and H have been implemented, as described in the next section.

2 Pilate and Herod

The system Pilate ("What crime has he committed?") implements the calculus P, and Herod is the implementation of the slaughter of the innocents represented by H. The two systems are implemented in Objective Caml [12] and are available at Herod web page [13]. In the rest of this section a description of the implementations is given, with some simplifications and abstractions.

Both systems take as input a file specifying a set of hybrid formulae and build a tableau for them in a depth-first manner. If a complete and open branch is found, then a model of the initial set of formulae is extracted from it and given as output. Otherwise, the set is declared unsatisfiable. At any stage of the search process, the systems consider a single *active branch*, the others being kept in a stack, where the branching points are stored and retrieved upon backtracking.

A branch Θ is represented by a set (implemented by a hash table) of *worlds*. Each world w corresponds to a nominal occurring in the branch and is a structure (a record) with the following fields: *name(w)* stores the nominal a corresponding to w; *pending(w)* contains the formulae labelled by a (a=*name(w)*) in Θ, that still have to be processed; *memory(w)* contains the formulae labelled by a in Θ that have already been processed but have to be kept in memory for future use (i.e. literals and formulae of the form $\Box F$ and $\Diamond F$); *children(w)* is a set of pointers to the children of a in Θ, i.e. the nominals b such that $a \prec_\Theta b$.

In the implementation of Herod, beyond the set of its worlds, each branch is associated a *table of replacements*, that is updated with the application of the substitution rule, and used whenever accessing a nominal, looking for its presently replacing nominal. Substitution is in fact treated in a lazy way. In the following, we shall denote the nominal replacing a by *replaces(a)*, meaning that *replaces(a)* $= a$ if a has not been replaced in the branch.

The expansion loop obeys the following basic principle: for each world w, *memory(w)* is "saturated" with respect to any rule application, in the sense that every formula (or pair of formulae) in *memory(w)* has been expanded. At each stage, a world w is chosen for expansion and a formula F selected from *pending(w)*. According to the form of F, different operations are performed.

– If F is a literal ℓ, then its consistency is checked with respect to the literals in *memory(w)*. If the branch is closed, then the systems backtrack; otherwise, if ℓ is not a nominal, it is moved to *memory(w)* and if it is a nominal, the respective rules for equalities, described below, are fired.

– If F is either a disjunction, or a conjunction or a satisfaction statement, its expansion (or, in the case of a disjunction, one of its expansions, the other being recorded in the stack) is added to *pending(w)* and F is deleted.

The treatment of the other rules differs in the two systems. We begin by describing Herod.

H1. The expansion of a formula of the form $\Diamond F$ in a world w causes the moving of $\Diamond F$ from *pending(w)* to *memory(w)*, the creation of a new world w' with $F \in$*pending(w′)*, and the addition of w' to *children(w)*.

H2. The $\Box$-rule in Herod is fired whenever the selected formula has the form of one of its premises: if it has the form $\Box F$, then the $\Box$-rule is applied to it and each formula of the form $\Diamond b \in$*memory(w)*. Moreover, it is applied to each $\Diamond b$ implicitly represented by *children(w)*. Symmetrically, if the extracted formula has the form $\Diamond b$, then the $\Box$-rule is applied to it and each formula of the form $\Box F \in$*memory(w)*. The obtained expansions are in both cases added to *pending(w)*.

H3. Finally, if the selected formula is a nominal b, the substitution rule is fired. If w' is the world representing b, then: (a) every formula in *pending(w)* is copied to *pending(w′)*; (b) every formula of the form $\Diamond F \in$*memory(w)* is copied to *pending(w′)*; (c) every formula of the form $\Box F \in$*memory(w)* is copied to *memory(w′)* and the $\Box$-rule is fired against each $\Diamond c \in$*memory(w′)* and the children of w'; (d) the literals in *memory(w)* are added to *memory(w′)*, after a consistency check. Finally, the table of replacements is updated, and the world w and, recursively, all its descendants are deleted. This suffices to implement nominal deletion; in fact, every information related to a descendant w'' of w is contained either in w'' itself, or in its parent, which is either w or, in turn, a descendant of w (that is deleted altogether).

Let us now turn to consider the implementation of the Id rule in Pilate. The basic principle is always that the set in *memory(w)* has to be maintained saturated with respect to every rule application. Therefore:

P1. When an equality $@_a b$ is processed, i.e. a nominal b is chosen from *pending(w)* for some world w with *name(w)=a*, it is moved to *memory(w)* and the Id rule is fired against $@_a b$ and: (a) any already processed formula of the form $@_a F$ that is not an accessibility formula, yielding $@_b F$ (i.e. F is added to to the *pending* field of the world w' representing b – provided that F is not already in *memory(w′)*); (b) any already processed formula of the form $@_a c$, producing $@_c b$ (i.e. for all nominal c in *memory(w)*, b is added to the *pending* field of the world w' representing c – provided that b is not already in *memory(w′)*); and, finally (c) Id is applied to $@_a b$ and $@_a b$ itself, if $a \neq b$ (producing $@_b b$, i.e. b is added to the *pending* field of the world representing b); such an operation is in fact necessary for completeness. The membership tests are needed in order to avoid that, in the presence of a looping chain of equalities, formulae are copied forever, passing from the memory of a node to the pending formulae of another one and then back to the pending formulae of the first one.

The symmetric case is subtler, since the leftmost premise of the Id rule can have any form. Pilate's treatment consists in firing the Id rule against any memorised equality every time a formula is moved to *memory(w)*:

P2. when processing a literal or a formula of the form $\Box F$ or $\Diamond F$ chosen from a world w with *name(w)=a*, beyond the operations performed by Herod (H1 and H2), the Id rule is fired against such a formula and any already processed equality of the form $@_a b$ (i.e. any nominal in *memory(w)*, and the results are added to the *pending* field of the world representing b).

3 Experimental Results

The relative performances of Herod and Pilate have been evaluated running the provers on a set of 1600 sets of formulae, randomly generated by hGen [1], and approximately equally partitioned into satisfiable and unsatisfiable. The modal depth (greatest number of nested modal operators) of the tests varies from 10 to 40, and the number of clauses varies from 60 to 200. The sets of formulae used for the benchmarks and the parameters used for their generation can be found at Herod web page [13]. The experiments were run on an Intel Pentium 4 3GHz, with 3Gb RAM, running under Linux, and the provers were given 1 minute timeout.

Considering the 1274 tests that both Pilate and Herod solved in the allowed time, Pilate is in the average more than 50 times slower than Herod, and the median run time of Pilate is more than 5 times Herod's one.[1] Moreover, Pilate runs out of time almost 50% more often than Herod. The diagram on the left in Figure 1 plots the average run time of the two systems against the number of clauses of the tests solved by both provers. In the the diagram, points on the X-axis group all sets with the same number of clauses, independently of their modal depth. Pilate seems to be more sensible to the phase transition in the easy-hard-easy pattern of the benchmarks [8].

Maybe more interesting is the comparison between the two systems on a set of hand-written formulae which involve many $\Diamond$'s and equalities, where the differences in treating equalities should be pushed to the limit. The formulae we have used have the form $@_{a_1}\Diamond^n(@_{a_1}a_2 \wedge ... \wedge @_{a_n}a_{n+1} \wedge \Diamond F)$, where $\Diamond^n$ is a sequence of n $\Diamond$'s, dominating n nominal equalities, and F is a (non trivially) unsatisfiable formula. Processing such formulae forces the provers to generate n new worlds before processing the equalities. The size of such formulae is taken to be n. The results are represented by the diagram on the right in Figure 1. As can be seen, Pilate can only solve problems up to size 100 in the allowed time of one minute, while Herod solved the problems up to the maximal tested size (600), within 0.11 seconds.

The empirical results described above confirm that Pilate consumes, in general, more resources than Herod. It is important to point out, moreover, that the

[1] The median times are computed counting timeouts as values greater than all the others.

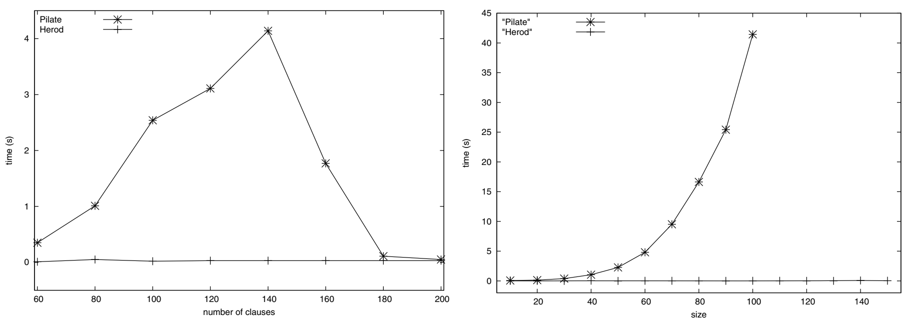


Fig. 1. Pilate and Herod average run times

different performances are effectively due to the different treatment of equalities. In fact, on a set of 400 modal formulae (without nominals and satisfaction operators) randomly generated by hGen, of modal depth varying from 30 to 70, the two provers had the same cases of timeouts and the same average and median execution times. The same results, showing that that all the difference between the systems is due to their treatment of equalities, are obtained when running the two provers on the hand-tailored collection of modal formulae proposed in [2], where in fact Pilate and Herod show the same behaviour.

It must be remarked, however, that Pilate constitutes a more or less "ad literam" implementation of P and that more effective ways of treating its Id rule can be conceived. For this reason, Herod has been compared with HTab [11], which implements the prefixed calculus presented in [3]. HTab implements equality by means of equivalence classes of nominals and prefixes, that are created, enlarged and merged while the tableau construction proceeds. Many redundancies are avoided by processing only formulae true at the representative of each class. However, in HTab, *the descendants of any nominal are always expanded,* thus consuming time and space. The comparison has also considered another prover, Spartacus [10].[2] Spartacus processes nominal equalities by merging the content of the corresponding "nodes", and electing one of them as the representative of both [9]. Both HTab and Spartacus are much more mature provers than Herod, handling a richer logic and implementing many important optimisation strategies. On the contrary, at present, the only simple rule application strategy adopted by Herod, beyond semantic branching, consists of delaying tableau branching as far as possible. And in fact, HTab and Spartacus behave definitely much better than Herod when run on the set of hand-tailored collection of modal formulae presented in [2], as well as on hybrid formulae of a very low modal depth (personal communication by the maintainer of HTab). The experiments reported below aimed at testing whether Herod gains any advantage from its treatment of equalities.

[2] In the tests, HTab 1.4.0 and Spartacus 1.0.1 were used, both run with the respective default options.

The three provers were run on the same sets of random formulae used for the comparison with Pilate. As can be seen in the tables below, HTab could solve 95% of the tests in the allowed time (one minute) and space (7% more than Herod), and Spartacus failed to solve only one test in the allowed one minute time. Although the number of Herod's failures is the highest one, surprisingly enough its median run time is much better than HTab's and Spartacus's. Moreover, the average execution times on all the problems (modal depth 10 to 40) solved by both Herod and HTab are in favour of Herod. In the average, moreover, the execution times of Herod and Spartacus are comparable.

modal	Number of failures			Median times		
depth	Herod	HTab	Spartacus	Herod	HTab	Spartacus
10	56	22	1	0.02	0.07	0.07
20	47	12	0	0.02	0.14	0.12
30	44	24	0	0.03	0.20	0.16
50	42	24	0	0.04	0.29	0.21
total	189	82	1	0.03	0.16	0.13

Average times						
	Herod vs HTab			Herod vs Spartacus		
modal	number of tests	Average time		number of tests	Average time	
depth	solved by both	Herod	HTab	solved by both	Herod	Spartacus
10	340/400	0.16	0.12	344/400	0.16	0.06
20	348/400	0.03	0.21	353/400	0.03	0.11
30	347/400	0.03	0.27	356/400	0.04	0.16
50	346/400	0.25	0.31	358/400	0.24	0.20
total	1381/1600	0.12	0.23	1411/1600	0.12	0.13

The efficiency of Herod's treatment of equalities is apparent when comparing the three systems on the set of hand-written formulae earlier defined. The interest of such tests relies on the fact that they are meant not so much to compare the provers in themselves, but rather their different treatments of nominal equalities. The provers were run on formulae up to size 600. Herod employed 10 seconds to solve the problem of maximal size, while the execution times of Spartacus and HTab were, respectively, of 24 and 28 seconds.

4 Concluding Remarks

The experimental results are encouraging and suggest that, although an important refinement and optimisation work still has to be done, Herod's approach to nominal equalities is algorithmically interesting. In fact, although Herod is still at a very early stage of development, the very nature of the treatment of nominal equalities in H already gives rather good results, thanks to nominal deletion that allows one to economize on the number of processed nominals. Therefore it seems worth going on and refine its implementation, both by some routine work that still can be done, as well as by studying and experimenting more effective rule application strategies and the implementation of basic optimisation techniques. Moreover, Herod has to be extended to handle different accessibility relations and the global and converse modalities [6].

Acknowledgements. The very first versions of Herod and Pilate were developed as a bachelor project at "Università di Roma Tre" by Emanuele Benassi, Fabio Giammarinaro and Chiara Varani.

References

1. Areces, C., Heguiabehere, J.: hGen: A random CNF formula generator for hybrid languages. In: Methods for Modalities 3 (M4M-3), Nancy, France (2003)
2. Balsiger, P., Heuerding, A., Schwendimann, S.: A benchmark method for the propositional modal logics K, KT, S4. Journal of Automated Reasoning 24(3), 297–317 (2000)
3. Bolander, T., Blackburn, P.: Termination for hybrid tableaus. Journal of Logic and Computation 17(3), 517–554 (2007)
4. Cerrito, S., Cialdea Mayer, M.: An efficient approach to nominal equalities in hybrid logic tableaux. Journal of Applied Non-classical Logics (to appear)
5. Cerrito, S., Cialdea Mayer, M.: Terminating tableaux for HL(@) without loop-checking. Technical Report IBISC-RR-2007-07, Ibisc Lab., Université d'Evry Val d'Essonne (2007), http://www.ibisc.univ-evry.fr/Vie/TR/2007/IBISC-RR2007-07.pdf
6. Cerrito, S., Cialdea Mayer, M.: Tableaux with substitution for hybrid logic with the global and converse modalities. Technical Report RT-DIA-155-2009, Dipartimento di Informatica e Automazione, Università di Roma Tre (2009)
7. Cialdea Mayer, M., Cerrito, S., Benassi, E., Giammarinaro, F., Varani, C.: Two tableau provers for basic hybrid logic. Technical Report RT-DIA-145-2009, Dipartimento di Informatica e Automazione, Università di Roma Tre (2009)
8. Gent, I.P., Walsh, T.: The SAT phase transition. In: Proceedings of the Eleventh European Conference on Artificial Intelligence, ECAI'94, pp. 105–109 (1994)
9. Götzmann, D.: Spartacus: A tableau prover for hybrid logic. Master's thesis, Saarland University (2009)
10. Götzmann, D., Kaminski, M., Smolka, G.: Spartacus: A tableau prover for hybrid logic. In: M4M6. Computer Science Research Reports, vol. 128, pp. 201–212. Roskilde University (2009)
11. Hoffmann, G., Areces, C.: HTab: A terminating tableaux system for hybrid logic. Electronic Notes in Theoretical Computer Science, vol. 231, pp. 3–19 (2007); Proceedings of the 5th Workshop on Methods for Modalities, M4M-5 (2007)
12. Leroy, X.: The Objective Caml system, release 3.11. Documentation and user's manual (2008), http://caml.inria.fr/
13. Herod web page (2009), http://cialdea.dia.uniroma3.it/herod/

Automated Synthesis of Induction Axioms for Programs with Second-Order Recursion

Markus Aderhold

Technische Universität Darmstadt, Germany
aderhold@informatik.tu-darmstadt.de

Abstract. In order to support the verification of programs, verification tools such as ACL2 or Isabelle try to extract suitable induction axioms from the definitions of terminating, recursively defined procedures. However, these extraction techniques have difficulties with procedures that are defined by second-order recursion: There a first-order procedure f passes itself as an argument to a second-order procedure like *map*, *every*, *foldl*, etc., which leads to *indirect* recursive calls. For instance, second-order recursion is commonly used in algorithms on data structures such as *terms* (variadic trees). We present a method to automatically extract induction axioms from such procedures. Furthermore, we describe how the induction axioms can be optimized (i. e., generalized and simplified). An implementation of our methods demonstrates that the approach facilitates straightforward inductive proofs in a verification tool.

1 Introduction

For the verification of programs one usually needs to show that a program behaves as expected for *all* possible inputs. Therefore formal specifications of expected properties often contain universal quantifications. In order to prove a universal formula $\forall x : \tau.\ \psi[x]$, many theorem provers employ *explicit induction* [4,5,7,10,11,16]. Given a well-founded relation $\succ$ on the domain τ that the quantification ranges over (i. e., a relation without infinite chains $q_0 \succ q_1 \succ q_2 \succ \ldots$), the general schema of *well-founded induction* permits the inference

$$\frac{\forall x : \tau.\ \big(\forall x' : \tau.\ x \succ x' \rightarrow \psi[x']\big) \rightarrow \psi[x]}{\forall x : \tau.\ \psi[x]}\ . \qquad (1)$$

For a concrete well-founded relation $\succ$, we call (1) an *induction axiom*.[1]

From the infinitely many well-founded relations $\succ$ that exist for each nontrivial data type τ, in general only few relations are suitable to prove $\forall x : \tau.\ \psi[x]$ for a given formula ψ. Thus finding an appropriate well-founded relation $\succ$ for a formula ψ is an essential challenge in program verification.

One particularly successful approach to finding a suitable induction axiom for a formula ψ is *recursion analysis*, which was pioneered by Boyer and Moore [5].

[1] The term "axiom" emphasizes that well-foundedness of $\succ$ need not necessarily be proved within the formal system, but may be assumed when applying (1).

Variants have been developed that are used in current theorem provers, see [4,9,10,13] for instance. The idea is to exploit the strong relationship between recursion and induction by uniformly extracting well-founded relations from terminating, recursively defined procedures occurring in formula ψ.

In this paper we describe a method for recursion analysis of procedures with *second-order recursion*. A procedure f is defined by second-order recursion if f calls a second-order[2] procedure g using f in a function argument for g, e. g., $g(f, \ldots)$ [8,12]. Typical examples of second-order recursion arise in algorithms on variadic trees such as terms; e. g., applying a substitution to a term, counting the variables in a term, computing the size of a term (cf. Figs. 1 and 2). The following examples illustrate how recursion analysis works and why second-order recursion is a challenge for current theorem provers.

Example 1. Fig. 1(a) shows an example program that defines data types *bool*, $\mathbb{N}$, and $list[@A]$ (where $@A$ is a type variable) by enumerating the respective data constructors *true*, *false*, $\mathtt{0}$, *succ*, $\varnothing$, and "::". Each argument position of a data constructor is assigned a *selector* function; e. g., selector *pred* denotes the predecessor function. Procedure *sum* computes the sum of all numbers in a list k. An induction axiom for proofs about *sum* can be directly read off from the recursive definition:

$$\frac{\forall k : list[\mathbb{N}].\ k = \varnothing \rightarrow \psi[k] \qquad \forall k : list[\mathbb{N}].\ k \neq \varnothing \wedge \psi[tl(k)] \rightarrow \psi[k]}{\forall k : list[\mathbb{N}].\ \psi[k]} \tag{2}$$

The base case of the recursion becomes a base case of the induction. The recursive call $sum(tl(k))$ gives rise to the induction hypothesis $\psi[tl(k)]$ in the step case. $\diamondsuit$

Example 2. In Fig. 1(b), procedure *map* is a second-order procedure that gets a first-order function f as argument. Procedure *varcount* uses second-order recursion to count the number of variables in a term t, modeled by data type *term*. (Expressions of the form $?cons(t)$ check if t denotes a value of the form $cons(\ldots)$.) While it is easy to see that $?var(t)$ is a base case of the recursion, the arguments of the recursive calls of *varcount* are not obvious from the source code. However, this information about the indirect recursion via *map* is necessary to synthesize an induction axiom for *varcount*. $\diamondsuit$

Isabelle builds on the concept of so-called *congruence rules* that tell the system which function calls need to be evaluated [12,8]. For example, a procedure call $map(f, k)$ requires evaluation of $f(z)$ for all $z \in k$. From this knowledge one can infer that *varcount* is recursively called on all terms $z \in args(t)$. A drawback of congruence rules is that the *user* needs to state and prove the corresponding congruence theorems. Moreover, for a fixed set of congruence rules—possibly supplied by libraries—the resulting induction axioms may easily become suboptimal (e. g., due to weak induction hypotheses) [8].

[2] As in [3], we define the order $o(\tau)$ of base types τ like $\mathbb{N}$ or $list[\mathbb{N}]$ as 0; the order of a functional type $\tau_1 \times \ldots \times \tau_n \rightarrow \tau$ is $1 + \max_i o(\tau_i)$ for a base type τ.

```
(a) structure bool <= true, false
    structure ℕ <= 0, succ(pred : ℕ)
    structure list[@A] <= ø, ::(hd : @A, tl : list[@A])
    procedure sum(k : list[ℕ]) : ℕ <=
    if k = ø then 0 else hd(k) + sum(tl(k))

(b) structure variable.symbol <= variable(varID : ℕ)
    structure function.symbol <= func(funcID : ℕ)
    structure term <=
      var(vsym : variable.symbol),
      apply(fsym : function.symbol, args : list[term])
    procedure map(f : @A → @B, k : list[@A]) : list[@B] <=
    if k = ø then ø else f(hd(k)) :: map(f, tl(k))
    procedure varcount(t : term) : ℕ <=
    if ?var(t) then 1 else sum(map(varcount, args(t)))
```

Fig. 1. A functional program with (a) the first-order procedure *sum* and (b) the second-order procedure *map* and second-order recursion in procedure *varcount*

The contributions of this paper

(1) allow the automated extraction of induction axioms from procedures that are defined by second-order recursion (e. g., procedure *varcount*) and
(2) facilitate the optimization (i. e., generalization and simplification) of induction axioms, which permits more straightforward inductive proofs.

The optimization also helps to reveal the essence of the recursion structure of a procedure. This supports the heuristic selection of an induction axiom for a formula ψ (as ψ usually involves more than just one procedure). However, such a heuristic selection is beyond the scope of this paper.

The input for our methods is the source code of the procedures and their termination proofs. In particular, our approach does not require additional user input such as congruence theorems. It has been implemented and integrated into √eriFun, a semi-automated verifier for functional programs [16].

In Sect. 2 we give a brief overview over the programming language and some terminology that we use afterwards. Sect. 3 describes the synthesis of so-called *quantification procedures* that we use to formulate induction hypotheses. The synthesis of induction axioms is presented in Sect. 4. We describe techniques for their optimization in Sect. 5 and compare our methods with related techniques in Sect. 6. We conclude with experimental results in Sect. 7.

2 Programming Language and Terminology

We briefly summarize the relevant features of √eriFun's input language $\mathcal{L}$ [1,15] that roughly corresponds to the second-order fragment of ML or Haskell with strict evaluation; additional details can be found in [1,15].

$\mathcal{L}$ offers definition principles for freely generated polymorphic data types, for first-order and second-order procedures that operate on these data types, and for statements about the data types and procedures. A *base type* is a type variable $@A$ or an expression of the form $\mathit{str}[\tau_1, \ldots, \tau_k]$, where $\tau_1, \ldots, \tau_k$ are base types and str is a k-ary type constructor ($k \geq 0$). A *type* is a base type or an expression of the form $\tau_1 \times \ldots \times \tau_k \to \tau$ for types $\tau_1, \ldots, \tau_k, \tau$. *Type constructors* are defined by expressions of the following form:

$$\texttt{structure}\ \mathit{str}[@A_1, \ldots, @A_k] <= \ \ldots,\ \mathit{cons}(\mathit{sel}_1 : \tau_1, \ldots, \mathit{sel}_n : \tau_n),\ \ldots$$

The τ_j are base types, and str may only occur as $\mathit{str}[@A_1, \ldots, @A_k]$ in the τ_j. Each *cons* is called a *data constructor* and the sel_j are called *selectors*.

Let $\Sigma(P)$ denote the signature of all function symbols defined by an $\mathcal{L}$-program P. As usual, $\mathcal{T}(\Sigma(P), \mathcal{V})$ denotes the set of all *terms* over $\Sigma(P)$ and a set $\mathcal{V}$ of variables. We write $\mathcal{T}(\Sigma(P))$ instead of $\mathcal{T}(\Sigma(P), \emptyset)$ for the set of all *ground terms* over $\Sigma(P)$. $\Sigma(P)^c \subset \Sigma(P)$ contains all data constructors of P. A *literal* is an *if*-free Boolean term or the negation *if b then false else true* of such a term. $\mathcal{CL}(\Sigma(P), \mathcal{V})$ is the set of *clauses* over $\Sigma(P)$, i. e., the set of all finite sets of literals. For a term $t \in \mathcal{T}(\Sigma(P), \mathcal{V})$, we let $\Pi(t) \subset \mathbb{N}^*$ denote the set of all positions of t, i. e., $\Pi(t)$ comprises the positions of all subterms of t. We write $t|_\pi$ for the subterm of t at position $\pi \in \Pi(t)$.

For a ground type[3] τ, $\mathbb{V}(P)_\tau$ denotes the "values" of type τ: If τ is a ground base type, $\mathbb{V}(P)_\tau := \mathcal{T}(\Sigma(P)^c)_\tau$, and for each ground type $\tau = \tau_1 \times \ldots \times \tau_k \to \tau_{k+1}$, $\mathbb{V}(P)_\tau$ contains all closed (i. e., no free variables) λ-expressions of type τ; e. g., $\lambda t : \mathit{term}.\ \mathit{varcount}(t) \in \mathbb{V}(P)_{\mathit{term} \to \mathbb{N}}$.

The call-by-value interpreter $\mathsf{eval}_P : \mathcal{T}(\Sigma(P)) \mapsto \mathbb{V}(P)$ defines the operational semantics of $\mathcal{L}$ [1] by mapping ground terms $t \in \mathcal{T}(\Sigma(P))_\tau$ to values $\mathsf{eval}_P(t) \in \mathbb{V}(P)_\tau$. It is a partial function, because some procedures in program P may not terminate. A universally quantified formula of the form $\forall x_1 : \tau_1, \ldots, x_n : \tau_n.\ b$, where $b \in \mathcal{T}(\Sigma(P), \mathcal{V})_{\mathit{bool}}$, is *true* iff all procedures in P terminate and $\mathsf{eval}_{P'}(b[q_1, \ldots, q_n]) = \mathit{true}$ for each terminating program $P' \supseteq P$ and all $q_1, \ldots, q_n \in \mathbb{V}(P')$.[4]

We implicitly assume procedure bodies to be in η-long form; e. g., $\mathit{map}(f, \mathit{tl}(k))$ abbreviates $\mathit{map}(\lambda z : @A.\ f(z), \mathit{tl}(k))$ in Fig. 1, because $f =_\eta \lambda z : @A.\ f(z)$. The following definition formalizes the notion "$f(q)$ requires the evaluation of $g(q')$":

Definition 1. *For a procedure or λ-expression f with body B_f and parameters $x_1, \ldots, x_n$, a procedure or λ-expression g, and $q_1, \ldots, q_n$, $q'_1, \ldots, q'_m \in \mathbb{V}(P)$, we write $f(q_1, \ldots, q_n) \rhd g(q'_1, \ldots, q'_m)$ iff B_f contains a subterm $h(t'_1, \ldots, t'_m)$ under some call context*[5] *C such that for $\sigma := \{x_1/q_1, \ldots, x_n/q_n\}$, $\sigma(h) =_\eta g$, $\mathsf{eval}_P(\sigma(c)) = \mathit{true}$ for all $c \in C$, and $q'_j = \mathsf{eval}_P(\sigma(t'_j))$ for all $j = 1, \ldots, m$. We write $f(q_1, \ldots, q_n) \rhd_g g(q'_1, \ldots, q'_m)$ iff $f(q_1, \ldots, q_n) \rhd h_1(\ldots) \rhd \ldots \rhd h_k(\ldots) \rhd g(q'_1, \ldots, q'_m)$ such that $h_i \neq_\eta g$ for all $i = 1, \ldots, k$.*

For example, $\mathit{map}(\mathit{varcount},\ t_1 :: t_2 :: t_3 :: \varnothing) \rhd \mathit{varcount}(t_1)$.

[3] A *ground (base) type* is a (base) type without type variables; e. g., $\mathit{list}[\mathbb{N}]$.

[4] Program P' may define additional data types and procedures to instantiate the x_i.

[5] $C \in \mathcal{CL}(\Sigma(P), \mathcal{V})$ consists of the conditions in B_f that lead to the call $h(\ldots)$.

```
procedure $every(f : @A \to bool,\ k : list[@A]) : bool$ <=
$if\ k = \emptyset\ then\ true\ else\ if\ f(hd(k))\ then\ every(f, tl(k))\ else\ false$
procedure $foldl(f : @A \times @B \to @A,\ x : @A,\ k : list[@B]) : @A$ <=
$if\ k = \emptyset\ then\ x\ else\ foldl(f,\ f(x, hd(k)),\ tl(k))$
procedure $groundterm(t : term) : bool$ <=
$if\ ?var(t)\ then\ false\ else\ every(groundterm, args(t))$
procedure $termsize(t : term) : \mathbb{N}$ <=
$if\ ?var(t)\ then\ 1\ else\ foldl(\lambda n : \mathbb{N}, s : term.\ n + termsize(s),\ 1, args(t))$
```

Fig. 2. Second-order recursion in procedures *groundterm* and *termsize*

3 Quantification Procedures

Quantification procedures are system-generated procedures that iterate over certain values z and check if a given predicate p is satisfied for all these values z.

Quantification Procedures for Data Types. Consider the usual structural induction axiom for terms: In the base case, one proves that $\psi[t]$ holds if t is an arbitrary variable. In the step case, t is of the form $f(t_1, \ldots, t_n)$ and one proves $\psi[t]$ under the induction hypothesis that "$\psi[t_i]$ holds for all $i = 1, \ldots, n$". In general a program does not contain procedures to access the i-th element of list $args(t) = t_1 :: \ldots :: t_n :: \emptyset$ or to quantify over all elements of list $args(t)$. Hence we assume that for each data type $str[@A]$ a *quantification procedure*

$$\texttt{procedure}\ forall.str(p : @A \to bool,\ x : str[@A]) : bool \tag{3}$$

is synthesized that returns *true* iff $p(z)$ holds for all items $z : @A$ in x. PVS and √eriFun synthesize such quantification procedures automatically [11,1].

Example 3. For data type $list[@A]$, Fig. 3(a) shows quantification procedure *forall.list* that checks if some predicate p on $@A$ is satisfied for all elements $z : @A$ of a list k. Thus the axiom for structural induction on terms can be expressed (and automatically extracted by PVS and √eriFun) as

$$\frac{\begin{array}{c}\forall t : term.\ ?var(t) \to \psi[t] \\ \forall t : term.\ ?apply(t) \wedge forall.list(\lambda s : term.\ \psi[s],\ args(t)) \to \psi[t]\end{array}}{\forall t : term.\ \psi[t]}$$

where the induction hypothesis $forall.list(\ldots)$ states that $\psi[s]$ may be assumed for all terms s in list $args(t)$. ♢

Quantification Procedures for Second-Order Procedures. As Example 2 shows, the recursion analysis for procedure *varcount* needs to find out which arguments z the second-order procedure *map* calls its first-order parameter $f := varcount$ with. The induction hypothesis in the induction axiom for *varcount* will then quantify over all these arguments z to ensure $\psi[z]$.

```
(a) procedure $forall.list(p : @A \to bool,\ k : list[@A]) : bool$ <=
    $if\ k = \emptyset\ then\ true\ else\ if\ p(hd(k))\ then\ forall.list(p, tl(k))\ else\ false$

(b) procedure $forall.map(p : @A \to bool,\ f : @A \to @B,\ k : list[@A]) : bool$ <=
    $if\ k = \emptyset\ then\ true\ else\ if\ p(hd(k))\ then\ forall.map(p, f, tl(k))\ else\ false$

    procedure $forall.every(p, f : @A \to bool,\ k : list[@A]) : bool$ <=
    $if\ k = \emptyset\ then\ true$
          $else\ \ if\ p(hd(k))\ then\ if\ f(hd(k))\ then\ forall.every(p, f, tl(k))\ else\ true$
                                $else\ \ false$

    procedure $forall.foldl(p : @A \times @B \to bool,\ f : @A \times @B \to @A,$
                            $x : @A,\ k : list[@B]) : bool$ <=
    $if\ k = \emptyset\ then\ true$
          $else\ \ if\ p(x, hd(k))\ then\ forall.foldl(p, f, f(x, hd(k)), tl(k))\ else\ false$
```

Fig. 3. Automatically synthesized quantification procedures

For that purpose we introduce a new concept, namely quantification procedures *forall.proc* for second-order procedures *proc*. For the sake of readability, we define *forall.proc* for second-order procedures *proc* with one first-order parameter f and an (optional) second formal parameter x. This definition can be generalized to more parameters in a straightforward way [1].

Definition 2. *For each terminating second-order procedure*

$$\texttt{procedure}\ proc(f : \tau_1 \times \ldots \times \tau_m \to \tau_f,\ x : \tau_x) : \tau_{proc} <= B_{proc}$$

the quantification procedure *forall.proc* for *proc is defined by*

$$\texttt{procedure}\ forall.proc(p : \tau_1 \times \ldots \times \tau_m \to bool,\\ f : \tau_1 \times \ldots \times \tau_m \to \tau_f,\ x : \tau_x) : bool <= \mathsf{ALL}_f(B_{proc})$$

where

$$\begin{aligned}
&\mathsf{ALL}_f(v) := true\\
&\mathsf{ALL}_f(f(t_1, \ldots, t_m)) := p(t_1, \ldots, t_m) \wedge \mathsf{ALL}_f(t_1) \wedge \ldots \wedge \mathsf{ALL}_f(t_m)\\
&\mathsf{ALL}_f(g(t_1, \ldots, t_n)) := \mathsf{ALL}_f(t_1) \wedge \ldots \wedge \mathsf{ALL}_f(t_n)\\
&\mathsf{ALL}_f(h(\lambda \boldsymbol{y}.\, t,\ t')) := \mathsf{ALL}_f(t') \wedge forall.h(\lambda \boldsymbol{y}.\, \mathsf{ALL}_f(t),\ \lambda \boldsymbol{y}.\, t,\ t')\\
&\mathsf{ALL}_f(if\ t_1\ then\ t_2\ else\ t_3) := \mathsf{ALL}_f(t_1) \wedge if\ t_1\ then\ \mathsf{ALL}_f(t_2)\ else\ \mathsf{ALL}_f(t_3)
\end{aligned}$$

for any variable v, any first-order function $g \neq if$, $g \neq f$, and any second-order procedure h (including proc). We write $\boldsymbol{y}$ as an abbreviation of $y_1, \ldots, y_k$, and $A \wedge B$ abbreviates "if A then B else false".

Quantification procedure *forall.proc* checks if $p(z_1, \ldots, z_m)$ holds for all tuples $(z_1, \ldots, z_m)$ that occur as arguments of f-calls:

Example 4. Procedure *forall.map* shown in Fig. 3 checks if $p(z)$ is satisfied for all elements z of list k, as procedure *map* applies f to all elements z of k. ♢

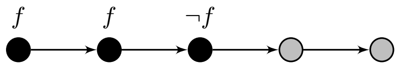


Fig. 4. Procedure *every* examines only the black elements of this list

Example 5. Procedure *every* in Fig. 2 checks if $f(z)$ is satisfied for all elements z of list k. As soon as an element z is encountered with $\neg f(z)$, procedure *every* stops with result *false*. This is illustrated in Fig. 4, where *every* evaluates $f(z)$ only for the black elements of the list. Consequently, procedure *forall.every* in Fig. 3 checks if $p(z)$ is satisfied for the first n elements of k, where $n \in \{1, \ldots, |k|\}$ is the smallest index such that f is *not* satisfied for the n-th element of k. (If there is no element z with $\neg f(z)$, then $n := |k|$, the length of k.) ♢

Example 6. Procedure *forall.foldl* checks if $p(a, b)$ is satisfied for all pairs (a, b) that f is applied to by *foldl*. ♢

The following lemma asserts that the quantification procedures according to Definition 2 compute the expected result. It demands that p and f be *fresh* functions, which means that these functions do not occur in the body of *proc* or in the bodies of auxiliary procedures for *proc*. (Alternatively, one can imagine p and f as uniquely labeled to distinguish these function calls from hard-coded function calls in the procedure bodies.)

Lemma 1. *For all* $\mathsf{x} \in \mathbb{V}(P)$ *and all fresh functions* $\mathsf{p} \in \mathbb{V}(P)$ *and* $\mathsf{f} \in \mathbb{V}(P)$*:*

(1) $\mathsf{eval}_P(\mathit{forall.proc}(\mathsf{p}, \mathsf{f}, \mathsf{x})) \in \{\mathit{true}, \mathit{false}\}$
(2) $\mathsf{eval}_P(\mathit{forall.proc}(\mathsf{p}, \mathsf{f}, \mathsf{x})) = \mathit{true} \iff \mathsf{eval}_P(\mathsf{p}(q_1, \ldots, q_m)) = \mathit{true}$ *for all* $q_1, \ldots, q_m \in \mathbb{V}(P)$ *with* $\mathit{proc}(\mathsf{f}, \mathsf{x}) \rhd_{\mathsf{f}} \mathsf{f}(q_1, \ldots, q_m)$

Proof. The proof is given in [1] (Sect. 3.2.2). □

4 Synthesis of Induction Axioms

In order to synthesize an induction axiom for a procedure

$$\texttt{procedure}\ p(x : \tau) : \tau' <= B_p$$

we analyze the recursive calls in the body B_p of procedure p. In case of second-order recursion, the indirect recursive calls are nested in λ-expressions, so in general we need to analyze a subterm t of B_p.

A *result term of t* is a maximal subterm of t that occurs outside of *if*-conditions and λ-expressions and does not contain *if*-expressions. We define $\Pi_p^{\mathsf{base}}(t) \subseteq \Pi(t)$ as the set of the positions of the base cases of p in t, i. e., the positions of those result terms that do not contain calls of p. $\Pi_p^{\mathsf{rec1}}(t) \subseteq \Pi(t)$ denotes the set of the positions of *direct* recursive calls, i. e., calls $p(\ldots)$ outside of λ-expressions. Finally, $\Pi_p^{\mathsf{rec2}}(t) \subseteq \Pi(t)$ denotes the set of positions of *second-order recursive* *calls*, i. e., calls $p(\ldots)$ inside a λ-expression that is passed to a second-order procedure. For some $\pi \in \Pi_p(t) := \Pi_p^{\mathsf{base}}(t) \cup \Pi_p^{\mathsf{rec1}}(t) \cup \Pi_p^{\mathsf{rec2}}(t)$, we write C_t^π for the call context of the subterm at position π in t (i. e., the set of conditions that lead to π).

In the base and step cases of an inductive proof of $\forall x : \tau.\ \psi[x]$, $\psi[x]$ needs to be shown under certain premises. Given a subterm t of B_p and a position $\pi \in \Pi_p(t)$, the premise $\mathsf{Prem}_p^\pi(\psi, t)$ is constructed as follows:

- If $\pi \in \Pi_p^{\mathsf{base}}(t)$, we get a base case of the induction: $\mathsf{Prem}_p^\pi(\psi, t) := \bigwedge C_t^\pi$.
- If $\pi \in \Pi_p^{\mathsf{rec1}}(t)$, we have a recursive call $t|_\pi = p(t')$ for some t', which gives rise to an induction hypothesis: $\mathsf{Prem}_p^\pi(\psi, t) := \bigwedge C_t^\pi \wedge \psi[t']$.
- If $\pi \in \Pi_p^{\mathsf{rec2}}(t)$, then there is a minimal prefix π' of π such that $t|_{\pi'} = h(\lambda \boldsymbol{y}.\, t'',\ t')$ for some second-order procedure h, and t'' contains a recursive call at position $\pi'' \in \Pi_p(t'')$ that is a suffix of π. Thus we use the quantification procedure *forall.h* in the induction hypothesis to assert that $\psi[\ldots]$ holds for the arguments of the respective p-call within $\lambda \boldsymbol{y}.\, t''$:
  $\mathsf{Prem}_p^\pi(\psi, t) := \bigwedge C_t^{\pi'} \wedge \mathit{forall.h}\big(\lambda \boldsymbol{y}.\, \mathsf{Prem}_p^{\pi''}(\psi, t''),\ \lambda \boldsymbol{y}.\, t'',\ t'\big)$

These premises are used in the induction axiom for procedure p as follows:

Definition 3. *For a terminating procedure* `procedure` $p(x : \tau) : \tau' \mathrel{<=} B_p$, *the* induction axiom for p *is given by*

$$\frac{\big\{\forall x : \tau.\ \mathsf{Prem}_p^\pi(\psi, B_p) \rightarrow \psi[x] \bigm| \pi \in \Pi_p(B_p)\big\}}{\forall x : \tau.\ \psi[x]}\ .$$

Example 7. The base case of procedure *varcount* (cf. Fig. 1) is given by result term "1" under call context $\{?\mathit{var}(t)\}$. After η-expansion, second-order recursion occurs in $\mathit{map}(\lambda s : \mathit{term}.\, \mathit{varcount}(s),\ \mathit{args}(t))$. Thus the induction axiom is:

$$\frac{\begin{array}{l}\forall t : \mathit{term}.\ ?\mathit{var}(t) \rightarrow \psi[t] \\ \forall t : \mathit{term}.\ \neg\, ?\mathit{var}(t) \wedge \mathit{forall.map}(\lambda s : \mathit{term}.\, \psi[s],\ \mathit{varcount},\ \mathit{args}(t)) \rightarrow \psi[t]\end{array}}{\forall t : \mathit{term}.\ \psi[t]}$$

In the induction hypothesis, procedure *forall.map* asserts $\lambda s : \mathit{term}.\, \psi[s]$ for all calls of $\lambda s : \mathit{term}.\, \mathit{varcount}(s)$ by *map*. ♢

Example 8. In the induction axiom for *groundterm* (cf. Fig. 2), the step case is $\forall t : \mathit{term}.\ \neg\, ?\mathit{var}(t) \wedge \mathit{forall.every}(\lambda s : \mathit{term}.\, \psi[s],\ \mathit{groundterm},\ \mathit{args}(t)) \rightarrow \psi[t]$. ♢

Definition 3 can easily be generalized to accommodate procedures with more parameters. We illustrate this with two examples:

Example 9. Procedure *termsize* (cf. Fig. 2) is defined by second-order recursion via *foldl*, which receives a third argument that is just passed on to *forall.foldl*: The induction hypothesis

$$\mathit{forall.foldl}\big(\lambda n : \mathbb{N}, s : \mathit{term}.\, \psi[s],\ \lambda n : \mathbb{N}, s : \mathit{term}.\, n + \mathit{termsize}(s),\ 1,\ \mathit{args}(t)\big)$$

asserts $\psi[s]$ for all elements of $\mathit{args}(t)$. ♢

```
structure $predefinedSymbol$ <= $T, CONS, CAR, CDR, LIST, QUOTE, IF, \ldots$
structure $sexpr$ <=
  $nil,\ lispsymbol(name : predefinedSymbol),\ cons(car : sexpr, cdr : sexpr), \ldots$
structure $maybe[@A]$ <= $nothing,\ just(what : @A)$
procedure $mapsx(f : sexpr \to maybe[sexpr],\ x : sexpr) : maybe[sexpr]$ <= ...
procedure $eval(expr, va, fa : sexpr,\ n : \mathbb{N}) : maybe[sexpr]$ <=
  $\ldots mapsx(\lambda arg : sexpr.\ eval(arg, va, fa, n),\ cdr(expr)) \ldots$
```

Fig. 5. Excerpt from a LISP Interpreter *eval*

Example 10. In [6], Boyer and Moore describe a LISP Interpreter *eval* that evaluates LISP s-expressions (cf. Fig. 5). Since the evaluation of a LISP function call `(F T1 ... Tn)` requires the evaluation of the list `(T1 ... Tn)` of arguments, they introduce an auxiliary procedure

$$\texttt{procedure}\ evlist(expr, va, fa : sexpr,\ n : \mathbb{N}) : maybe[sexpr]$$

that considers *expr* as a list of s-expressions and successively evaluates these s-expressions by calling *eval* on each of them. Thus *eval* and *evlist* are mutually recursive. Due to lacking support of mutual recursion, Boyer and Moore merge both procedures into a single procedure *ev* that is parameterized by a flag to indicate if a single s-expression or a list of s-expressions is to be evaluated.

Second-order recursion provides a much more elegant way to implement the interpreter: Procedure *mapsx* considers parameter x as a list, applies f to $car(x)$, $car(cdr(x))$, $car(cdr(cdr(x)))$, ..., and returns an s-expression that represents the list of the result values. If an application of f yields *nothing*, the iteration stops and *mapsx* returns *nothing*. Procedure *eval* then uses second-order recursion via *mapsx* to evaluate a "list" $cdr(expr)$ of s-expressions.[6]

According to Definition 2 our approach synthesizes a quantification procedure $forall.mapsx(p : sexpr \to bool,\ f : sexpr \to maybe[sexpr],\ x : sexpr) : bool$ that checks $p(z)$ for all calls $f(z)$ by *mapsx*. In one of the step cases of the induction axiom for *eval* for a proof of $\forall expr, va, fa : sexpr,\ n : \mathbb{N}.\ \psi[expr, va, fa, n]$ the induction hypothesis is

$$forall.mapsx(\lambda arg : sexpr.\ \psi[arg, va, fa, n],\quad \lambda arg : sexpr.\ eval(arg, va, fa, n),\ cdr(expr))\ . \qquad \diamondsuit$$

Theorem 1. *The induction axiom from Definition 3 for a terminating procedure p is an instance of well-founded induction.*

Proof (sketch). The relation $\succ$ on τ, defined by $x \succ x'$ iff $p(x) \rhd_p p(x')$, is well-founded, because p terminates. This relation can be syntactically represented by

[6] Parameter *va* models the variable assignment, and *fa* associates function symbols with their definition. If the resource limit n for the evaluation of *expr* does not suffice, *eval* returns *nothing* as in [6]. The complete source code is several pages long [1].

a formula that may use the quantification procedures from Sect. 3. This formula can be used to instantiate the schema (1) of well-founded induction to obtain the induction axiom from Definition 3, see [1] (Sect. 5.2.2 and 5.3). □

Hence Definition 3 describes a method to extract induction axioms from the source code of procedures with second-order recursion. These induction axioms precisely mirror the recursive structure of the respective procedure.

5 Optimization of Induction Axioms

Induction axioms from terminating procedures often are overly specific and thus suboptimal [5,8,9,13,14]. This also holds for many induction axioms that are synthesized according to Definition 3. In the following, we describe optimization techniques for the case of second-order recursion.

Similarly to many existing optimization techniques for procedures without second-order recursion, our approach examines the termination proof of the respective procedure to find optimizations: Intuitively, “components” of induction axioms (e. g., subformulas or parameters) that are irrelevant for the termination proof are also irrelevant for the induction axiom, because well-foundedness of the underlying relation obviously does not depend on these components.

5.1 Optimization of Quantification Procedures

Quantification procedures as in Definition 2 play a pivotal role in induction axioms for procedures with second-order recursion. Our approach optimizes quantification procedures along the following three dimensions (in this order):

(1) Reduce the arity of the additional predicate p.
(2) Extend the range of the quantification.
(3) Reduce the number of parameters of the quantification procedure.

Optimizations along dimensions (1) and (3) obviously increase the readability of induction hypotheses by making them syntactically simpler. In addition, they facilitate a final polishing of induction axioms that simplifies their use in proofs. Optimizations along dimension (2) strengthen the induction hypotheses by generalizing them, so $\psi[z]$ may be assumed for further values z.

In the induction axiom for procedure *groundterm*, for example, the induction hypothesis $\mathit{forall.every}(\lambda s : \mathit{term}.\, \psi[s],\ \mathit{groundterm},\ \mathit{args}(t))$ only ensures that ψ holds on a prefix of list $\mathit{args}(t)$, because *every* in general only examines a prefix of list k (cf. Example 5). This is suboptimal, because from structural induction we know that it would be safe to assume that ψ holds for *all* elements of $\mathit{args}(t)$.

In a typical termination proof for procedure *groundterm*, one tries to show that the parameter of *groundterm* gets structurally smaller in recursive calls [2,8,11]. Clearly, $\mathit{args}(t)$ is structurally smaller than t, because the leading *apply*-constructor is missing. Procedure *every* applies $f := \mathit{groundterm}$ only to values $s \in \{\mathit{hd}(k),\ \mathit{hd}(\mathit{tl}(k)),\ \mathit{hd}(\mathit{tl}(\mathit{tl}(k))), \ldots\}$ for $k := \mathit{args}(t)$. Since each such value s is structurally not larger than $args(t)$, one concludes that each argument s of a recursive call of *groundterm* is structurally smaller than t, which proves termination of *groundterm*.

Apparently the proof that procedure *every* applies f only to values z that are structurally not larger than k does not use the fact that *every* stops as soon as it encounters an element z with $\neg f(z)$. Formally, condition $f(hd(k))$ from the body of *every* is not used in the proof. Thus *groundterm* would still terminate if *every* continued with the examination of list elements in case $\neg f(hd(k))$. Then the case analysis over $f(hd(k))$ in the body of *forall.every* would become unnecessary, and the induction hypothesis for *groundterm* would assert ψ for all elements of $args(t)$ as desired.

Consequently, we optimize quantification procedures as follows:

Definition 4. *Let* $proc(f : \tau_1 \times \ldots \times \tau_m \to \tau_f,\ x : \tau_x) : \tau_{proc}$ *be a procedure and* $i \in \{1, \ldots, m\}$*. Let* Prf *be a proof that proc calls f only with values* $q_1, \ldots, q_m$ *such that* q_i *is structurally not larger than x.*[7] *We say that proc is* call-bounded *wrt. the i-th argument of f and define the synthesis of the* optimized quantification procedure $forall_i^{opt}.proc$ for *proc as follows:*

(1) Procedure $forall_i^{opt}.proc(p : \tau_i \to bool,\ f : \tau_1 \times \ldots \times \tau_m \to \tau_f,\ x : \tau_x) : bool$ *is derived from forall.proc by replacing all subterms* $p(t_1, \ldots, t_m)$ *in the procedure body with* $p(t_i)$*.*
(2) For each case analysis over some term c in the body of proc such that c is not *used in* Prf*, the corresponding case analysis over c in the body of* $forall_i^{opt}.proc$ *is replaced with the conjunction of its branches.*
(3) Each unused parameter of $forall_i^{opt}.proc$ *is removed.*

Call-bounded procedures can be identified by the approach in [2], for example. Unused conditions c of case analyses can be read off from proofs Prf.

Example 11. Procedure *foldl* is call-bounded wrt. the 2nd argument of f, so step (1) reduces the arity of p to $p : @B \to bool$. In step (3), parameters x and f are removed from $forall_2^{opt}.foldl$ (in this order). Thus

```
procedure $forall_2^{opt}.foldl(p : @B \to bool,\ k : list[@B]) : bool$ <=
$if\ k = \emptyset\ then\ true\ else\ if\ p(hd(k))\ then\ forall_2^{opt}.foldl(p, tl(k))\ else\ false$
```

checks $p(z)$ for all elements z of k, and $forall_2^{opt}.foldl(p, k) \leftrightarrow forall.list(p, k)$. ♢

Example 12. For *forall.every*, steps (2) and (3) apply: A proof that *every* is call-bounded does not use condition $c := f(hd(k))$ (i. e., the fact that *every* stops the iteration over list k when $\neg f(hd(k))$ holds). Thus the case analysis over $f(hd(k))$ in the body of *forall.every* can be removed, and parameter f is no longer used. Hence $forall^{opt}.every$ in addition checks p for the gray elements in Fig. 4, and $forall^{opt}.every(p, k) \leftrightarrow forall.list(p, k)$. ♢

Example 13. For procedure *forall.map*, only step (3) applies, which removes the unused parameter f, so $forall^{opt}.map(p, k) \leftrightarrow forall.list(p, k)$. ♢

[7] The structural size of values can be determined by a size measure as in [2].

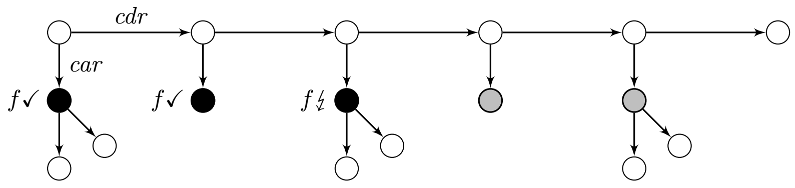


Fig. 6. Procedure *mapsx* applies f to the black entries of this s-expression

Example 14. Fig. 6 shows an exemplary s-expression x. When applying *mapsx* to x, function f is potentially applied to the black and the gray nodes (cf. Example 10). A node z is labeled with "$f\checkmark$" if $?just(f(z))$, whereas "$f\lightning$" means $f(z) = nothing$. As "$f\lightning$" holds for the third black node, procedure *mapsx* stops here and does not apply f to the gray nodes. Since a proof that *mapsx* is call-bounded (i. e., that f is only applied to s-expressions z that are structurally not larger than the whole s-expression x) does not use the fact that the iteration may stop early, the optimized quantification procedure

$$\texttt{procedure}\ forall^{opt}.mapsx(p : sexpr \rightarrow bool,\ x : sexpr) : bool$$

checks $p(z)$ for both the black and the gray nodes. ♢

5.2 Optimized Induction Hypotheses For Second-Order Recursion

We optimize induction axioms for procedures with second-order recursion by using the optimized quantification procedures if possible:

Definition 5. *Let p be terminating procedure. If the termination proof for p exploits that some second-order procedure h is call-bounded, the* optimized induction axiom *for p is obtained by replacing forall.h with forall^{opt}.h in the induction axiom from Definition 3. If forall^{opt}.h is equivalent*[8] *to forall.str for some type constructor str, then forall.str is used instead of forall^{opt}.h.*

Example 15. The optimized induction axioms for *varcount*, *groundterm*, and *termsize* are equivalent to the structural induction axiom from Example 3:

$$\frac{\begin{array}{l}\forall t : term.\ ?var(t) \rightarrow \psi[t] \\ \forall t : term.\ \neg\, ?var(t) \wedge forall.list(\lambda s : term.\, \psi[s],\ args(t)) \rightarrow \psi[t]\end{array}}{\forall t : term.\ \psi[t]}$$

This induction axiom is significantly stronger than the non-optimized induction axiom for *groundterm*: In the optimized axiom $\psi[s]$ may be assumed for *all* terms s in list $args(t)$ as induction hypothesis. In contrast, in the non-optimized axiom $\psi[s]$ may only be assumed for the first n terms in $args(t)$, where n is the index of the first term s in $args(t)$ with $\neg\, groundterm(s)$. ♢

[8] Syntactical identity up to a renaming of formal parameters is a sufficient and practically useful criterion for equivalence of quantification procedures.

Example 16. For the LISP interpreter of Example 10 and some s-expression $\mathit{cdr}(\mathit{expr})$ as in Fig. 6, the induction hypothesis asserts $\psi[\mathit{arg}, \ldots]$ only for black entries before the optimization (where f corresponds to the LISP interpreter eval). After the optimization, $\mathit{forall}^{\mathit{opt}}.\mathit{mapsx}(\lambda \mathit{arg} : \mathit{sexpr}.\, \psi[\mathit{arg}, n],\ \mathit{cdr}(\mathit{expr}))$ asserts $\psi[\mathit{arg}, \ldots]$ also for the gray nodes (i. e., for all elements of the "list"). $\diamondsuit$

As the examples demonstrate, the optimization leads to intuitive induction axioms. The induction hypotheses correspond to the recursive calls of the respective procedure without being restricted by unnecessary preconditions.

Theorem 2. *The optimized induction axiom from Definition 5 for a terminating procedure p is an instance of well-founded induction.*

Proof (sketch). The optimization drops case analyses (in quantification procedures $\mathit{forall}.h$) on conditions that are irrelevant for the termination proof of p. Thus there is a modified copy p' of p where these case analyses are dropped in h (cf. Sect. 5.2.3 in [1]). Procedure p' terminates and the non-optimized induction axiom for p' is equivalent to the optimized induction axiom for p. $\square$

6 Related Work

In Isabelle [8,10,12] induction theorems are synthesized (and proved within Isabelle's higher-order logic) for terminating procedures and data types. Since higher-order logic is *not* a programming language and thus lacks an operational semantics, Isabelle cannot determine which function calls are required to evaluate a given term. Therefore, induction axioms for procedures with second-order recursion cannot be synthesized from just the source code. To solve this problem, the user can specify *congruence rules* by proving *congruence theorems* such as $k = k' \wedge \big(\forall z : @A.\ z \in k \rightarrow f(z) = f'(z)\big) \rightarrow \mathit{map}(f, k) = \mathit{map}(f', k')$, which tells Isabelle that for $\mathit{map}(f, k)$ the values $f(z)$ for at most all $z \in k$ are relevant. The resulting induction theorem for procedure *varcount* is equivalent to our induction axiom from Example 15. Syntactically, the quantification over the elements s in $\mathit{args}(t)$ is expressed by $\forall s : \mathit{term}.\ s \in \mathit{args}(t) \rightarrow \psi[s]$, where the notion "$\in$" of list membership stems from the user's congruence rule. Thus the induction theorems directly depend on the congruence rules, and the only way to optimize induction theorems is to (manually) modify the congruence rules. However, this becomes impossible when two function calls require different sets of congruence rules (e. g., see the example with procedure *testany* in [8]), so "in general, there is no 'best' or 'complete' set of congruence rules" [8]. Apart from that, the induction theorem for data type *term* is different from the usual structural induction and targets the simultaneous proof of two formulas $\forall t : \mathit{term}.\ \phi[t]$ and $\forall k : \mathit{list}[\mathit{term}].\ \psi[k]$ based on the mutual recursion of types *term* and $\mathit{list}[\mathit{term}]$.

In contrast, PVS [11] synthesizes quantification procedures for parameterized data types such as $\mathit{list}[@A]$ and uses these procedures for structural induction axioms (e. g., for data type *term*). While PVS uses *constructor induction*, our induction axioms use *destructor induction*. PVS does *not* synthesize induction axioms for (terminating) procedures and hence does not offer techniques to optimize induction axioms.

In ACL2 [5,9] induction axioms are synthesized for data types and for terminating procedures. Induction axioms are optimized using various techniques (e. g., [9]). However, procedures cannot be defined by second-order recursion.

For Coq, Barthe et al. [4] describe a tool that synthesizes induction axioms for terminating procedures, but second-order recursion is not supported.

Bundy et al. [7] developed a technique to construct induction axioms for the *synthesis* of procedures. In their approach, the goal is to find *novel* induction axioms that do not correspond to the recursive structure of existing procedures. Second-order recursion is not considered in this approach.

7 Conclusion

Our approach to automatically extract induction axioms from terminating procedures consists of two main steps: Firstly, it synthesizes induction axioms that precisely mirror the recursive structure of the procedures. For procedures with second-order recursion, the indirect recursive calls are captured using so-called *quantification procedures* that are synthesized automatically for the respective second-order procedures. Secondly, induction axioms are optimized automatically (i. e., generalized and simplified) by inspecting the termination proofs of the respective procedures. For that purpose our approach in particular optimizes the quantification procedures to strengthen the induction hypotheses.

The vision behind our approach is that a degree of automation can be achieved for the verification of *second-order programs* that is comparable to highly automated verification tools for *first-order programs*, e. g., ACL2. Practical experiments in √eriFun[9] (involving 21 procedures with second-order recursion, 14 main theorems and 28 auxiliary lemmas) showed that our methods in fact synthesize induction axioms that are neither too specific (as "precise" induction axioms tend to be) nor too general (as axioms for *complete induction* would be). This facilitates intuitive proofs, i. e., proofs that are quite similar to what one would do using paper and pencil. Hence our approach contributes to achieving such a high degree of automation.

For example, the optimization of induction axioms considerably simplifies the proof that $varcount(t) = 0$ implies $groundterm(t)$. With the optimization, a simple auxiliary lemma is required: If $p(z_i)$ and $p(z_i) \rightarrow q(z_i)$ hold for all elements z_i of a list k, then $q(z_i)$ holds for all elements z_i of k. Without the optimization, the user needs to discover and prove a much more complicated auxiliary lemma: Let n be the index of the first element z_n of a list k with $\neg\, q(z_n)$, or $n := |k|$ if there is no such element in k; if $p(z_i)$ and $p(z_i) \rightarrow q(z_i)$ hold for the first n elements z_i of k, then $q(z_i)$ holds for *all* elements z_i of k.

We expect that our approach can be transferred to other programming languages with call-by-value semantics; for ML, this might require to also consider axioms for constructor-style induction. Our commitment to an evaluation strategy makes it possible to uniformly determine which function calls need to be evaluated for a given term. In contrast, Isabelle does not commit to an evaluation

[9] See `http://www.mais.informatik.tu-darmstadt.de/Markus_Aderhold.html`

strategy; the price for this increased flexibility is that the user needs to formulate and prove additional theorems that at least approximate an evaluation strategy for particular functions.

Procedures in *continuation passing style* provide numerous additional examples of second-order recursion, because there *each* procedure has a function parameter (representing the continuation). However, in certain cases this may involve indirect recursive calls in continuations of direct recursive calls, which we leave as an area for further research.

Acknowledgment. I am grateful to Nathan Wasser for the implementation of the approach and to the anonymous referees for constructive feedback.

References

1. Aderhold, M.: Verification of Second-Order Functional Programs. Doctoral dissertation, TU Darmstadt (2009)
2. Aderhold, M.: Automated termination analysis for programs with second-order recursion. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 221–235. Springer, Heidelberg (2010)
3. Andrews, P.B.: An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof. Kluwer Academic Publishers, Dordrecht (2002)
4. Barthe, G., Forest, J., Pichardie, D., Rusu, V.: Defining and reasoning about recursive functions: A practical tool for the Coq proof assistant. In: Hagiya, M., Wadler, P. (eds.) FLOPS 2006. LNCS, vol. 3945, pp. 114–129. Springer, Heidelberg (2006)
5. Boyer, R.S., Moore, J.S.: A Computational Logic. Academic Press, Inc., London (1979)
6. Boyer, R.S., Moore, J.S.: A mechanical proof of the unsolvability of the halting problem. Journal of the ACM 31(3), 441–458 (1984)
7. Bundy, A., Dixon, L., Gow, J., Fleuriot, J.: Constructing induction rules for deductive synthesis proofs. In: Proceedings of Constructive Logic for Autom. Softw. Engineering 2005. ENTCS, vol. 153, pp. 3–21. Elsevier, Amsterdam (2006)
8. Krauss, A.: Automating Recursive Definitions and Termination Proofs in Higher-Order Logic. Doctoral dissertation, TU München, Germany (2009)
9. Manolios, P., Turon, A.: All-termination(T). In: Kowalewski, S., Philippou, A. (eds.) TACAS-2009. LNCS, vol. 5505, pp. 398–412. Springer, Heidelberg (2009)
10. Nipkow, T., Paulson, L.C., Wenzel, M.T. (eds.): Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)
11. Owre, S., Shankar, N., Rushby, J.M., Stringer-Calvert, D.W.J.: PVS Language Reference. Computer Science Laboratory, SRI International (November 2001)
12. Slind, K.: Reasoning about Terminating Functional Programs. PhD thesis, TU München, Germany (1999)
13. Walther, C.: Computing induction axioms. In: Voronkov, A. (ed.) LPAR 1992. LNCS, vol. 624, pp. 381–392. Springer, Heidelberg (1992)
14. Walther, C.: Mathematical induction. In: Handbook of Logic in Artificial Intelligence and Logic Programming, vol. 2. Oxford University Press, Oxford (1994)
15. Walther, C., Aderhold, M., Schlosser, A.: The $\mathcal{L}$ 1.0 Primer. Technical Report VFR 06/01, TU Darmstadt (2006)
16. Walther, C., Schweitzer, S.: Verification in the classroom. Journal of Automated Reasoning 32(1), 35–73 (2004)

Focused Inductive Theorem Proving

David Baelde[1], Dale Miller[2], and Zachary Snow[1]

[1] Digital Technology Center and Dept of CS, University of Minnesota
[2] INRIA & LIX, École Polytechnique

Abstract. *Focused proof systems* provide means for reducing and structuring the non-determinism involved in searching for sequent calculus proofs. We present a focused proof system for a first-order logic with inductive and co-inductive definitions in which the introduction rules are partitioned into an *asynchronous* phase and a *synchronous* phase. These focused proofs allow us to naturally see proof search as being organized around interleaving intervals of computation and more general deduction. For example, entire Prolog-like computations can be captured using a single synchronous phase and many model-checking queries can be captured using an asynchronous phase followed by a synchronous phase. Leveraging these ideas, we have developed an interactive proof assistant, called Tac, for this logic. We describe its high-level design and illustrate how it is capable of automatically proving many theorems using induction and coinduction. Since the automatic proof procedure is structured using focused proofs, its behavior is often rather easy to anticipate and modify. We illustrate the strength of Tac with several examples of proved theorems, some achieved entirely automatically and others achieved with user guidance.

1 Introduction

The sequent calculus of Gentzen is a well-studied proof framework used to describe provability for a number of logics. This framework also seems to be a natural setting for organizing the search for proofs in a theorem prover. For example, a sequent of the form $\Sigma; \Gamma \vdash B$ denotes the obligation of showing that the formula B follows from the assumptions in the (multi)set Γ for every instantiation of the variables in Σ. An attempt to prove a formula, say B_0, then gives rise to attempts to apply inference rules repeatedly to the root sequent $\cdot; \cdot \vdash B_0$ leaving, at some point, open premises $\Sigma_1; \Gamma_1 \vdash B_1, \ \ldots, \ \Sigma_n; \Gamma_n \vdash B_n$. This set of sequents represents one way to decompose the original proof obligation into $n \geq 0$ subgoals. The frontier of the open proof tree can represent the abstract state of an idealized theorem prover.

The sequent calculus is, unfortunately, far too non-deterministic to directly organize a theorem prover. For example, consider the case when there are even just two hypotheses on which to work. The sequent calculus does not specify on which to work first, and so one might first work on one, then the other, alternating back and forth. This creates an explosive number of alternatives to explore, many of which are often redundant. Similarly, the structural rules of weakening and contraction can be applied, in principle, to every formula anytime.

Anyone who has attempted to build a theorem prover based on the sequent calculus (or related systems such as tableaux) has undoubtedly observed that there are different ways to give some structure to many of these choices. For example, some inference rules are invertible and, as a result, choices in the order of their application do not affect provability. Additionally, sometimes when a formula is introduced it no longer needs to be maintained: thus the contraction rule need not be considered for that formula. Finally, sometimes selecting one formula for introduction can be seen as causing a cascade of other introduction rules. In recent years, a series of *proof theory* papers have appeared that present various *focused proof systems* for classical and intuitionistic logic. These new proof systems formalize exactly these kinds of observations and turn them into elegant and deep normal form theorems.

Focused proof systems require classifying connectives into two *polarities*, called synchronous and asynchronous. From a proof-search point of view, asynchronous connectives can be introduced early and in any order, since these connectives generally have invertible inference rules. In contrast, once a synchronous formula has been chosen for introduction, then all synchronous subformulas must also be selected immediately for introduction: the synchronous phase ends when the proof is finished or only asynchronous subformulas are reached. This discipline gives rise to the notion of *synthetic* connectives aggregating logical connectives of the same polarity, and focused proof systems can be seen as introducing such synthetic connectives, building their introduction rules from collections of individual, small, introduction rules. The identification of synthetic connectives allows us to view proof search in sequent calculus as revolving around *big step* inference rules and not the usual *small step* introduction rules of Gentzen. Furthermore, since the proof theory behind focused proof systems for classical and intuitionistic logic contains some ambiguity (for example, conjunctions and atoms can be considered as being part of an asynchronous or a synchronous phase), the theorem prover designer has some flexibility in what she wants to have as a synthetic connective. The formal results about focused proof systems provide a solid foundation for these engineered, big-step inference rules: they remain sound and complete with respect to the original small-step proof system.

From the perspective of designing a theorem prover, the above concepts are invaluable. As we shall see, focusing allows mixing computation and deduction in natural and transparent ways. For example, it is entirely possible to describe, say, the concatenation of two lists (in the relational style of Prolog) and then embed the entire computation of such a relationship within one synthetic connective. This is in striking contrast with the treatment in most resolution-style theorem provers where such a computation is emulated by a possibly large number of small-step resolution rules. Going one step further, a model-checking problem (*e.g.*, all members of one finite set are members of another set) can be naturally modeled as just two synthetic connectives: the first asynchronous (enumerating all members by case analysis) and the second synchronous (showing that they belong to the other set). Thus, the high-level viewpoint of proof brought by the notion of *synthetic connective* can make for more effective proof-search.

In this paper, we present the design and applications of an automatic theorem prover, called Tac. Section 2 describes μLJ, the logic underlying Tac: it is an intuitionistic logic containing least and greatest fixed points. In Section 3 we introduce a focused proof system for μLJ. In Section 4, we describe the high-level design of Tac, in particular its automatic proof-search strategy involving the use of (co)induction. This design has been governed by the following three principles. First, while Tac is an interactive prover based on tacticals, it is hoped that a single, automatic tactic can be used to fill in the gaps between a theorem and a list of (human supplied) lemmas. Second, the automatic tactic is organized around the search for focused proofs via the use of synthetic connectives. Third, the only influence we allow on the automatic tactic's behavior involves those aspects of focused proof systems that proof theory has not fixed. Section 5 summarizes the behavior of Tac and compares it to some other theorem proving systems.

2 The Logic μLJ

The logic μLJ [2] is the extension of first-order intuitionistic logic[1] with inductive and coinductive definitions given using least and greatest fixed points. The proof system for μLJ contains familiar rules for inductive and coinductive inference based on the selection of invariants, which notably provides the intuitionistic version of Peano's arithmetic. Its study is inspired by that of μMALL [4].

We consider the following simply typed language of formulas:

$$\begin{aligned} P ::= &\; P \wedge P \mid P \vee P \mid P \supset P \mid \bot \mid \top \\ \mid &\; \exists_\gamma x.\ P \mid \forall_\gamma x.\ P \mid s \overset{\gamma}{=} t \mid \mu_{\gamma_1 \ldots \gamma_n}(\lambda p \lambda \boldsymbol{x}.\ P)\boldsymbol{t} \mid \nu_{\gamma_1 \ldots \gamma_n}(\lambda p \lambda \boldsymbol{x}.\ P)\boldsymbol{t}. \end{aligned}$$

The syntactic variable γ represents a term type, *e.g.,* natural numbers or lists. The quantifiers have type $(\gamma \to o) \to o$ and the equality has type $\gamma \to \gamma \to o$. The least fixed point connective μ and the greatest fixed point connective ν have type $(\tau \to \tau) \to \tau$ where τ is $\gamma_1 \to \cdots \to \gamma_n \to o$ for some arity $n \geq 0$. We shall almost always elide the references to γ, assuming that they can be determined from the context when it is important to know their value. Note that we do not consider atoms, *i.e.,* predicate constants: although μLJ accomodates them without any problem, atoms are often unnecessary since fixed points play their role in practice, and thus we leave them out for simplicity.

Formulas with top-level connective μ or ν are called fixed point expressions. Fixed points can be arbitrarily nested and interleaved — that is, we can have mutually recursive definitions. The first argument of a fixed point connective is a predicate operator expression, called its *body*, and shall be denoted by B. In order for the logic to enjoy consistency and other useful properties, all fixed

[1] While intuitionistic logic is the natural choice for the specifications we consider, note that our proof-theoretical approach also applies well to linear or classical settings.

point bodies are required to be *monotonic*, *i.e.,* there should be no negative occurrence of the bound predicate variable p in $\lambda p \lambda \boldsymbol{x}.\ Bp\boldsymbol{x}$.

Example 1. Assuming a term type n and two constants $0 : n$ and $s : n \to n$, the natural number predicate nat of type $n \to o$ can be defined as the inductive expression μB_{nat}, where B_{nat} is defined as $\lambda N \lambda x.\ x = 0 \vee \exists y.\ x = s\ y \wedge N\ y$.

The inference rules of μLJ deal with usual first-order intuitionistic sequents, of the form $\Sigma; \Gamma \vdash P$ where Σ is a set of universal (eigen)variables $x_1, \ldots, x_n$ and Γ is a set of formulas $P_1, \ldots, P_m$. The logical reading of such a sequent is $\forall x_1 \ldots \forall x_n.\ (P_1 \wedge \ldots \wedge P_m \supset P)$.

The inference rules of μLJ are the usual ones for the propositional connectives and first-order quantifiers. The left and right-introduction rules for equality date back to [6,12]:

$$\frac{\{(\Sigma; \Gamma \vdash Q)\theta : \theta \in csu(t \doteq t')\}}{\Sigma; \Gamma, t = t' \vdash Q} =L \qquad \frac{}{\Sigma; \Gamma \vdash t = t} =R$$

In the left equality rule ($=L$), csu stands for *complete set of unifiers* [7]. This set can be restricted to have at most one element when terms are first-order but might be infinite if terms are interpreted modulo some algebraic theory or if they are simply typed λ-terms. The application of a substitution to the signature of a sequent consists in removing instantiated variables and adding newly introduced ones; the application to the rest of the sequent simply propagates it to the terms of every formula. Note that this treatment of equality is stronger than Leibniz equality, as it notably expresses the injectivity of term constructors. More generally, it provides our proof system with an approach to *negation-as-failure*: if the equality $t = t'$ is a *failure* (that is, $csu(t, t')$ is empty) then the equality left rule yields a *successful* proof (that is, the rule has no premises).

The least fixed point μB is characterized as the least of the prefixed points.

$$\frac{\Sigma; \Gamma, S\boldsymbol{t} \vdash P \qquad \boldsymbol{x}; BS\boldsymbol{x} \vdash S\boldsymbol{x}}{\Sigma; \Gamma, \mu B\boldsymbol{t} \vdash P}\ induction \qquad \frac{\Sigma; \Gamma \vdash B(\mu B)\boldsymbol{t}}{\Sigma; \Gamma \vdash \mu B\boldsymbol{t}}\ \mu\text{-}unfolding$$

The right *unfolding* rule expresses $B(\mu B)\boldsymbol{t} \supset \mu B\boldsymbol{t}$, and the left *induction* rule expresses that μB entails any prefixed point S, also called an *invariant*. Notice that the universal variables $\boldsymbol{x}$ in the induction rule are new.

From the induction rule one can always derive a left unfolding rule for μ, using the invariant $B(\mu B)$:

$$\frac{\Sigma; \Gamma, B(\mu B)\boldsymbol{t} \vdash P}{\Sigma; \Gamma, \mu B\boldsymbol{t} \vdash P}$$

Thus, the least prefixed point is a fixed point, *i.e.,* $\mu B\boldsymbol{x}$ and $B(\mu B)\boldsymbol{x}$ are provably equivalent. The introduction rules for greatest fixed points are the dual rules:

$$\frac{\Sigma; \Gamma, B(\nu B)\boldsymbol{t} \vdash P}{\Sigma; \Gamma, \nu B\boldsymbol{t} \vdash P}\ \nu\text{-}unfolding \qquad \frac{\Sigma; \Gamma \vdash S\boldsymbol{t} \qquad \boldsymbol{x}; S\boldsymbol{x} \vdash BS\boldsymbol{x}}{\Sigma; \Gamma \vdash \nu B\boldsymbol{t}}\ coinduction$$

Finally, the initial rule can be restricted to fixed point expressions.

$$\begin{aligned}
eq &\stackrel{def}{=} \mu(\lambda E \lambda x \lambda y.\ (x = 0 \wedge y = 0) \vee (\exists x' \exists y'.\ x = s\ x' \wedge y = s\ y' \wedge E\ x'\ y'))\\
leq &\stackrel{def}{=} \mu(\lambda L \lambda x \lambda y.\ x = y \vee (\exists y'.\ y = s\ y' \wedge L\ x\ y'))\\
half &\stackrel{def}{=} \mu(\lambda H \lambda x \lambda h.\ ((x = 0 \vee x = s\ 0) \wedge h = 0)\\
&\qquad\qquad \vee (\exists x' \exists h'.\ x = s^2\ x' \wedge h = s\ h' \wedge H\ x'\ h'))\\
append &\stackrel{def}{=} \mu(\lambda A \lambda x \lambda y \lambda z.\ (x = nil \wedge y = z)\\
&\qquad\qquad \vee (\exists e \exists x' \exists z'.\ x = e :: x' \wedge z = e :: z' \wedge A\ x'\ y\ z'))\\
reverse &\stackrel{def}{=} \mu(\lambda R \lambda l \lambda r.\ (l = nil \wedge r = nil)\\
&\qquad\qquad \vee (\exists h \exists l' \exists r'.\ l = h :: l' \wedge R\ l'\ r' \wedge append\ r'\ (h :: nil)\ r))\\
sim &\stackrel{def}{=} \nu(\lambda S \lambda p \lambda q.\ \forall l \forall p'.\ step\ p\ a\ p' \supset \exists q'.\ step\ q\ a\ q' \wedge S\ p'\ q')
\end{aligned}$$

Fig. 1. Examples of fixed point expressions

Example 2. In the particular case of *nat*, the induction rule with invariant S yields the usual induction principle:

$$\dfrac{\Sigma; \Gamma, S\ t \vdash P \qquad \dfrac{\vdash S(0) \quad y; S(y) \vdash S(s\ y)}{x; (B_{nat} S)x \vdash Sx}\ \vee L, \exists L, \wedge L, =L}{\Sigma; \Gamma, nat\ t \vdash P}$$

The logic μLJ results from a line of work on definitions [6,12] and induction and coinduction [9,11]. The presentation using μ and ν makes for a more direct proof theoretical study, and notably naturally brings the possibility to treat mutual (co)inductive definitions in an expressive way. Figure 1 contains several example fixed point definitions. These examples use *nil* as the empty list constructor and :: as the non-empty list constructor.

For brevity, we shall omit the signature Σ from the sequents in the next sections; its treatment should be clear from the above presentation.

3 Focused Proofs for μLJ

The proof system for μLJ described in the previous section is *unfocused* since there is no particular structure imposed on how one occurrence of an inference rule relates to another. In contrast, a focused proof system classifies inference rules into synchronous and asynchronous ones and then groups those of similar classification into one "synthetic introduction rule". The first focused proof system for a full logic was given by Andreoli for linear logic [1]. Eventually, focused proof systems have been developed for other logics where it was revealed that, unlike in linear logic, proof theory concerns do not fix all polarization choices (in particular, for atoms [1], conjunctions [8], and fixed points [2]). As a result, these non-fixed items could be placed into either the asynchronous or the synchronous phases: such choices do not affect provability but can have a striking effect on the size and shape of proofs. In linear logic, asynchronous connectives are exactly those with invertible right-introduction rules; this does not hold for richer logics, such as μLJ. Indeed, fixed points can always be treated in an invertible way (provability is never lost by unfolding) but completeness cannot be obtained with a strategy that eagerly unfolds all fixed points. For example, proving $nat\ x \vdash nat\ x$ cannot succeed by repeatedly unfolding the hypothesis; at some point, one has to stop unfolding and, instead, use the initial rule.

Definition 1 (Polarities for μLJ). *The connectives $\wedge$, $\vee$, $\exists$, $=$, and μ are* synchronous *while the connectives $\forall$, $\supset$ and ν are* asynchronous*. A synchronous (resp. asynchronous) formula is one whose top-level connective is synchronous (resp. asynchronous). If* every *connective of a formula is synchronous (resp. asynchronous), it is called* fully *synchronous (resp. asynchronous). Finally, a fixed point formula can be annotated as* frozen*, which is denoted by $(\mu B\boldsymbol{t})^*$ and $(\nu B\boldsymbol{t})^*$, in which case it is neither synchronous nor asynchronous.*

Figures 2, 3 and 4 present μLJF, a focused proof system for μLJ. There are two kinds of sequents: the *unfocused sequent* is written $\Gamma \vdash P$ (as before) and the *focused sequent* is written with a bracketed formula (the focus) as either $\Gamma \vdash [P]$ or $\Gamma, [P] \vdash Q$. In each of these sequents, Γ is a multiset of formulas. There is an unsurprising symmetry between left and right hand-sides of sequents: a synchronous connective is treated as asynchronous on the left and *vice-versa*. The *asynchronous phase* contains sequents of the form $\Gamma \vdash P$ and introduces asynchronous connectives on the right and synchronous ones on the left. The *synchronous phase* contains sequents containing one distinguished (bracketed) formula that is *under focus*. When the focus is on the right ($\Gamma \vdash [P]$) only the toplevel synchronous connectives of P can be introduced. When the focus is on the left ($\Gamma, [P] \vdash Q$) only toplevel asynchronous connectives of P can be introduced. The alternation between the two phases is allowed only when no other rule applies: the asynchronous phase ends when no synchronous formula remains on the left, and the conclusion is synchronous; the synchronous phase ends when the focus is on the left on a synchronous formula, or on the right on an asynchronous one. Finally, the structural rule of contraction is used (implicitly) only in the rule establishing a left focus formula — and thus, only for asynchronous formulas.

Each fixed point has two rules per phase: one of these rules treats the fixed point as a structured formula; the other treats it as an atom. The synchronous rules are unfolding and the initial rule and the asynchronous rules are (co)induction and *freezing*. A strong constraint of the asynchronous phase is that it requires that any least fixed point hypothesis (and greatest fixed point conclusion) is either immediately used for (co)induction (which includes unfolding) or frozen, in which case it can never again be unfolded or used for induction: it can only be used in an initial rule later in the proof. Also note that when one focuses on a fully synchronous least fixed point, such as nat and all predicates of Figure 1 except sim, focus can never be released. Hence, the proof has to be completed in that phase, eventually reaching units, equality, or the initial rule if an appropriate frozen side-formula is available.

The standard, unfocused proof system for μLJ can be recovered from μLJF by removing all focusing annotations.

$$\frac{\Gamma, P, P' \vdash Q}{\Gamma, P \wedge P' \vdash Q} \quad \frac{\Gamma, P \vdash Q \quad \Gamma, P' \vdash Q}{\Gamma, P \vee P' \vdash Q} \quad \frac{\Gamma, P \vdash Q}{\Gamma \vdash P \supset Q} \quad \frac{}{\Gamma, \bot \vdash P} \quad \frac{\Gamma \vdash P}{\Gamma, \top \vdash P}$$

$$\frac{\Gamma \vdash Px}{\Gamma \vdash \forall x.\ Px} \quad \frac{\Gamma, Px \vdash Q}{\Gamma, \exists x.\ Px \vdash Q} \quad \frac{\{(\Gamma \vdash P)\theta : \theta \in csu(t \doteq t')\}}{\Gamma, t = t' \vdash P}$$

$$\frac{\Gamma, S\boldsymbol{t} \vdash P \quad BS\boldsymbol{x} \vdash S\boldsymbol{x}}{\Gamma, \mu B\boldsymbol{t} \vdash P} \quad \frac{\Gamma, (\mu B\boldsymbol{t})^* \vdash P}{\Gamma, \mu B\boldsymbol{t} \vdash P} \quad \frac{\Gamma \vdash S\boldsymbol{t} \quad S\boldsymbol{x} \vdash BS\boldsymbol{x}}{\Gamma \vdash \nu B\boldsymbol{t}} \quad \frac{\Gamma \vdash (\nu B\boldsymbol{t})^*}{\Gamma \vdash \nu B\boldsymbol{t}}$$

Fig. 2. μLJF: asynchronous rules

$$\frac{\Gamma \vdash [P] \quad \Gamma \vdash [P']}{\Gamma \vdash [P \wedge P']} \quad \frac{\Gamma \vdash [P_i]}{\Gamma \vdash [P_0 \vee P_1]} \quad \frac{\Gamma, [P'] \vdash Q \quad \Gamma \vdash [P]}{\Gamma, [P \supset P'] \vdash Q} \quad \frac{}{\Gamma \vdash [\top]}$$

$$\frac{\Gamma, [Pt] \vdash Q}{\Gamma, [\forall x.\ Px] \vdash Q} \quad \frac{\Gamma \vdash [Pt]}{\Gamma \vdash [\exists x.\ Px]} \quad \frac{}{\Gamma \vdash [t = t]}$$

$$\frac{\Gamma, [B(\nu B)\boldsymbol{t}] \vdash P}{\Gamma, [\nu B\boldsymbol{t}] \vdash P} \quad \frac{}{\Gamma, [\nu B\boldsymbol{t}] \vdash (\nu B\boldsymbol{t})^*} \quad \frac{\Gamma \vdash [B(\mu B)\boldsymbol{t}]}{\Gamma \vdash [\mu B\boldsymbol{t}]} \quad \frac{}{\Gamma, (\mu B\boldsymbol{t})^* \vdash [\mu B\boldsymbol{t}]}$$

Fig. 3. μLJF: synchronous rules

$$\frac{\Gamma, Q, [Q] \vdash P}{\Gamma, Q \vdash P} \quad \frac{\Gamma \vdash [P]}{\Gamma \vdash P} \quad \frac{\Gamma, P \vdash Q}{\Gamma, [P] \vdash Q} \quad \frac{\Gamma \vdash Q}{\Gamma \vdash [Q]}$$

Fig. 4. μLJF: structural rules (P synchronous, Q asynchronous)

Theorem 1 (Completeness [2]). *The sequent $\Gamma \vdash P$ is provable in μLJ if and only if it is provable in μLJF.*

Although not visible in the statement of completeness, the asynchronous rules can be applied in any order — permuting them actually leaves the proof essentially unchanged.

As usual, the completeness of the focused proof system justifies a reading of logic based on synthetic connectives and synthetic introduction rules. A synthetic introduction rule for a synthetic synchronous connective is a big-step rule that has a focused sequent as its conclusion and which extends upwards until there are only unfocused sequents present. Dually, a synthetic introduction rule for a synthetic asynchronous connective is a big-step rule that has an unfocused sequent as its conclusion and which extends upwards until no asynchronous rule can be applied. Note that this is especially powerful with fixed points, since we can now have synthetic introduction rules built from unbounded numbers of micro-rules. An interesting particular case of this is that of fully synchronous formulas, such as *nat* and more generally any Prolog-style computation, which constitute a synthetic unit, with an infinity of synthetic introduction rules.

Example 3. Consider the synthetic introduction rule that ends with the right-focused sequent $\Gamma \vdash [(leq\ m\ n \wedge B_1) \vee (leq\ n\ m \wedge B_2)]$, where m and n are natural numbers (terms over s and 0) and leq is the purely synchronous fixed point in Figure 1 denoting the less-than-or-equal-to relation. If both B_1 and B_2 are asynchronous formulas, then there are exactly two possible synthetic rules: one with premise $\Gamma \vdash B_1$ when $m \leq n$ and one with premise $\Gamma \vdash B_2$ when $n \leq m$ (thus, if $m = n$, both premises are possible). In this sense, a synthetic synchronous connective can contain an entire Prolog-style computation.

Being complete, the focused proof system for μLJ obviously does not render theorem proving decidable. But the focused structure of proofs can be very useful when building a theorem prover, as we shall see in the main contribution of this paper: the design of an automatic tactic that is organized around synthetic connectives.

4 Tac

There are several ways to exploit focusing for proof-search. For instance, the inverse method — performing top-down proof-search — yields impressive results when combined with focusing [5,10]. Proof search that must generate (co)invariants is hard to do in such a top-down style, particularly when contexts are used to generate the (co)invariants. Thus, we use bottom-up proof search. As we outline next, that choice is also compatible with the use of tactics and tacticals in a proof assistant.

Tac is built around a small kernel implementing elementary operations and (small-step) inferences rules. Each inference rule gives rise to a primitive tactic. Complex tactics can then be formed using tacticals such as `then`, `repeat`, etc. The successful application of a series of tactics to prove a theorem triggers the production of a proof, which can be inspected. There is no specific support for any particular datatype.

Focusing is built into the foundations of Tac: formulas are annotated with polarity information and that information is used to guide the application of logical rules. Such annotations are useful not only for the automation of inductive proof search but also for human interaction. For example, the tactic `async` simplifies a goal by repeatedly applying asynchronous rules, the common tactic `apply` actually consists of focusing on a lemma and performing a synchronous phase, and the tactic `freeze` is used to prevent induction and thereby guide automated theorem proving. Finally, we also use focusing to display proofs more concisely by showing synthetic inference rules instead of the "micro" rules.

In the following we describe our automated tactic, called `prove`. This tactic performs focused proof-search as described in Section 3, with a special treatment of computation and of the crucial deduction step of (co)induction.

4.1 Progress

We generally think of proof search as being a process composed primarily of deduction, but of course large portions of a particular proof may be given over to computation. Any implementation of proof search should recognize and exploit this distinction, not only for efficiency (computations should not involve (co)inductions or certain other deductive techniques) but also for robustness and predictability. Focusing allows us to circumscribe computations to single phases, even if that computation is non-deterministic. A useful fragment to identify is that of deterministic computation. For example, given the goal $\forall x.\ \mathit{mult}\ 10\ 10\ x \supset P\ x$ or the goal $\exists x.\ \mathit{mult}\ 10\ 10\ x \wedge P\ x$, the prover should compute the value of x immediately in a single step. The notion of *progressing unfolding* presented below allows us to do so and, in fact, to treat these examples in exactly the same way.

Definition 2 (Patterns). *A* pattern $\mathcal{C}$ *of type* $\gamma_1, \ldots, \gamma_n \rightarrow \gamma'_1, \ldots, \gamma'_m$ *is a vector of* m elementary patterns p_i*, which are themselves closed terms of type* $\gamma_1, \ldots, \gamma_n \rightarrow \gamma'_i$*. The* input arity *of the pattern is* n*, and* m *is its* output arity*. Both can be zero. When* $\boldsymbol{t}$ *is a vector of terms* $\langle t_1 : \gamma_1, \ldots, t_n : \gamma_n\rangle$*, the expression* $\mathcal{C}\boldsymbol{t}$ *denotes the vector* $\langle p_1\boldsymbol{t}, \ldots, p_n\boldsymbol{t}\rangle$*. For two vectors of terms of equal length* n*, the expression* $\boldsymbol{t} = \boldsymbol{t'}$ *denotes the formula* $t_1 = t'_1 \wedge \ldots \wedge t_n = t'_n$*.*

Definition 3 (Matcher). *Let* $\boldsymbol{\mathcal{C}}$ *be a vector* $\langle \mathcal{C}_1, \ldots, \mathcal{C}_n\rangle$ *of patterns, all of the same output arity* m*. The matcher* $M_{\boldsymbol{\mathcal{C}}}$ *is defined as the term:*

$$M_{\boldsymbol{\mathcal{C}}} \overset{def}{=} \lambda\phi_1 \ldots \lambda\phi_n \lambda x_1 \ldots \lambda x_m.(\exists \boldsymbol{y}_1.\ \boldsymbol{x} = \mathcal{C}_1\boldsymbol{y} \wedge \phi_1\boldsymbol{y}) \vee \ldots \vee (\exists \boldsymbol{y}_n.\ \boldsymbol{x} = \mathcal{C}_n\boldsymbol{y} \wedge \phi_n\boldsymbol{y})$$

Example 4. We can define *nat* from the matcher on the patterns $\mathcal{C}_1 := \langle 0\rangle$ and $\mathcal{C}_2 := \langle \lambda p.\ s\ p\rangle$ by $\mathit{nat} := \mu(\lambda N \lambda x.\ M_{\boldsymbol{\mathcal{C}}} \top N x)$. The fixed point *half* is built on the patterns $\langle 0, 0\rangle$, $\langle s\ 0, 0\rangle$, and $\langle \lambda p.\ s^2 p, \lambda p.\ s\ p\rangle$. Finally, the binary fixed point *eq* is built on the patterns $\langle 0, 0\rangle$ and $\langle \lambda p.\ s\ p, \lambda p.\ s\ p\rangle$.

In fact, matchers correspond to synchronous synthetic connectives, leaving out the least fixed points. In most fixed point definitions, the structure of matchers is used literally, but even when it is not strictly followed, it can be recovered by re-arranging the outermost layer of synchronous connectives, namely $\wedge$, $\vee$ and $\exists$. Hence, any fixed point body can be assumed of the form $(\lambda p \lambda \boldsymbol{x}.\ M_{\boldsymbol{\mathcal{C}}}(\boldsymbol{\phi} p)\boldsymbol{x})$ for some patterns $\boldsymbol{\mathcal{C}}$ and predicate operator expressions $\boldsymbol{\phi}$.

Definition 4 (Progressing unfolding). *Let* $\boldsymbol{\mathcal{C}}$ *be a vector* $\langle \mathcal{C}_1, \ldots, \mathcal{C}_n\rangle$ *of patterns of the same output arity. A least fixed point instance* $\mu(\lambda p \lambda \boldsymbol{x}.\ M_{\boldsymbol{\mathcal{C}}}(\boldsymbol{\phi} p)\boldsymbol{x})\boldsymbol{t}$ *has a progressing unfolding if* $\exists \boldsymbol{x}.\ \boldsymbol{t} = \mathcal{C}_j\boldsymbol{x}$ *holds for at most one* $j \in \{0, \ldots, n\}$*.*

Example 5. The formulas *nat* 0 and *nat* $(s\ t)$ have progressing unfoldings, for any term t, *e.g.,* $s^n 0$, x, *nil*. For any h, the formulas *half* $0\ h$, *half* $(s\ 0)\ h$ and *half* $(s^2 t)\ h$ for any t have progressing unfoldings. This is not true of *half* $(s\ x)\ h$, because the term $s\ x$ satisfies two patterns in the definition of *half*. For *eq*, the progressing unfoldings are on instances of *eq* $0\ 0$, *eq* $(s\ x)\ y$ and *eq* $x\ (s\ y)$.

This definition is critically tied to focusing: we only inspect one synchronous synthetic connective, namely a synchronous fixed point and the outermost synchronous layer of its body. After an unfolding on the left hand-side, all absurd branches of that structure are discarded during the current asynchronous phase, and at most one remains. Symmetrically, after an unfolding on the right hand-side, at most one branch remains by the end of the synchronous phase.

Note that the definition of progressing unfolding described above does not attempt to embody a notion of termination. There is no restriction placed on the structure under the pattern, hence no guarantee that the result of the unfolding is simpler than the initial fixed point. It is in fact undecidable to identify non-termination, and not even desired, since we also want to partially explore infinite computations, as we shall see in Example 6.

4.2 Discovering (Co)invariants

The construction of (co)invariants in theorem proving is an extremely difficult problem, one that has been addressed by a large number of researchers. We take a simple approach with the `prove` tactic, trying only one invariant per possible induction site. That invariant is obtained directly from the context:

$$\frac{\Sigma;\, \Gamma, S\boldsymbol{t} \vdash G \quad \boldsymbol{x}\,;\, BS\boldsymbol{x} \vdash S\boldsymbol{x}}{\Sigma;\, \Gamma, \mu B\boldsymbol{t} \vdash G} \quad \text{with } S := \lambda\boldsymbol{x}.\ \forall\Sigma.\ \boldsymbol{x} = \boldsymbol{t} \supset (\bigwedge \Gamma) \supset G.$$

With this invariant, the first premise is trivially provable, as it is essentially an instance of the identity. The second premise is where proof-search continues. This approach to induction is similar to that used in Coq, where the induction tactic always uses the current goal as the invariant. When that is not sufficient, one has to generalize the goal manually — for example by introducing a lemma.

We proceed dually for coinduction:

$$\frac{\Sigma;\, \Gamma \vdash S\boldsymbol{t} \quad \boldsymbol{x}\,;\, S\boldsymbol{x} \vdash BS\boldsymbol{x}}{\Sigma;\, \Gamma \vdash \nu B\boldsymbol{t}} \quad \text{with } S := \lambda\boldsymbol{x}.\ \exists\Sigma.\ \boldsymbol{x} = \boldsymbol{t} \wedge (\bigwedge \Gamma).$$

Such trivial (co)inductions suffice for many examples. For instance, when proving $\forall n.\ \mathit{even}\ n \supset \mathit{nat}\ n$, the generated formula $\lambda x.\ \mathit{nat}\ x$ is actually an invariant of *even*. Similarly, when proving $\forall p.\ \mathit{sim}\ p\ p$, the context does provide a coinvariant: $\lambda p_1 \lambda p_2.\ \exists p.\ p_1 = p \wedge p = p_2$, that is, $\lambda p_1 \lambda p_2.\ p_1 = p_2$.

Example 6. Consider the following theorem of μLJ: $\forall x.\ \mathit{eq}\ (s\ x)\ (s\ (s\ x)) \supset \bot$. In the asynchronous phase, $\forall$ and $\supset$ are introduced, after which the fixed point has to be treated. It can be either frozen, inducted on, or unfolded. Obviously, freezing cannot lead to a proof. Induction fails as the context does not yield a valid invariant ($\lambda m \lambda n.\ \forall x.\ m = s\ x \supset n = s\ (s\ x) \supset \bot$). The last possibility is to perform the (progressing) unfolding of the fixed point, in which case we obtain the subgoal $\mathit{eq}\ x\ (s\ x) \vdash \bot$. At this point one can quickly obtain a proof, as the context does yield an invariant: $\lambda m \lambda n.\ n = s\ m \supset \bot$. Notice that performing another progressing unfolding of *eq* would loop, producing the same subgoal.

4.3 Organization of the `prove` Tactic

In order to turn focused proof search into a practical automated tactic, several compromises must be made and these compromises will result, ultimately, in the loss of completeness. We detail below the main compromises for `prove`, namely the design of search in the asynchronous phase and how termination is ensured.

Except for freezing, all asynchronous rules are invertible – in the case of induction, this is obtained by instantiating it into a left unfolding. However, from a bottom-up proof-search point of view, this does not mean that these rules can be applied eagerly without backtracking. Namely, an asynchronous fixed point instance can be treated in a number of different ways: it may be frozen, (co)inducted on with an arbitrary (co)invariant (although in automated proof search we only use the (co)invariant generated from the context) or unfolded, perhaps in a progressing way. Our approach is to postpone these choices as much as possible, first applying all non-backtracking asynchronous rules. Moreover, we treat progressing unfoldings as non-backtracking rules, reflecting their deterministic nature. Therefore the asynchronous phase proceeds as follows: (1) apply the non-backtracking asynchronous rules, (2) try for each remaining asynchronous fixed point to either freeze, (co)induct or unfold[2], backtracking on those possibilities and coming back to Step (1) after each attempt.

The usual problem of top-down proof-search is that it may encounter infinite branches during search — in μLJ, such branches are caused by contractions, fixed point unfoldings, and (co)induction. A common way to address this issue is to bound the depth of search. This technique can be too rough, but we tame its downsides by working at the level of synthetic connectives. Moreover, we exploit the distinction between computation and deduction. First, we make use of a *deductive* bound, attached to individual sequents, which limits the number of non-progressing fixed point unfoldings, (co)inductions, and contractions in one branch. Second, we introduce a *computational* bound to limit the number of progressing unfoldings performed on a given fixed point. Computational bounds must be attached to individual formulas: since progressing unfoldings are performed eagerly, we must prevent one chain of such unfoldings from starving another. The deductive bound controls the number of critical choices made in a proof, and therefore the complexity of proof-search; increasing the bound even slightly can lead to significantly longer attempts. However, since computations are factored out of the deductive bound, a low bound typically suffices; as a default we perform iterative deepening up to a depth of 3. In contrast, the computational bound can be set much higher without affecting the cost of proof-search. This design has proved critical to the success of our tactic.

5 Comparison and Experimental Results

In order to show the strengths of our approach, we provide several examples of its successes in Figure 5 and use them to compare Tac with two established inductive theorem provers. The first group of examples come from the IWC suite (`http://www.cs.nott.ac.uk/~lad/research/challenges`) and the second one from Twelf's examples. The last group consists of interesting

[2] It might be surprising that we attain the best results by attempting (co)induction before unfolding; this is because these non-progressing unfoldings are rarely useful.

original examples leveraging Tac's specificities: *sim* illustrates that Tac deals with coinductive definitions and coinduction just as naturally as with inductive definitions; the other examples show interesting developments[3] that exploit Tac's support for generic quantification [3] and the role that the automatic tactic `prove` can play within interactive proof development. We do not provide timings since our implementation was not optimized for speed, but stress that all automated examples pass in less than a few seconds. The implementation, including all of the examples mentioned in this paper, is available at `http://slimmer.gforge.inria.fr/tac/`

Name	Description	Success
IWC 02	$append(l, l')$ of even length iff $append(l', l)$ is too	Automatic
IWC 03	$x \in l$ implies $x \in append(l, l')$	Automatic
IWC 04	$l = rotate(length(l), l)$	Guided, Lemma
IWC 06	equiv. of mutual and straight definitions of even	Automatic
IWC 07	natural numbers are even or odd	Automatic
IWC 12	verifying abstractions in model checking	Guided, Lemmas
IWC 16	whisky problem	Automatic, Lemma
plus	commutativity and associativity of *plus*	Automatic
arith	totality of many Horn programs, *e.g., half, ack*	Automatic
prop-calc	Hilbert's abstraction theorem	Automatic
reverse	involutivity of list reversal	Automatic
binarytree	antisymmetry of the subtree ordering on binary trees	Automatic
sim	reflexivity and transitivity of *sim*	Automatic
PCF	subject reduction and determinacy of typing for PCF	Manual, Lemmas
POPL-1A	transitivity of subtyping for $F^{\leq}$	Manual, Lemmas

Fig. 5. Examples of Tac proofs. "Lemma" indicates that a lemma had to be manually specified. "Automatic" indicates that `prove` derived the theorem and all lemmas. "Guided" denotes a small amount of user guidance while "Manual" denotes a mostly interactive development.

5.1 Comparison with Rewriting Based Approaches

Rewriting based approaches to inductive theorem proving are common: for example, ACL2 and the many provers that make use of rippling. These specialized foundations, together with refined heuristics, make for powerful tools, but also have some drawbacks. Notably, they do not provide as solid proof witnesses as sequent calculus, and in general cannot be related to general-purpose proof assistants such as Coq or Isabelle. Leaving aside problems that involve higher-order functions, and thus could not even be stated in our framework, the IWC challenges highlight the main differences between our approach and rewriting-based techniques.

Several challenges were infeasible in Tac because they relied too heavily on equational reasoning. When stated in a relational way in our tool, the complexity

[3] We also have a partial solution of POPLMark problem 2A, and we do not see any obstacle to its short-term completion.

of such statements increases a lot: for example, $(x+y)+z = t$ becomes $\exists i.\ x+y = i \wedge i+z = t$. This weakness is not so important in application areas such as operational semantics, where commutativity and associativity are rarely relevant and equational reasoning is little used.

On the other hand, the challenges that Tac passed show that our relatively straightforward but general approach replicates reasoning schemes that have to be implemented as special techniques in rewriting and termination-based approaches, such as case analysis, generalization, simultaneous and mutual induction schemes. Our single induction principle and limited invariant generation already embeds an interesting amount of generalization, and the nesting and interleaving of inductions gives rise to complex induction schemes (corresponding, for example, to multiset and lexicographic orderings). This is visible in some of our custom examples such as the totality of *ack* or the involutivity of list reversal.

In some cases, it is unreasonable to expect an automatic proof: the user will need to explicitly introduce generalizations or lemmas. The same goes for all provers and Tac is no exception. In our test suite, once lemmas were stated, Tac was able to prove them automatically and deduce the final theorem. In IWC challenges 4 and 12, however, some non-trivial human guidance was required in this process.

5.2 Comparison with Twelf

Twelf [13] allows writing specifications in LF, a dependently typed intuitionistic framework, and can perform proof-search in LF as well as meta-reasoning about LF specifications. Meta-reasoning is done in two ways.

First, Twelf is equipped with a number of specialized procedures for establishing whether some LF relation is total, functional, etc. That feature is well developed and widely used. We can replicate many of the established meta-theorems, but those dedicated procedures outperform our generic tactic, which is not able to re-use properties of nested fixed points. It should be noted, however, that in order to check a totality assertion, a termination order should be explicitly given, while our system infers it.

Secondly, Twelf has a more generic meta-theorem prover [13] which searches for proofs in the $\mathcal{M}_2$ meta-logic, whose objects are LF terms. This logic only deals with Π_2 statements and Twelf implements a simply structured proof-search strategy for them. The main phases of this strategy can be formulated in terms of focusing, but their general organization differs. Notably, this strategy only attempts an outermost induction, whose validity is based on a termination ordering provided by the user. In contrast, the (co)invariant generation of Tac can discover induction schemes without assistance from the user and can notably use nested inductions to discover proofs that are inaccessible to Twelf. A striking example is the involutivity of list reversal, which Tac proves without any additional lemmas thanks to nested inductions. On the other hand, Twelf can sometimes prove a theorem, *e.g.,* the totality of *half*, using a single induction when Tac would require nested inductions, since Twelf's meta-logic allows the use of an induction hypothesis on an arbitrary predecessor, while our scheme corresponds to restricting to the immediate predecessor. Finally, there is no notion of progress in Twelf's meta-theorem prover, which results in a critical need to tweak bounds.

The major strength of Twelf is not so much the architecture of its proof-search strategy, but the expressivity of the LF objects manipulated in the $\mathcal{M}_2$ logic. Twelf handles higher-order abstract syntax (HOAS) specifications easily, enabling its simple meta-theorem prover to prove important results such as type preservation and progress for programming languages. Tac supports higher-order terms and features minimal generic quantification [3] to expressively reason about such objects. Focused proof search is not affected by the introduction of these rich notions, and `prove` indeed handles them without any modification. But HOAS notably involves dealing with hypothetical contexts, for which there is no built-in support in Tac. When working with such contexts in Tac one must therefore implement them by hand, generally using lists, which tends to create artificial details. As a result, while Tac seems to be in general more powerful than Twelf's meta-theorem prover on examples not involving HOAS, it is unable to carry out automatically developments significantly dealing with hypothetical contexts. However, since Tac is an interactive proof assistant, the user can guide the proving of such theorems, still benefiting from `prove` to fill in many simple proof obligations.

6 Conclusion

We have developed a strong focusing proof system for the expressive logic μLJ, and we have shown how this important proof-theoretic result can be applied to the construction of an inductive theorem prover. The Tac prover is capable of proving automatically many non-trivial theorems, which is particularly impressive given the genericity and simplicity of its design.

There are a number of ways to enhance the current system. A general challenge with proof-search and our treatment of equality lies in the handling of logical variables in left hand-side equalities, which obviously hinders automated proof search in several examples; we are currently exploring ways to address this unusual aspect of unification. Another important challenge to address is integrating support for hypothetical contexts when reasoning about HOAS specifications. We also plan to explore the use of flexibilities in polarity assignment left in the design of focused systems for μLJ; in our experience, such choices can have an important impact on proof-search. Finally, it is crucial that we develop efficient techniques for re-using previously proved lemmas — a common problem in theorem proving.

Acknowledgments. The authors wish to thank Alexandre Viel for his help in developing Tac and the reviewers of an earlier version of this paper for their comments. This work has been supported in part by INRIA through the "Equipes Associées" Slimmer and by the NSF grant CCF-0917140. Opinions, findings, and conclusions or recommendations expressed in this papers are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

1. Andreoli, J.-M.: Logic programming with focusing proofs in linear logic. J. of Logic and Computation 2(3), 297–347 (1992)
2. Baelde, D.: A linear approach to the proof-theory of least and greatest fixed points. PhD thesis, Ecole Polytechnique (December 2008)
3. Baelde, D.: On the expressivity of minimal generic quantification. In: Abel, A., Urban, C. (eds.) International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP 2008). ENTCS, vol. 228, pp. 3–19 (2008)
4. Baelde, D., Miller, D.: Least and greatest fixed points in linear logic. In: Dershowitz, N., Voronkov, A. (eds.) LPAR 2007. LNCS (LNAI), vol. 4790, pp. 92–106. Springer, Heidelberg (2007)
5. Chaudhuri, K., Pfenning, F.: Focusing the inverse method for linear logic. In: Ong, C.-H.L. (ed.) CSL 2005. LNCS, vol. 3634, pp. 200–215. Springer, Heidelberg (2005)
6. Girard, J.-Y.: A fixpoint theorem in linear logic. An email posting to the mailing list linear@cs.stanford.edu. (February 1992)
7. Huet, G.: A unification algorithm for typed λ-calculus. Theoretical Computer Science 1, 27–57 (1975)
8. Liang, C., Miller, D.: Focusing and polarization in intuitionistic logic. In: Duparc, J., Henzinger, T.A. (eds.) CSL 2007. LNCS, vol. 4646, pp. 451–465. Springer, Heidelberg (2007)
9. McDowell, R., Miller, D.: Cut-elimination for a logic with definitions and induction. Theoretical Computer Science 232, 91–119 (2000)
10. McLaughlin, S., Pfenning, F.: Imogen: Focusing the polarized focused inverse method for intuitionistic propositional logic. In: Cervesato, I., Veith, H., Voronkov, A. (eds.) LPAR 2008. LNCS (LNAI), vol. 5330, pp. 174–181. Springer, Heidelberg (2008)
11. Momigliano, A., Tiu, A.: Induction and co-induction in sequent calculus. In: Berardi, S., Coppo, M., Damiani, F. (eds.) TYPES 2003. LNCS, vol. 3085, pp. 293–308. Springer, Heidelberg (2004)
12. Schroeder-Heister, P.: Rules of definitional reflection. In: Vardi, M. (ed.) Eighth Annual Symposium on Logic in Computer Science, June 1993, pp. 222–232. IEEE Computer Society Press, Los Alamitos (1993)
13. Schürmann, C., Pfenning, F.: Automated theorem proving in a simple meta-logic for LF. In: Kirchner, C., Kirchner, H. (eds.) CADE 1998. LNCS (LNAI), vol. 1421, pp. 286–300. Springer, Heidelberg (1998)

A Decidable Class of Nested Iterated Schemata[⋆]

Vincent Aravantinos, Ricardo Caferra, and Nicolas Peltier

Grenoble University (LIG/CNRS)

Abstract. Many problems can be specified by patterns of propositional formulae depending on a parameter, e.g. the specification of a circuit usually depends on the number of bits of its input. We define a logic whose formulae, called *iterated schemata*, allow to express such patterns. Schemata extend propositional logic with indexed propositions, e.g. P_i, $P_{\mathsf{i}+1}$, P_1 or P_n, and with generalized connectives, e.g. $\bigwedge_{\mathsf{i}=1}^{\mathsf{n}}$ or $\bigvee_{\mathsf{i}=1}^{\mathsf{n}}$ where n is an (unbound) integer variable called a *parameter*. The expressive power of iterated schemata is strictly greater than propositional logic: it is even out of the scope of first-order logic. We define a proof procedure, called DPLL⋆, that can prove that a schema is satisfiable for at least one value of its parameter, in the spirit of the DPLL procedure [9]. But proving that a schema is unsatisfiable *for every value of the parameter*, is undecidable [1] so DPLL⋆ does not terminate in general. Still, DPLL⋆ terminates for schemata of a syntactic subclass called *regularly nested*.

1 Introduction

The specification of problems in propositional logic often leads to propositional formulae that depend on a parameter: the n-queens problem depends on n, the pigeonhole problem depends on the number of considered pigeons, a circuit depends on the number of bits of its input, etc. Consider for instance a specification of a ripple-adder circuit that takes as input two n-bit vectors and computes their sum: $Adder \overset{\text{def}}{=} \bigwedge_{\mathsf{i}=1}^{\mathsf{n}} Sum_\mathsf{i} \wedge \bigwedge_{\mathsf{i}=1}^{\mathsf{n}} Carry_\mathsf{i} \wedge \neg C_1$ where n is the number of bits of the input, $Sum_\mathsf{i} \overset{\text{def}}{=} S_\mathsf{i} \Leftrightarrow (A_\mathsf{i} \oplus B_\mathsf{i}) \oplus C_\mathsf{i}$, $Carry_\mathsf{i} \overset{\text{def}}{=} C_{\mathsf{i}+1} \Leftrightarrow (A_\mathsf{i} \wedge B_\mathsf{i}) \vee (B_\mathsf{i} \wedge C_\mathsf{i}) \vee (A_\mathsf{i} \wedge C_\mathsf{i})$, $\oplus$ is the exclusive OR, $A_1, \ldots, A_\mathsf{n}$ (resp. $B_1, \ldots, B_\mathsf{n}$) are the bits of the first (resp. second) operand of the circuit, $S_1, \ldots, S_\mathsf{n}$ is the output (the **S**um), and $C_1, \ldots, C_\mathsf{n}$ are the intermediate **C**arries.

Presently, automated reasoning on such specifications requires that we give a concrete value to the parameter n. Besides the obvious loss of generality, this instantiation hides the *structure* of the initial problem which can be however a useful information when reasoning about such specifications: the structure of the proof can in many cases be *guided* by the one of the original specification. This gave us the idea to consider parameterized formulae at the object level and to design a logic to reason about them.

Notice that schemata not only arise naturally from practical problems, but also have a deep conceptual interpretation, putting bridges between logic and

⋆ This work has been partly funded by the project ASAP of the French *Agence Nationale de la Recherche* (ANR-09-BLAN-0407-01).

computation. As well as first-order logic abstracts from propositional logic via *quantification*, schemata allow to abstract via *computation*, in a complementary way. Indeed, a schema can be considered as a very specific algorithm taking as input a value for the parameter and generating a propositional formula depending on this value. So a schema can be seen as an algorithm whose codomain is the set of propositional formulae (its domain is the set of integers).

If we want to prove, e.g. that the implementation of a parameterized specification is correct, we need to prove that the corresponding schema is valid *for every value of the parameter*. As usual we actually deal with unsatisfiability: we say that a schema is *unsatisfiable* iff it is (propositionally) unsatisfiable for every value of its parameter. In [1] we introduced a first proof procedure for propositional schemata, called STAB. Notice that there is an easy way to systematically look for a counter-example: we can just enumerate all the values and check the satisfiability of the corresponding formula with a SAT solver. However this naive procedure does not terminate when the schema is unsatisfiable. On the other hand, STAB not only terminates (and much more efficiently) when the schema is satisfiable, but it can also terminate when the schema is unsatisfiable. However it still *does not terminate in general*, as we proved that the (un)satisfiability problem is undecidable for schemata [1]. Still, we proved that STAB terminates for a particular class of schemata, called *regular*, which is thus decidable.

An important restriction of the class of regular schemata is that it cannot contain nested iterations, e.g. $\bigvee_{\mathsf{i}=1}^{\mathsf{n}} \bigvee_{\mathsf{j}=1}^{\mathsf{n}} P_{\mathsf{i}} \Rightarrow Q_{\mathsf{j}}$. Nested iterations occur frequently in the specification of practical problems. We take the example of a binary multiplier which computes the product of two bit vectors $A = (A_1, \ldots, A_{\mathsf{n}})$ and $B = (B_1, \ldots, B_{\mathsf{n}})$ using the following decomposition: $A.B = A.\sum_{\mathsf{i}=1}^{\mathsf{n}} B_{\mathsf{i}}.2^{\mathsf{i}-1} = \sum_{\mathsf{i}=1}^{\mathsf{n}} A.B_{\mathsf{i}}.2^{\mathsf{i}-1}$. The circuit is mainly an iterated sum: "$S^1 = 0$" $\wedge \bigwedge_{\mathsf{i}=1}^{\mathsf{n}} (B_{\mathsf{i}} \Rightarrow Add(S^{\mathsf{i}},$ "$A.2^{\mathsf{i}-1}$"$, S^{\mathsf{i}+1})) \wedge (\neg B_{\mathsf{i}} \Rightarrow (S^{\mathsf{i}+1} \Leftrightarrow S^{\mathsf{i}}))$ where S^{i} denotes the i^{th} partial sum (hence S^{n} denotes the final result) and $Add(x, y, z)$ denotes any schema specifying a circuit which computes the sum z of x and y (for instance the previous *Adder* schema). We express "$A.2^{\mathsf{i}-1}$" by the bit vector $Sh^{\mathsf{i}} = (Sh^{\mathsf{i}}_1, \ldots, Sh^{\mathsf{i}}_{2\mathsf{n}})$ (*Sh* for *Sh*ift): $(\bigwedge_{\mathsf{j}=1}^{\mathsf{n}} Sh^1_{\mathsf{j}} \Leftrightarrow A_{\mathsf{j}}) \wedge (\bigwedge_{\mathsf{j}=\mathsf{n}}^{2\mathsf{n}} \neg Sh^1_{\mathsf{j}}) \wedge (\bigwedge_{\mathsf{i}=1}^{\mathsf{n}} \neg Sh^{\mathsf{i}}_1 \wedge \bigwedge_{\mathsf{j}=1}^{2\mathsf{n}} (Sh^{\mathsf{i}}_{\mathsf{j}+1} \Leftrightarrow Sh^{\mathsf{i}}_{\mathsf{j}}))$ and "$S^1 = 0$" by $\bigwedge_{\mathsf{i}=1}^{\mathsf{n}} \neg S^1_{\mathsf{i}}$. This schema obviously contains nested iterations.

STAB does not terminate in general on such specifications. We introduce in this paper a new proof procedure, called DPLL*, which is an extension of the DPLL procedure [9]. Extending DPLL to schemata is a complex task, because the formulae depend on an *unbounded* number of propositional variables (e.g. $\bigvee_{\mathsf{i}=1}^{\mathsf{n}} P_{\mathsf{i}}$ "contains" $P_1, \ldots, P_{\mathsf{n}}$). Furthermore, propagating the value given to an atom is not straightforward as in DPLL (in $\bigvee_{\mathsf{i}=1}^{\mathsf{n}} P_{\mathsf{i}}$ if the value of e.g. P_2 is fixed then we must propagate the assignment to P_{i} but only in the case where $\mathsf{i} = 2$). The main advantage of DPLL* over STAB is that it can operate on subformulae occurring at a deep position in the schema. This feature is essential for handling nested iterations. DPLL* is sound, complete for satisfiability detection and terminates on a class of schemata, called *regularly nested*.

The paper is organized as follows. Section 2 defines the syntax and semantics of iterated schemata. Section 3 presents the DPLL* proof procedure. Section 4 deals with the detection of cycles in proofs, which is the main tool allowing termination. Section 5 presents the class of *regularly nested schemata*, for which DPLL* terminates. Section 6 concludes the paper and overviews related works. Due to space restrictions, proofs are omitted or simply sketched (detailed proofs can be found in [2]).

2 Schemata of Propositional Formulae

Terms on the signature $\{0, s, +, -\}$ and on a countable set of integer variables $\mathcal{IV}$ are called *linear expressions*, whose set is written $\mathcal{LE}$. As usual we simply write n for $s^n(0)$ $(n > 0)$ and $n.e$ for $e + \cdots + e$ (n times). Linear expressions are considered modulo the usual properties of the arithmetic symbols (e.g. $s(0) + s(s(0)) - 0$ is assumed to be the same as $s(s(s(0)))$ and written 3). The set of first-order formulae built on $\mathcal{LE}$ and $=, <, >$ is called the set of *linear constraints* (or in short *constraints*), written $\mathcal{LC}$. If $C_1, C_2 \in \mathcal{LC}$, we write $C_1 \models C_2$ iff C_2 is a logical consequence of C_1. This relation is well known to be decidable see e.g. [7]. It is also well known that linear arithmetic admits quantifier elimination. Closed terms of Σ (i.e. integers) are denoted by n, m, i, j, k, l, linear expressions by e, f, constraints by $C, C_1, C_2, \ldots$ and integer variables by $\mathsf{n}, \mathsf{i}, \mathsf{j}$ to make clear the distinction between integer variables and expressions of the meta-language.

To make technical details simpler, and w.l.o.g., only schemata in negation normal form (n.n.f.) are considered. A linear constraint *encloses* a variable i iff there exist $e_1, e_2 \in \mathcal{LE}$ s.t. i does not occur in e_1, e_2 and $C \models e_1 \leq \mathsf{i} \wedge \mathsf{i} \leq e_2$.

Definition 1 (Schemata). *For every $k \in \mathbb{N}$, let $\mathcal{P}_k$ be a set of symbols. The set $\mathfrak{P}$ of* formula patterns *(or, for short,* patterns*) is the smallest set such that:*

- $\top, \bot \in \mathfrak{P}$.
- *If $k \in \mathbb{N}$, $P \in \mathcal{P}_k$ and $e_1, \ldots, e_k \in \mathcal{LE}$ then $P_{e_1,\ldots,e_k} \in \mathfrak{P}$ and $\neg P_{e_1,\ldots,e_k} \in \mathfrak{P}$.*
- *If $\pi_1, \pi_2 \in \mathfrak{P}$ then $\pi_1 \vee \pi_2 \in \mathfrak{P}$ and $\pi_1 \wedge \pi_2 \in \mathfrak{P}$.*
- *If $\pi \in \mathfrak{P}$, $\mathsf{i} \in \mathcal{IV}$, $C \in \mathcal{LC}$ and C encloses i then $\bigwedge_{\mathsf{i}|C} \pi \in \mathfrak{P}$ and $\bigvee_{\mathsf{i}|C} \pi \in \mathfrak{P}$.*

A schema *S is a pair (written as a conjunction) $\pi \wedge C$, where π is a pattern and C is a constraint. C is called the* constraint *of S, written C_S. π is called its* pattern*, written Π_S.*

The first three items differ from propositional logic only in the atoms which we call *indexed propositions* ($e_1, \ldots, e_k$ are called *indices*). The real novel part is the last item. Patterns of the form $\bigwedge_{\mathsf{i}|C} \pi$ or $\bigvee_{\mathsf{i}|C} \pi$ are called *iterations*. C is called the *domain* of the iteration. In [1] only domains of the form $e_1 \leq \mathsf{i} \wedge \mathsf{i} \leq e_2$ were handled, but as we shall see in Section 3, more general classes of constraints are required to define the DPLL* procedure. If C is unsatisfiable then the iteration is *empty*. Any occurrence of i in π is *bound* by the iteration. A variable occurrence which is not bound is *free*. A variable which has free occurrences in a pattern is a *parameter* of the pattern. A pattern which is just an indexed proposition $P_{e_1,\ldots,e_k}$ is called an *atom*. An atom or the negation of an atom is called a *literal*.

In [1] a schema was just a pattern, however constraints appear so often that it is more convenient to integrate them to the definition of schema. Informally, a pattern gives a "skeleton" with "holes" and the constraint specifies how the holes can be filled (this choice fits the abstract definition of schema in [8]). In the following we assume w.l.o.g. that C_S entails $\mathsf{n}_1 \geq 0 \wedge \cdots \wedge \mathsf{n}_k \geq 0$ where $\mathsf{n}_1, \ldots, \mathsf{n}_k$ are the parameters of Π_S.

For instance, $S \overset{\text{def}}{=} P_1 \wedge \bigwedge_{1 \leq \mathsf{i} \wedge \mathsf{i} \leq \mathsf{n}} (Q_\mathsf{i} \wedge \bigvee_{1 \leq \mathsf{j} \leq \mathsf{n}+1 \wedge \mathsf{i} \neq \mathsf{j}} \neg P_\mathsf{n} \vee P_{\mathsf{j}+1}) \wedge \mathsf{n} \geq 1$ is a schema. P_1, Q_i, P_n and $P_{\mathsf{j}+1}$ are indexed propositions. $\bigvee_{1 \leq \mathsf{j} \leq \mathsf{n}+1 \wedge \mathsf{i} \neq \mathsf{j}} \neg P_\mathsf{i} \vee P_{\mathsf{i}+1}$ and $\bigwedge_{1 \leq \mathsf{i} \wedge \mathsf{i} \leq \mathsf{n}} (Q_\mathsf{j} \wedge \bigvee_{1 \leq \mathsf{j} \leq \mathsf{n}+1 \wedge \mathsf{i} \neq \mathsf{j}} \neg P_\mathsf{i} \vee P_{\mathsf{i}+1})$ are the only iterations, of domains $1 \leq \mathsf{j} \leq \mathsf{n}+1 \wedge \mathsf{i} \neq \mathsf{j}$ and $1 \leq \mathsf{i} \leq \mathsf{n}$. n is the only parameter of S. Finally Π_S is $P_1 \wedge \bigwedge_{1 \leq \mathsf{i} \wedge \mathsf{i} \leq \mathsf{n}} (Q_\mathsf{i} \wedge \bigvee_{1 \leq \mathsf{j} \leq \mathsf{n}+1 \wedge \mathsf{i} \neq \mathsf{j}} \neg P_\mathsf{n} \vee P_{\mathsf{j}+1})$ and C_S is $\mathsf{n} \geq 1$. Schemata are denoted by S, S_1, $S_2 \ldots$, parameters by $\mathsf{n}, \mathsf{n}_1, \mathsf{n}_2 \ldots$, bound variables by i, j. $\Delta_{\mathsf{i}|C}\, S$ and $\nabla_{\mathsf{i}|C}\, S$ denote generic iterations (i.e. $\bigvee_{\mathsf{i}|C} S$ or $\bigwedge_{\mathsf{i}|C} S$), $\triangle$ and $\triangledown$ denote generic binary connectives ($\vee$ or $\wedge$), $\Delta_{\mathsf{i}=e_1}^{e_2}\, S$ denotes $\Delta_{\mathsf{i}|e_1 \leq \mathsf{i} \wedge \mathsf{i} \leq e_2}\, S$.

Let S be a schema and $\Delta_{\mathsf{i}_1|C_1}\, S_1$, $\ldots$, $\Delta_{\mathsf{i}_k|C_k}\, S_k$ be all the iterations occurring in S. Then $C_S \wedge C_1 \wedge \cdots \wedge C_k$ is called the *constraint context* of S, written Context(S). Notice that Context(S) loses the information on the *binding positions* of variables. This can be annoying if a variable name is bound by two different iterations or if it is both bound and free in the schema. So we assume that all schemata are such that this situation does not hold[1].

Substitutions on integer variables map integer variables to linear arithmetic expressions. We write $[e_1/\mathsf{i}_1, \ldots, e_k/\mathsf{i}_k]$ for the substitution mapping $\mathsf{i}_1, \ldots, \mathsf{i}_k$ to $e_1, \ldots, e_k$ respectively. The application of a substitution σ to an arithmetic expression e, written $e\sigma$, is defined as usual. Substitution application is naturally extended to schemata (notice that bound variables are not replaced). A substitution is *ground* iff it maps integer variables to integers (i.e. ground arithmetic expressions). An *environment* ρ of a schema S is a ground substitution mapping all parameters of S and such that $C_S\rho$ is true.

Definition 2 (Propositional Realization). *Let π be a pattern and ρ a ground substitution. The propositional formula $|\pi|_\rho$ is defined as follows:*

- $|P_{e_1,\ldots,e_k}|_\rho \overset{def}{=} P_{e_1\rho,\ldots,e_k\rho}$, $|\neg P_{e_1,\ldots,e_k}|_\rho \overset{def}{=} \neg P_{e_1\rho,\ldots,e_k\rho}$
- $|\top|_\rho \overset{def}{=} \top$, $|\bot|_\rho \overset{def}{=} \bot$, $|\pi_1 \wedge \pi_2|_\rho \overset{def}{=} |\pi_1|_\rho \wedge |\pi_2|_\rho$, $|\pi_1 \vee \pi_2|_\rho \overset{def}{=} |\pi_1|_\rho \vee |\pi_2|_\rho$
- $|\bigvee_{\mathsf{i}|C} \pi|_\rho \overset{def}{=} \bigvee \left\{ |\pi[i/\mathsf{i}]|_{\rho \cup [i/\mathsf{i}]} \middle| i \in \mathbb{Z} \text{ s.t. } C[i/\mathsf{i}]\rho \text{ is valid} \right\}$
- $|\bigwedge_{\mathsf{i}|C} \pi|_\rho \overset{def}{=} \bigwedge \left\{ |\pi[i/\mathsf{i}]|_{\rho \cup [i/\mathsf{i}]} \middle| i \in \mathbb{Z} \text{ s.t. } C[i/\mathsf{i}]\rho \text{ is valid} \right\}$

When ρ is an environment of a schema S, we define $|S|_\rho$ as $|\Pi_S|_\rho$. $|S|_\rho$ is called a propositional realization *of S.*

Notice that $\top, \bot, \vee, \wedge, \neg$ on the right-hand sides of equations have their standard *propositional* meanings. $\bigvee$ and $\bigwedge$ on the right-hand side are meta-operators denoting respectively the *propositional* formulae $\cdots \vee \cdots \vee \cdots$ and $\cdots \wedge \cdots \wedge \cdots$ or $\bot$ and $\top$ when the sets are empty. In contrast, all those symbols on the left-hand side are *pattern* connectives.

We now make precise the semantics outlined in the introduction. Propositional logic semantics are defined as usual. A *propositional interpretation* is a function mapping every propositional variable to a truth value *true* or *false*.

[1] The rule *Emptiness* of the proof system defined in Section 3 does not preserve this property, but it is easily circumvented by renaming variables.

Definition 3 (Semantics). *Let S be a schema. An* interpretation $\mathcal{I}$ of the schemata language *is a pair consisting of an environment $\rho_{\mathcal{I}}$ of S and a propositional interpretation $\mathcal{I}_p$ of $|S|_{\rho_{\mathcal{I}}}$. A schema S is* true *in $\mathcal{I}$ iff $|S|_{\rho_{\mathcal{I}}}$ is true in $\mathcal{I}_p$, in which case $\mathcal{I}$ is a* model *of S. S is* satisfiable *iff it has a model.*

Notice that an iteration $\bigvee_{\mathsf{i}|C} \pi$ (resp. $\bigwedge_{\mathsf{i}|C} \pi$) where C is equivalent to $\bot$ (i.e. the iteration is empty), is equivalent to $\bot$ (resp. $\top$).

Example 1. Let $S \overset{\text{def}}{=} P_1 \wedge \bigwedge_{\mathsf{i}=1}^{\mathsf{n}} (P_{\mathsf{i}} \Rightarrow P_{\mathsf{i}+1}) \wedge \neg P_{\mathsf{n}+1} \wedge \mathsf{n} \geq 0$ (as usual, $S_1 \Rightarrow S_2$ is a shorthand for $\neg S_1 \vee S_2$). Then $|S|_{\mathsf{n}\mapsto 0} = P_1 \wedge \neg P_1$, $|S|_{\mathsf{n}\mapsto 1} = P_1 \wedge (P_1 \Rightarrow P_2) \wedge \neg P_2$, $|S|_{\mathsf{n}\mapsto 2} = P_1 \wedge (P_1 \Rightarrow P_2) \wedge (P_2 \Rightarrow P_3) \wedge \neg P_3$, etc. S is clearly unsatisfiable.

The set of satisfiable schemata is recursively enumerable but not recursive [1]. Hence there cannot be a refutationally complete proof procedure for schemata.

The next definitions will be useful in the definition of DPLL*. Let ϕ be a propositional formula and L a (propositional) literal. We say that L occurs *positively* in ϕ, written $L \sqsubset \phi$, iff there is an occurrence of L in ϕ which is not in the scope of a negation.

Definition 4. *Let S be a schema and L a literal s.t. the parameters of L are parameters of S. We write $L \sqsubset_\Box S$ iff for every environment ρ of S, $|L|_\rho \sqsubset |S|_\rho$. We write $L \sqsubset_\Diamond S$ iff there is an environment ρ of S s.t. $|L|_\rho \sqsubset |S|_\rho$.*

Example 2. Consider S as in Example 1. We have $P_1 \sqsubset_\Box S$, $P_{\mathsf{n}+1} \sqsubset_\Box S$, $P_2 \not\sqsubset_\Box S$. However $P_2 \sqsubset_\Diamond S$ and $P_2 \sqsubset_\Box (S \wedge \mathsf{n} \geq 1)$. Finally $P_0 \not\sqsubset_\Diamond S$ and $P_{\mathsf{n}+2} \not\sqsubset_\Diamond S$.

Suppose L has the form $P_{e_1,\dots,e_k}$ (resp. $\neg P_{e_1,\dots,e_k}$). For a literal $L' \sqsubset S$ of indices $f_1, \dots, f_k$, $\phi_L(L')$ denotes the formula $\exists \mathsf{i}_1 \dots \mathsf{i}_n (C_{\mathsf{i}_1} \wedge \dots \wedge C_{\mathsf{i}_n} \wedge e_1 = f_1 \wedge \dots \wedge e_k = f_k)$ where $\mathsf{i}_1, \dots, \mathsf{i}_n$ are all the bound variables of S occurring in $f_1, \dots, f_k$ and $C_{\mathsf{i}_1}, \dots, C_{\mathsf{i}_n}$ are the domains of the iterations binding $\mathsf{i}_1, \dots, \mathsf{i}_n$. $\phi_L(S)$ denotes $\bigvee \{\phi_L(P_{f_1,\dots,f_k}) \mid P_{f_1,\dots,f_k} \sqsubset S\}$ (resp. $\bigvee \{\phi_L(\neg P_{f_1,\dots,f_k}) \mid \neg P_{f_1,\dots,f_k} \sqsubset S\}$).

Proposition 1. *$L \sqsubset_\Box S$ iff $\forall \mathsf{n}_1, \dots, \mathsf{n}_l (C_S \Rightarrow \phi_L(S))$ is valid, where $\mathsf{n}_1 \dots \mathsf{n}_l$ are all the parameters of S. $L \sqsubset_\Diamond S$ iff $\exists \mathsf{n}_1, \dots, \mathsf{n}_l (C_S \wedge \phi_L(S))$ is valid.*

Consider e.g. S as in Example 1. For any e, $P_e \sqsubset_\Box S$ iff $\forall \mathsf{n} (\mathsf{n} \geq 0) \Rightarrow [e = 1 \vee \exists \mathsf{i}(1 \leq \mathsf{i} \wedge \mathsf{i} \leq \mathsf{n} \wedge e = \mathsf{i}) \vee \exists \mathsf{i}(1 \leq \mathsf{i} \wedge \mathsf{i} \leq \mathsf{n} \wedge e = \mathsf{i} + 1) \vee e = \mathsf{n} + 1]$ is valid. By decidability of linear arithmetic, both $\sqsubset_\Box$ and $\sqsubset_\Diamond$ are decidable. Besides, it is easy to compute the set $\mathcal{L}(S) \overset{\text{def}}{=} \{L \mid L \sqsubset_\Box S\}$ for a schema S.

3 A Proof Procedure: DPLL*

We provide now a set of (sound) deduction rules (in the spirit of the Davis-Putnam-Logemann-Loveland procedure for propositional logic [9]) complete w.r.t. satisfiability (we know that it is not possible to get refutational completeness). Compared to other proof procedures [1] DPLL* allows to rewrite subformulae occurring at deep positions inside a schema — in particular occurring in the scope of iterated connectives: this is crucial to handle nested iterations. DPLL* is a tableaux-like procedure: rules are given to construct a tree whose root is the formula that one wants to refute. The formula is refuted iff all the branches are contradictory.

As usual with tableaux related methods, the aim of branching is to browse the possible interpretations of the schema. As a schema interpretation assigns a truth value to each atom and a number to each parameter, there are two branching rules: one for atoms, called *Propositional splitting* (this rule assigns a value to propositional variables, as the splitting rule in DPLL), and one for parameters, called *Constraint splitting*. However *Constraint splitting* does not give a value to the parameters, but rather restricts their values by refining the constraint of the schema (i.e. C_S), e.g. the parameter can be either greater or lower than a given integer, leading to two branches in the tableaux. Naturally, in order to analyze a schema, one has to investigate the contents of iterations. So a relevant constraint to use for the branching is the one that states the emptiness of some iteration. In the branch where the iteration is empty, we can replace it by its neutral element (i.e. $\top$ for $\bigwedge$ and $\bot$ for $\bigvee$), which is done by *Constraint splitting* (this may also entails the emptiness of some other iterations, and thus their replacement by their neutral elements too, this is handled by *Algebraic simplification*). Then in the branch where the iteration is not empty, we can unfold the iteration: this is done by the *Unfolding* rule.

Iterations might occur in the scope of other iterations. Thus their domains might depend on variables bound by the outer iterations. *Constraint splitting* is of no help in this case, indeed it makes a branching only according to the values of the *parameter*: bound variables are out of its scope. Hence we define the rule *Emptiness* that can make a "deep" branching, i.e. a branching not in the tree, but in the schema itself: it "separates" an iteration into two distinct ones, depending on the constraint stating the emptiness of the inner iteration, e.g. $\bigvee_{\mathsf{i}=1}^{\mathsf{n}} \bigvee_{\mathsf{j}=3}^{\mathsf{i}} P_{\mathsf{i}} \wedge \mathsf{n} \geq 2$ is replaced by $\bigvee_{\mathsf{i}=3}^{\mathsf{n}} \bigvee_{\mathsf{j}=3}^{\mathsf{i}} P_{\mathsf{i}} \vee \bigvee_{\mathsf{i}=1}^{2} \bot \wedge \mathsf{n} \geq 2$.

Constraint splitting strongly affects the application of *Propositional splitting*. Indeed *Propositional splitting* only applies on atoms occurring in *all* instances of the schema (formalized by Definition 4), and we saw in Example 2 that this depends on the constraint. Once an atom A is given the value true (resp. false) we can replace it by $\top$ (resp. $\bot$). But this is not as simple as in the propositional case as A may occur in a realization of the schema without occurring in the schema itself (e.g. P_1 in $\bigwedge_{\mathsf{i}=1}^{\mathsf{n}} P_{\mathsf{i}}$ $(\star)$), so we cannot just replace it by $\top$. The simplification is performed by the rule *Expansion* which wraps the indexed propositions that are more general than the considered atom (P_{i} in $(\star)$) with an iteration whose domain is a disunification constraint stating that the proposition is distinct from the atom (for $(\star)$ this gives: $\bigwedge_{\mathsf{i}=1}^{\mathsf{n}} \bigwedge_{\mathsf{j}|\mathsf{i}\neq 1 \wedge \mathsf{j}=0} P_{\mathsf{i}}$). The introduced iteration is very specific because the bound variable always equals 0 (this variable is not used but we assign it 0 to satisfy the condition in Definition 1 that it has to be enclosed by the domain). Whereas usual iterations shall be considered as "for loops", this one is an "if then else". It makes sense when *Emptiness* or *Constraint splitting* is applied: *if* the condition holds (i.e. if the wrapped proposition differs from the atom) *then* the contents of the iteration hold (i.e. we keep the indexed proposition) *else* the iteration is empty (i.e. we replace it by its neutral element). In $(\star)$, *Emptiness* applies: $\bigwedge_{\mathsf{i}|1\leq \mathsf{i}\leq \mathsf{n} \wedge \exists \mathsf{j}(\mathsf{i}\neq 1 \wedge \mathsf{j}=0)} \bigwedge_{\mathsf{j}|\mathsf{i}\neq 1 \wedge \mathsf{j}=0} P_{\mathsf{i}} \wedge \bigwedge_{\mathsf{i}|1\leq \mathsf{i}\leq \mathsf{n} \wedge \forall \mathsf{j}(\mathsf{i}=1 \vee \mathsf{j}\neq 0)} \top$

(of course the domains can be simplified to allow reader-friendly presentation). Then *Algebraic simplification* gives: $\bigwedge_{\mathsf{i}|1\leq \mathsf{i}\leq \mathsf{n}\wedge\exists \mathsf{j}(\mathsf{i}\neq 1\wedge \mathsf{j}=0)} P_{\mathsf{i}}$, i.e. $\bigwedge_{\mathsf{i}=2}^{\mathsf{n}} P_{\mathsf{i}}$, as expected. This process may seem cumbersome, but it is actually a uniform and powerful way of propagating constraints about nested iterations along the schema.

Finally we may know that an iteration is empty without knowing which value of the bound variable satisfies the domain constraint. Then the *Interval splitting* rule adds some constraints on the involved expressions to ensure this knowledge.

We now define DPLL* formally.

Definition 5 (Tableau). *A* tableau *is a tree $\mathcal{T}$ s.t. each node α in $\mathcal{T}$ is labeled with a pair $(S_{\mathcal{T}}(\alpha), \mathcal{L}_{\mathcal{T}}(\alpha))$ containing a schema and a finite set of literals.*

If α is the root of the tree then $\mathcal{L}_{\mathcal{T}}(\alpha) = \emptyset$ and $S_{\mathcal{T}}(\alpha)$ is called the *root schema*. The transitive closure of the child-parent relation is written $\prec$. For a set of literals $\mathcal{L}$, $\bigwedge_{\mathcal{L}}$ denotes the pattern $\bigwedge_{L\in\mathcal{L}} L$.

As usual a tableau is generated from another tableau by applying extension rules written $\frac{P}{C}$ (resp. $\frac{P}{C_1|C_2}$) where P is the premise and C (resp. C_1, C_2) the conclusion(s). Let α be a leaf of a tree $\mathcal{T}$, if the label of α matches the premise then we can *extend* the tableau by adding to α a child (resp. two children) labeled with $C\sigma$ (resp. $C_1\sigma$ and $C_2\sigma$), where σ is the matching substitution. A leaf α is *closed* iff $\Pi_{S_{\mathcal{T}}(\alpha)}$ is equal to $\bot$ or $C_{S_{\mathcal{T}}(\alpha)}$ is unsatisfiable.

When used in a premise, $S[\pi]$ means that the schema π occurs in S; when used in a conclusion, $S[\pi']$ denotes S in which π has been substituted with π'.

Definition 6 (DPLL* rules). *The* extension rules *are:*

– Propositional splitting.

$$\frac{(S,\mathcal{L})}{(S,\mathcal{L}\cup P_{e_1,\dots,e_k}) \;\Big|\; (S,\mathcal{L}\cup\neg P_{e_1,\dots,e_k})}$$

if either $P_{e_1,\dots,e_k} \sqsubset_{\Box} S$ or $\neg P_{e_1,\dots,e_k} \sqsubset_{\Box} S$, and neither $P_{e_1,\dots,e_k} \sqsubset_{\Diamond} \bigwedge_{\mathcal{L}} \wedge C_S$ nor $\neg P_{e_1,\dots,e_k} \sqsubset_{\Diamond} \bigwedge_{\mathcal{L}} \wedge C_S$.

– Constraint splitting. *For $(\Delta,\varepsilon) \in \{(\bigwedge,\top),(\bigvee,\bot)\}$:*

$$\frac{(S[\Delta_{\mathsf{i}|C}\,\pi],\mathcal{L})}{(S[\Delta_{\mathsf{i}|C}\,\pi]\wedge\exists\mathsf{i}C,\mathcal{L}) \;\Big|\; (S[\varepsilon]\wedge\forall\mathsf{i}\neg C,\mathcal{L})}$$

if $C_S \wedge \forall\mathsf{i}\neg C$ is satisfiable and free variables of C other than i *are parameters.*

– *Rewriting:*

$$\frac{(S_1,\mathcal{L})}{(S_2,\mathcal{L})}$$

where $C_{S_2} = C_{S_1}$ and $\Pi_{S_1} \rightarrow \Pi_{S_2}$ by the following rewrite system:

• Algebraic simplification. *For every pattern π:*

$$\neg\top\rightarrow\bot \quad \pi\wedge\top\rightarrow\pi \quad \pi\wedge\bot\rightarrow\bot \quad \bigwedge_{\mathsf{i}|C}\top\rightarrow\top \quad \pi\wedge\pi\rightarrow\pi$$

$$\neg\bot\rightarrow\top \quad \pi\vee\top\rightarrow\top \quad \pi\vee\bot\rightarrow\pi \quad \bigvee_{\mathsf{i}|C}\bot\rightarrow\bot \quad \pi\vee\pi\rightarrow\pi$$

if Context$(S_1) \wedge \exists\mathsf{i}C$ *is unsatisfiable:* $\bigwedge_{\mathsf{i}|C}\pi\rightarrow\top \quad \bigvee_{\mathsf{i}|C}\pi\rightarrow\bot$

if Context$(S_1) \Rightarrow \exists\mathsf{i}C$ *is valid and π does not contain* i*:* $\Delta_{\mathsf{i}|C}\,\pi\rightarrow\pi$

- Unfolding. *For* $(\triangle, \Delta) \in \{(\wedge, \bigwedge), (\vee, \bigvee)\}$*:*

$$\Delta_{\mathsf{i}|C} \pi \to \pi[e/\mathsf{i}] \triangle \Delta_{\mathsf{i}|C \wedge \mathsf{i} \neq e} \pi \qquad \textit{if } \mathrm{Context}(S_1) \Rightarrow C[e/\mathsf{i}] \textit{ is valid}$$

  e can be chosen arbitrarily[2].
- Emptiness. *For* $(\triangle, \Delta) \in \{(\wedge; \bigwedge), (\vee; \bigvee)\}$, $(\nabla, \varepsilon) \in \{(\bigwedge; \top), (\bigvee; \bot)\}$*:*

$$\Delta_{\mathsf{i}|C}(\pi[\nabla_{\mathsf{i}'|C'} \pi']) \to \Delta_{\mathsf{i}|C \wedge \exists \mathsf{i}' C'}(\pi[\nabla_{\mathsf{i}'|C'} \pi']) \triangle \Delta_{\mathsf{i}|C \wedge \forall \mathsf{i}' \neg C'}(\pi[\varepsilon])$$

  if $\mathrm{Context}(S_1) \wedge \forall \mathsf{i}' \neg C'$ *is satisfiable and* i *occurs freely in* C'*.*
- Expansion.

$$P_{e_1,\dots,e_k} \to \bigwedge_{\mathsf{i}|(e_1 \neq f_1 \vee \cdots \vee e_k \neq f_k) \wedge \mathsf{i}=0} P_{e_1,\dots,e_k} \qquad \textit{if } P_{f_1,\dots,f_k} \in \mathcal{L}$$

$$P_{e_1,\dots,e_k} \to \bigvee_{\mathsf{i}|(e_1 \neq f_1 \vee \cdots \vee e_k \neq f_k) \wedge \mathsf{i}=0} P_{e_1,\dots,e_k} \qquad \textit{if } \neg P_{f_1,\dots,f_k} \in \mathcal{L}$$

  if $\mathrm{Context}(S_1) \wedge e_1 = f_1 \wedge \cdots \wedge e_k = f_k$ *is satisfiable.* i *is a fresh variable.*

– Interval splitting. *For* $k, l \in \mathbb{N}$, $\Delta \in \{\bigwedge, \bigvee\}$, $\lhd \in \{<, \leq, \geq, >\}$*:*

$$\frac{(S[\Delta_{\mathsf{i}|C \wedge k.\mathsf{i} \lhd e_1 \wedge l.\mathsf{i} \lhd e_2} \pi], \mathcal{L})}{(S[\Delta_{\mathsf{i}|C \wedge k.\mathsf{i} \lhd e_1} \pi] \wedge l.e_1 \lhd k.e_2, \mathcal{L}) \mid (S[\Delta_{\mathsf{i}|C \wedge l.\mathsf{i} \lhd e_2} \pi] \wedge l.e_1 \not\lhd k.e_2, \mathcal{L})}$$

  if every free variable of C *is either* i *or a parameter, all variables of* e_1, e_2 *are parameters and* $k > 0$, $l > 0$*.*

A *derivation* is a (possibly infinite) sequence of tableaux $(\mathcal{T}_i)_{i \in I}$ s.t. I is either $\mathbb{N}$ or $[0..k]$ for some $k \geq 0$, and s.t. for all $i > 0$, $\mathcal{T}_i$ is obtained from $\mathcal{T}_{i-1}$ by applying a rule. A derivation is *fair* iff no leaf can be indefinitely "freezed" i.e. either there is $i \in I$ s.t. $\mathcal{T}_i$ contains an irreducible, not closed, leaf or if for all $i \in I$ and every leaf α in $\mathcal{T}_i$ there is $j \geq i$ s.t. a rule is applied on α in $\mathcal{T}_j$.

Theorem 1 (Soundness and Completeness w.r.t. Satisfiability). *Consider a fair derivation* $(\mathcal{T}_i)_{i \in I}$*.* $\mathcal{T}_0$ *is satisfiable iff there is* $i \in I$ *s.t.* $\mathcal{T}_i$ *contains an irreducible and not closed leaf.*

The proof can be found in [2], which also contains a DPLL* tableau example.

4 Looping Detection

The above extension rules do not terminate in general, but this is not surprising as the satisfiability problem is undecidable [1]. Non-termination comes from the fact that iterations can be infinitely unfolded (consider e.g. $\bigvee_{\mathsf{i}=1}^{\mathsf{n}} P_{\mathsf{i}} \wedge \neg P_{\mathsf{i}}$), thus leading to infinitely many new schemata. However, often, newly obtained

[2] e.g. in Section 5 we choose the maximal integer fulfilling the desired property.

schemata have already been seen (up to some relation that remains to be defined) i.e. the procedure is *looping* (e.g. $\bigvee_{\mathsf{i}=1}^{\mathsf{n}} P_{\mathsf{i}} \wedge \neg P_{\mathsf{i}}$ generates $\bigvee_{\mathsf{i}=1}^{\mathsf{n}-1} P_{\mathsf{i}} \wedge \neg P_{\mathsf{i}}$, then $\bigvee_{\mathsf{i}=1}^{\mathsf{n}-2} P_{\mathsf{i}} \wedge \neg P_{\mathsf{i}}$, then $\bigvee_{\mathsf{i}=1}^{\mathsf{n}-k} P_{\mathsf{i}} \wedge \neg P_{\mathsf{i}}$, which are all equal up to a shift of n). This is actually an algorithmic interpretation of a proof by mathematical induction. We now define precisely the notion of looping. We start with a very general definition:

Definition 7 (Looping). *Let S_1, S_2 be two schemata having the same parameters $\mathsf{n}_1, \dots, \mathsf{n}_k$, we say that S_1* loops *on S_2 iff for every model $\mathcal{I}$ of S_1 there is a model $\mathcal{J}$ of S_2 s.t. $\rho_{\mathcal{J}}(\mathsf{n}_j) < \rho_{\mathcal{I}}(\mathsf{n}_j)$ for some $j \in 1..k$ and $\rho_{\mathcal{J}}(\mathsf{n}_l) \le \rho_{\mathcal{I}}(\mathsf{n}_l)$ for every $l \neq j$. The induced relation among schemata is called the* looping relation.

For instance, if $S_1 = \bigvee_{\mathsf{i}=1}^{\mathsf{n}-1} P_{\mathsf{i}}$ and $S_2 = \bigvee_{\mathsf{i}=1}^{\mathsf{n}} P_{\mathsf{i}}$, take any model $\mathcal{I}$ of S_1 and construct a model $\mathcal{J}$ of S_2 as follows: $\rho_{\mathcal{J}}(\mathsf{n}) \stackrel{\text{def}}{=} \rho_{\mathcal{I}}(\mathsf{n}) - 1$, and for every $i \in 1..\rho_{\mathcal{J}}(\mathsf{n})$, $\mathcal{J}_p(P_i) \stackrel{\text{def}}{=} \mathcal{I}_p(P_i)$. It is easy to see that $\mathcal{J}$ is indeed a model. Similarly $S_1 = \bigvee_{\mathsf{i}=2}^{\mathsf{n}} P_{\mathsf{i}}$ loops on $S_2 = \bigvee_{\mathsf{i}=1}^{\mathsf{n}} P_{\mathsf{i}}$: take any model $\mathcal{I}$ of S_1 and build a model $\mathcal{J}$ of S_2 as follows: $\rho_{\mathcal{J}}(\mathsf{n}) \stackrel{\text{def}}{=} \rho_{\mathcal{I}}(\mathsf{n}) - 1$, and for every $i \in 1..\rho_{\mathcal{J}}(\mathsf{n})$, $\mathcal{J}_p(P_i) \stackrel{\text{def}}{=} \mathcal{I}_p(P_{i+1})$. In both cases, it is intuitive that if DPLL* encounters S_1 after having seen S_2, then it will behave similarly as for S_2, hence the name of "looping". Notice that looping also applies e.g. with $\bigvee_{\mathsf{i}=1}^{\mathsf{n}-1} P_{\mathsf{i}}$ and $\bigvee_{\mathsf{i}=1}^{\mathsf{n}} Q_{\mathsf{i}}$, i.e. the name of symbols does not matter. Looping is undecidable (e.g. if $S_2 = \bot$ then S_1 loops on S_2 iff S_1 is unsatisfiable). It is trivially transitive.

Definition 8. *Let α, β be nodes in a tableau $\mathcal{T}$. $S\mathcal{L}_{\mathcal{T}}(\alpha)$ denotes the schema $S_{\mathcal{T}}(\alpha) \wedge \bigwedge_{\mathcal{L}_{\mathcal{T}}(\alpha)}$. Then β* loops *on α iff $S\mathcal{L}_{\mathcal{T}}(\beta)$ loops on $S\mathcal{L}_{\mathcal{T}}(\alpha)$.*

The *Looping* rule closes a leaf that loops on some existing node of the tableau. From now on, DPLL* denotes the extension rules, plus the *Looping* rule. Theorem 1 still holds [2]. The notion of loop introduced in Definition 8 is undecidable, thus, in practice, we use decidable refinements of looping. Termination proofs work by showing that the set of schemata which are generated by the procedure is finite up to some looping refinement. We make precise this notion:

Definition 9. *A binary relation between schemata is a* looping refinement *iff it is a subset of the looping relation. Let $\mathcal{S}$ be a set of schemata and $\rhd$ a looping refinement. A schema $S \in \mathcal{S}$ is a $\rhd$-*maximal companion *w.r.t. $\mathcal{S}$ iff there is no $S' \in \mathcal{S}$ s.t. $S \rhd S'$. The set of all $\rhd$-maximal companions w.r.t. $\mathcal{S}$ is written $\mathcal{S}/\rhd$. If it is finite then we say that $\mathcal{S}$ is* finite up to $\rhd$.

4.1 Equality Up to a Shift

We now present perhaps the simplest refinement of looping. A *shiftable* is a schema, a linear constraint, a pattern, a linear expression or a tuple of those. The refinement is defined on shiftables (and not only on schemata) in order to handle those objects in a uniform way. This is useful in the termination proof of Section 5 (and even more in [2]).

Definition 10. *Let s, s' be shiftables and n a variable. If $s' = s[\mathsf{n} - k/\mathsf{n}]$ for some $k > 0$, then s' is* equal to s up to a shift of k on n*, written $s' \rightrightarrows^{\mathsf{n}} s$ (or $s' \rightrightarrows_k^{\mathsf{n}} s$ when we want to make k explicit).*

For instance, $\bigvee_{\mathsf{i}=1}^{\mathsf{n}-1} P_{\mathsf{i}} \rightrightarrows_1^{\mathsf{n}} \bigvee_{\mathsf{i}=1}^{\mathsf{n}} P_{\mathsf{i}}$ but $\bigvee_{\mathsf{i}=2}^{\mathsf{n}} P_{\mathsf{i}} \not\rightrightarrows^{\mathsf{n}} \bigvee_{\mathsf{i}=1}^{\mathsf{n}} P_{\mathsf{i}}$. As examples of shiftables which are not schemata : $\mathsf{n}-2 \rightrightarrows_2^{\mathsf{n}} \mathsf{n}$ and $(\forall i \cdot \mathsf{n} \geq i) \rightrightarrows_1^{\mathsf{n}} (\forall i \cdot \mathsf{n}+1 \geq i)$.

The fact that we use syntactic equality makes the refinement less powerful but simpler to implement and easier to reason with. It is easy to check that $\rightrightarrows^{\mathsf{n}}$ is a looping refinement when restricted to schemata having n as a parameter. Furthermore $\rightrightarrows^{\mathsf{n}}$ is transitive but neither reflexive, nor irreflexive. It is irreflexive for shiftables containing n, and reflexive for shiftables not containing n.

We focus now on sets which are *finite up to equality up to a shift*, in short "$\rightrightarrows^{\mathsf{n}}$-finite": termination proofs go by showing that the set of all schemata possibly generated by DPLL* is $\rightrightarrows^{\mathsf{n}}$-finite, thus ensuring that the *Looping* rule will eventually apply. To prove such results we need to reason by induction on the structure of a schema. To do this properly we need closure properties for $\rightrightarrows^{\mathsf{n}}$-finite sets i.e. if we know that two sets are $\rightrightarrows^{\mathsf{n}}$-finite, we would like to be able to combine them and preserve the $\rightrightarrows^{\mathsf{n}}$-finite property. This is generally not possible, e.g. for two $\rightrightarrows^{\mathsf{n}}$-finite sets of shiftables $\mathcal{S}_1$ and $\mathcal{S}_2$, the set $\mathcal{S}_1 \times \mathcal{S}_2$ (remember that shiftables are closed by tuple construction) is generally *not* $\rightrightarrows^{\mathsf{n}}$-finite. For instance take $\mathcal{S}_1 = \{P_{\mathsf{n}}, P_{\mathsf{n}-1}, P_{\mathsf{n}-2}, , \dots\}$ ($\mathcal{S}_1$ is $\rightrightarrows^{\mathsf{n}}$-finite with $\mathcal{S}_1/\rightrightarrows^{\mathsf{n}} = \{P_{\mathsf{n}}\}$) and $\mathcal{S}_2 = \{P_{\mathsf{n}}\}$ (which is finite and thus $\rightrightarrows^{\mathsf{n}}$-finite). Then $\{(P_{\mathsf{n}}, P_{\mathsf{n}}), (P_{\mathsf{n}-1}, P_{\mathsf{n}}), (P_{\mathsf{n}-2}, P_{\mathsf{n}}), \dots\}$ is not $\rightrightarrows^{\mathsf{n}}$-finite: indeed for every $i \in \mathbb{N}$, $(P_{\mathsf{n}-i}, P_{\mathsf{n}})$ is a maximal companion in $\mathcal{S}_1 \times \mathcal{S}_2$, there is thus an infinite set of maximal companions. Consequently $\mathcal{S}_1 \times \mathcal{S}_2$ is not $\rightrightarrows^{\mathsf{n}}$-finite. Hence we have to restrict our closure operators.

Definition 11. *Let* n *be a variable. A shiftable* s *is* translated *w.r.t.* n *iff for every linear expression* e *occurring in* s *and containing* n *there is* $k \in \mathbb{Z}$ *s.t.* $e = \mathsf{n}+k$ *(i.e. neither* $k.\mathsf{n}$ *nor* $\mathsf{n}+\mathsf{i}$ *are allowed, where* $k \in \mathbb{Z}$*,* $k \neq 0$ *and* $\mathsf{i} \in \mathcal{IV}$*).*

Assume that s *is translated w.r.t.* n*. The* deviation *of* s *w.r.t.* n*, written* $\delta(s)$*, is defined as* $\delta(s) \overset{def}{=} \max\{k_1 - k_2 \mid k_1, k_2 \in \mathbb{Z}, \mathsf{n}+k_1, \mathsf{n}+k_2 \text{ occur in } s\}$*.* $\delta(s) \overset{def}{=} 0$ *if* s *does not contain* n*. Let* $k \in \mathbb{N}$*, we write* $\mathfrak{B}_k \overset{def}{=} \{s \mid \delta(s) \leq k\}$*.*

Theorem 2. *Let* $\mathcal{S}_1$ *and* $\mathcal{S}_2$ *be two sets of shiftables translated w.r.t. a variable* n*. If* $\mathcal{S}_1$ *and* $\mathcal{S}_2$ *are* $\rightrightarrows^{\mathsf{n}}$*-finite then, for any* $k \in \mathbb{N}$*,* $\mathcal{S}_1 \times \mathcal{S}_2 \cap \mathfrak{B}_k$ *is* $\rightrightarrows^{\mathsf{n}}$*-finite.*

Consequently (assume all shiftables are translated w.r.t. n): $\{S_1 \vartriangle S_2 \mid S_1 \in \mathcal{S}_1, S_2 \in \mathcal{S}_2\} \cap \mathfrak{B}_k$, where $\vartriangle \in \{\wedge, \vee\}$, is $\rightrightarrows^{\mathsf{n}}$-finite if $\mathcal{S}_1$ and $\mathcal{S}_2$ are $\rightrightarrows^{\mathsf{n}}$-finite; and $\{(\bigwedge_{i|C} \Pi_S) \wedge C_S \mid S \in \mathcal{S}, C \in \mathcal{C}\} \cap \mathfrak{B}_k$ is $\rightrightarrows^{\mathsf{n}}$-finite if $\mathcal{S}$ and $\mathcal{C}$ are $\rightrightarrows^{\mathsf{n}}$-finite.

4.2 Refinement Extensions

We now define simple extensions that allow better detection. Consider for example the schema S defined in Example 1. Using DPLL* there is a branch which contains: $S' \overset{\text{def}}{=} P_1 \wedge \bigwedge_{\mathsf{i}=1}^{\mathsf{n}-1}(P_{\mathsf{i}} \Rightarrow P_{\mathsf{i}+1}) \wedge \neg P_{\mathsf{n}} \wedge \neg P_{\mathsf{n}+1} \wedge \mathsf{n} \geq 0 \wedge \mathsf{n}-1 \geq 0$. S' loops on S but S' is not equal to S up to a shift. However $\neg P_{\mathsf{n}+1}$ is pure in S' (i.e. $P_{\mathsf{n}+1} \not\sqsubseteq S'$) so $\neg P_{\mathsf{n}+1}$ may be evaluated to true. Therefore we obtain $P_1 \wedge \bigwedge_{\mathsf{i}=1}^{\mathsf{n}-1}(P_{\mathsf{i}} \Rightarrow P_{\mathsf{i}+1}) \wedge \neg P_{\mathsf{n}} \wedge \mathsf{n} \geq 0 \wedge \mathsf{n}-1 \geq 0$, i.e. $S[\mathsf{n}-1/\mathsf{n}] \wedge \mathsf{n} \geq 0$. But $\mathsf{n}-1 \geq 0$ entails $\mathsf{n} \geq 0$ so we can remove $\mathsf{n} \geq 0$ and finally get $S[\mathsf{n}-1/\mathsf{n}]$. We now generalise this example, thereby introducing two new looping refinements: the *pure literal* extension and the *redundant constraint* extension.

As usual a literal L is *(propositionally) pure* in a formula ϕ iff its complement does not occur positively in ϕ. The pure literal rule is standard in propositional theorem proving: it consists in evaluating a literal L to true in a formula ϕ if L is pure in ϕ. It is well-known that this operation preserves satisfiability. The notion of pure literal has to be adapted to schemata. The conditions on L must be strengthened in order to take iterations into account. For instance, if $L = P_{\mathsf{n}}$ and $S = \bigvee_{\mathsf{i}=1}^{2\mathsf{n}} \neg P_{\mathsf{i}}$ then L is not pure in S since $\neg P_{\mathsf{i}}$ is the complement of L for $\mathsf{i} = \mathsf{n}$ (and $1 \leq \mathsf{n} \leq 2\mathsf{n}$). However $P_{2\mathsf{n}+1}$ is pure in S (since $2\mathsf{n}+1 \notin [1..2\mathsf{n}]$).

Definition 12. *A literal L is* pure *in a schema S iff for every environment ρ of S, $|L|_\rho$ is propositionally pure in $|S|_\rho$.*

It is easily seen that L is pure in S iff $L^c \not\sqsubseteq_\diamond S$, thus by decidability of $\sqsubseteq_\diamond$, it is decidable to determine if a literal is pure or not.

The substitution of an indexed proposition $P_{e_1,\dots,e_k}$ by a pattern π' in a pattern π, written $\pi[\pi'/P_{e_1,\dots,e_k}]$, is defined as follows: $P_{e_1,\dots,e_k}[\pi'/P_{e_1,\dots,e_k}] \stackrel{\text{def}}{=} \pi'$; $Q_{f_1,\dots,f_k}[\pi'/P_{e_1,\dots,e_k}] \stackrel{\text{def}}{=} Q_{f_1,\dots,f_k}$ if $P \neq Q$ or $f_i \neq e_i$ for some $i \in [1..k]$; $(\pi_1 \vartriangle \pi_2)[\pi'/P_{e_1,\dots,e_k}] \stackrel{\text{def}}{=} \pi_1[\pi'/P_{e_1,\dots,e_k}] \vartriangle \pi_2[\pi'/P_{e_1,\dots,e_k}]$; $(\Delta_{\mathsf{i}|C}\,\pi)[\pi'/P_{e_1,\dots,e_k}] \stackrel{\text{def}}{=} \Delta_{\mathsf{i}|C}\,\pi[\pi'/P_{e_1,\dots,e_k}]$. For a schema S, we set $S[\pi'/P_{e_1,\dots,e_k}] \stackrel{\text{def}}{=} \Pi_S[\pi'/P_{e_1,\dots,e_k}]$.

It is easy to show that for a literal L which is pure in a schema S, if S (resp. $S[\top/L]$) has a model $\mathcal{I}$ then $S[\top/L]$ (resp. S) has a model $\mathcal{J}$ s.t. $\rho_{\mathcal{I}}(\mathsf{n}) = \rho_{\mathcal{J}}(\mathsf{n})$ for every parameter n of S. A schema S in which all pure literals have been substituted with $\top$ is written purified(S).

Definition 13. *Let $\rhd$ be a looping refinement. We call* the pure extension *of $\rhd$ the relation $\rhd'$: $S_1 \rhd' S_2 \Leftrightarrow$* purified$(S_1) \rhd$ purified(S_2)*.*

$\rhd'$ is easily proved to be a looping refinement.

Finally we describe the redundant constraint extension:

Definition 14. *Any normal form of a schema S by the following rewrite rules:*

$$\begin{array}{rcll} C_1 \wedge \dots \wedge C_k & \to & C_1 \wedge \dots \wedge C_{k-1} & \textit{if } \{C_1,\dots,C_{k-1}\} \models C_k \\ C & \to & \bot & \textit{if } C \textit{ is unsatisfiable} \end{array}$$

is called a constraint-irreducible *schema of S.*

By decidability of satisfiability in linear arithmetic, it is easy to compute a constraint-irreducible schema of S.

Definition 15. *Let $\rhd$ be a looping refinement. We call* the constraint-irreducible extension *of $\rhd$ the relation $\rhd'$ s.t. for all S_1, S_2, $S_1 \rhd' S_2$ iff there exists S_1' (resp. S_2') a constraint-irreducible schema of S_1 (resp. S_2) s.t. $S_1' \rhd S_2'$.*

Once again $\rhd'$ is easily proved to be a looping refinement.

5 A Decidable Class: Regularly Nested Schemata

Definition 16 (Regularly Nested Schema). *An iteration $\Delta_{\mathsf{i}|C}\,\pi$ is* framed *iff there are two expressions e_1, e_2 s.t. $C \Leftrightarrow e_1 \leq \mathsf{i} \wedge \mathsf{i} \leq e_2$. $[e_1..e_2]$ is called the* frame *of the iteration. A schema S is:*

– Monadic *iff all indexed propositions occurring in S have only one index.*
– Framed *iff all iterations occurring in it are framed.*
– Aligned on $[e_1..e_2]$ *iff it is framed and all iterations have the same frame* $[e_1..e_2]$.
– Translated *iff it is translated (Def. 11) w.r.t. every variable occurring in it.*
– Regularly Nested *iff it has a unique parameter* n*, it is monadic, translated and aligned on* $[k..\mathsf{n}-l]$ *for some* $k, l \in \mathbb{Z}$.

Notice that regularly nested schemata allow the nesting of iterations. But they are too weak to express the binary multiplier presented in the Introduction (since only monadic propositions are considered). However $\bigwedge_{\mathsf{i}=1}^{\mathsf{n}} \bigvee_{\mathsf{j}=1}^{\mathsf{n}} (P_{\mathsf{i}} \Rightarrow Q_{\mathsf{j}+1}) \wedge \bigwedge_{\mathsf{i}=1}^{\mathsf{n}} \neg Q_{\mathsf{i}+1} \wedge \bigvee_{\mathsf{i}=1}^{\mathsf{n}} P_{\mathsf{i}}$, for instance, is a regularly nested schema: it is obviously monadic; all its iterations have the same domain $1 \leq \mathsf{i} \wedge \mathsf{i} \leq \mathsf{n}$ so they are clearly framed and aligned on $[1..\mathsf{n}]$ (so we have indeed the required alignment for $k = 1$ and $l = 0$); it is translated as no multiplication nor addition between variables occur inside the schema; finally it has of course only one parameter: n. Also, though not needing the nesting of iterations the ripple-adder presented in the introduction is a regularly nested schema. So is a carry look-ahead adder or an arithmetic comparison circuit, or actually any circuit performing linear arithmetic computations on bit-vectors of arbitrary size. One can also express the inclusion of a finite automaton into another one, and similarly for alternating automata.

We divide *Constraint splitting* into two disjoint rules: *framed-Constraint splitting* (resp. *non framed-Constraint splitting*) denotes *Constraint splitting* with the restriction that $\Delta_{\mathsf{i}|C}\,\pi$ (following the notations of the rule) is framed (resp. not framed). We consider the following strategy $\mathfrak{S}$ for applying the extension rules on a regularly nested schema:

1. First only framed-*Constraint splitting* is applied until irreducibility.
2. Then all other rules except *Unfolding* are applied until irreducibility with the restriction that *Expansion* rewrites P_{e_1} iff e_1 contains no variable other than the parameter of the schema.
3. Finally only *Unfolding* is applied until irreducibility, with the restriction that if the unfolded iteration is framed then e (in the definition of *Unfolding*) is the upper bound of the frame. We then go back to 1.

For the *Looping* rule we use equality up to a shift with its pure and constraint-irreducible extensions. It is easy to prove that $\mathfrak{S}$ preserves completeness.

Theorem 3. $\mathfrak{S}$ *terminates on every regularly nested schema.*

Proof. (Sketch) We show that the set $\{S\mathcal{L}_{\mathcal{T}}(\alpha) \mid \alpha \text{ is a node of } \mathcal{T}\}$ — i.e. the set of schemata generated all along the procedure — is finite up to the constraint-irreducible and pure extensions of equality up to a shift. As $S\mathcal{L}_{\mathcal{T}}(\alpha) = \Pi_{S_{\mathcal{T}}(\alpha)} \wedge C_{S_{\mathcal{T}}(\alpha)} \wedge \bigwedge_{\mathcal{L}_{\mathcal{T}}(\alpha)}$, this set is equal to $\{\Pi_{S_{\mathcal{T}}(\alpha)} \wedge C_{S_{\mathcal{T}}(\alpha)} \wedge \bigwedge_{\mathcal{L}_{\mathcal{T}}(\alpha)} \mid \alpha \text{ is a node of } \mathcal{T}\}$. So the task can approximately be divided into four: prove that the set of patterns is finite up to a shift, prove that the set of constraints is finite up to a shift, prove that the set of partial interpretations is finite up to a shift and combine the three results thanks to Theorem 2.

Among those tasks, the hardest is the first one, because it requires an induction on the structure of $\Pi_{S_{\mathcal{T}}(\alpha)}$. For this induction to be achieved properly we need to "trace" the evolution under $\mathfrak{S}$ of every subpattern of $\Pi_{S_{\mathcal{T}}(\alpha)}$. A subpattern can be uniquely identified by its position. So we extend DPLL* into T-DPLL* (for **T**raced DPLL*), by adding to the pair $(S_{\mathcal{T}}(\alpha), \mathcal{L}_{\mathcal{T}}(\alpha))$ labelling nodes in DPLL* a third component containing a set of positions of $\Pi_{S_{\mathcal{T}}(\alpha)}$. Along the execution of the procedure, this subpattern may be moved, duplicated, deleted, some context may be added around it, some of its subpatterns may be modified. Despite all those modifications, we are able to follow the subpattern thanks to the set of positions in the labels.

As usual, a position is a finite sequence of natural numbers, ϵ denotes the empty sequence, $s_1.s_2$ denotes the concatenation of s_1 and s_2 and $\leq$ denotes the prefix ordering. The *positions* of a pattern π are defined as follows: ϵ is a position in π; if p is a position in π then $1.p$ is a position in $\neg\pi$, $\bigwedge_{\mathsf{i}|C} \pi$ and $\bigvee_{\mathsf{i}|C} \pi$; let $i \in \{1, 2\}$, if p is a position in π_i then $i.p$ is a position in $\pi_1 \vee \pi_2$ and $\pi_1 \wedge \pi_2$.

For two sequences s_1, s_2 s.t. s_2 is a prefix of s_1, $s_2 \setminus s_1$ is the sequence s.t. $s_2.(s_2 \setminus s_1) = s_1$. In particular for two positions p_1, p_2 s.t. p_2 is a prefix of p_1, $p_2 \setminus p_1$ can be seen as the position *relatively to* p_2 of the subterm in position p_1.

Definition 17 (T-DPLL*). *A* T-DPLL* tableau $\mathcal{T}$ *is the same as a* DPLL* *tableau except that a node α is labeled with a triple $(S_{\mathcal{T}}(\alpha), \mathcal{L}_{\mathcal{T}}(\alpha), \mathcal{P}_{\mathcal{T}}(\alpha))$ where $\mathcal{P}_{\mathcal{T}}(\alpha)$ is a set of positions in $\Pi_{S_{\mathcal{T}}(\alpha)}$.* T-DPLL* *keeps the behavior of* DPLL* *for $S_{\mathcal{T}}(\alpha)$ and $\mathcal{L}_{\mathcal{T}}(\alpha)$, we only describe the additional behavior for $\mathcal{P}_{\mathcal{T}}(\alpha)$ as follows: $p \to p_1, \ldots, p_k$ means that p is deleted and $p_1, \ldots, p_k$ are added to $\mathcal{P}_{\mathcal{T}}(\alpha)$.*

- *Splitting rules and the* Expansion *rewrite rule leave $\mathcal{P}_{\mathcal{T}}(\alpha)$ as is.*
- *Rewrite rules. We write q for the position of the subpattern of Π_S which is rewritten. We omit* Emptiness *as it never applies.*
  - Algebraic simplification*. For $p > q$:*

    $p \to q.(1 \setminus (q \setminus p))$ — *for rules where π occurs on both sides of the rewrite (following the notations of Definition 6), and if p is the position of a subpattern of π*

    $p \to \emptyset$ — *otherwise*

  - Unfolding*. For $p > q$:* $\quad p \to\ q.1.(q \setminus p),\ q.2.1.(q \setminus p)$

*(*Interval splitting *and* Emptiness *are untouched as they never apply when the input schema is regularly nested, see [2])*

From now on, $\mathcal{T}$ is a T-DPLL* tableau whose root schema is regularly nested of parameter n and of alignment $[k..\mathsf{n} - l]$ for some $k, l \in \mathbb{Z}$.

The set $\{S\mathcal{L}_{\mathcal{T}}(\alpha) \mid \alpha$ is a node of $\mathcal{T}\}$ is actually *not* finite up to a shift. We have to restrict ourselves to a particular kind of nodes, called *alignment nodes.* Then $\{S\mathcal{L}_{\mathcal{T}}(\alpha) \mid \alpha$ is an *alignment* node of $\mathcal{T}\}$ will indeed be finite up to a shift. A node of $\mathcal{T}$ is an *alignment node* iff it is irreducible by step 2 of $\mathfrak{S}$. It is easy to check that every alignment node is framed and aligned on $[k..\mathsf{n} - l - j]$ for some $j > 0$. Thus every alignment node is regularly nested. Furthermore, by irreducility w.r.t. the *Propositional splitting* and *Expansion* rules, the parameter

n only occurs in the domain of the iteration (otherwise the corresponding literal would be added into $\mathcal{L}$). It can be shown (see [2] for details) that each of the steps 1-3 terminates. Thus every branch b containing a node α is either finite or contains an alignment node $\beta \prec \alpha$, i.e. *an alignment node is always reached.*

Once this preliminary work is done, we can tackle the four aforementioned tasks. Finiteness (up to a shift) of $\{\Pi_{S_{\mathcal{T}}(\alpha)} \mid \alpha$ is an alignment node of $\mathcal{T}\}$ is proved by induction: T-DPLL* enables to reason by induction and Theorem 2 enables to make use of the inductive hypotheses to prove each inductive case. Finiteness of $\{C_{S_{\mathcal{T}}(\alpha)} \mid \alpha$ is an alignment node of $\mathcal{T}\}$ is proved thanks to the constraint-irreducible extension of equality up to a shift. Finiteness of $\{\bigwedge \mathcal{L}_{\mathcal{T}}(\alpha) \mid \alpha$ is an alignment node of $\mathcal{T}\}$ is proved thanks to the pure literal extension of equality up to a shift. Finally Theorem 2 enables to combine the three results.

See [2] for the detailed proof. □

6 Conclusion

We have presented a proof procedure, called DPLL*, for reasoning with propositional formula schemata. The main originality of our calculus is that the inference rules may apply at a deep position in a formula, a feature that is essential for handling nested iterations. A looping mechanism is introduced to improve the termination behavior. We defined an abstract notion of looping which is very general, then instantiated this relation into a more concrete version that is decidable, but still powerful enough to ensure termination in many cases.

We identified a class of schemata, called regularly nested schemata, for which DPLL* always terminates. This class is much more expressive than the class of regular schemata handled in [1]. The principle of the termination proof is (to the best of our knowledge) original: we investigate how a given subformula is affected by the application of extension rules on the "global" schema. This is done by defining a "traced" version of the calculus in which additional information is provided concerning the evolution of a specific subformula (or set of subformulae, since a formula may be duplicated). This also required a thorough investigation of the properties of the looping relation. We think these ideas could be reused to prove termination of other calculi, sharing common features with DPLL* (namely calculi that operate at deep levels inside a formula and that allow cyclic proofs).

We do not know of any similar work in automated deduction. Schemata have been studied in logic (see e.g. [8,12]) but our approach is different from these (essentially proof theoretical) works both in the particular kind of targeted schemata and in the emphasis on the automation of the proposed calculi. However one can find similarities with other works.

Iterations are closely related to fixed-point constructions, in particular in the (modal) μ-calculus[3] [4] (with $\bigwedge_{\mathsf{i}=1}^{\mathsf{n}} \phi$ translated into something like $\mu X.\phi \wedge X$). However the semantics are very different: that of iterated schemata is restricted to *finite models* (since every parameter is mapped to an integer, the obtained interpretation is finite), whereas models of the μ-calculus may be infinite. Hence

[3] In which many temporal logics e.g. CTL, LTL, and CTL* can be translated.

the involved logic is very different from ours and actually simpler from a theoretical point of view: the μ-calculus admits complete proof procedures and is decidable, whereas schemata enjoy none of those properties. The relation between schemata and the μ-calculus is analogous to the relation between finite model theory [10] and classical first-order logic. The detailed comparison of all those formalisms is worth investigating but out of the scope of the present work.

One can also translate schemata into first-order logic by turning the iterations into (bounded) quantifications i.e. $\bigwedge_{\mathsf{i}=1}^{\mathsf{n}} \phi$ (resp. $\bigvee_{\mathsf{i}=1}^{\mathsf{n}} \phi$) becomes $\forall \mathsf{i}(1 \leq \mathsf{i} \leq \mathsf{n} \Rightarrow \phi)$ (resp. $\exists \mathsf{i}(1 \leq \mathsf{i} \leq \mathsf{n} \wedge \phi)$). This translation is completed by quantifying universally on the parameters and by axiomatizing first-order linear arithmetic. Then one can use inductive theorem provers [6,5]. For such provers, the only decidability results that we know of are due to Kapur et al. (see e.g. [11]) but they do not apply to the formulae obtained by the above translation: they all deal with formulae of the form $\forall \boldsymbol{x}.\phi$ where ϕ is *quantifier-free*. However another translation is possible that is better suited to those results: inductive theorem provers are designed to prove results about recursively defined functions (e.g. $\forall x \cdot double(x) = x + x$ where *double* is defined by $double(0) \overset{\text{def}}{=} 0$ and $double(s(x)) = s(s(double(x)))$), and it happens that we can define iterations with recursive functions e.g. $\bigvee_{\mathsf{i}=1}^{\mathsf{n}} P_{\mathsf{i}}$ can be defined by $f(0) \overset{\text{def}}{=} \perp$ and $f(s(\mathsf{i})) \overset{\text{def}}{=} P(\mathsf{i}) \vee f(\mathsf{i})$, where P is an *uninterpreted* symbol. And indeed ACL2 [3] is able to prove some *very simple* schemata translated this way. However such formulae still does not fit the shape required for the results in [11], due to the presence of uninterpreted symbol functions in the recursive definitions.

Future work includes the implementation of the DPLL* calculus and the investigation of its practical performances. It would also be interesting to extend the termination result in Section 5 to non monadic schemata. Extension of the previous results to first-order logic is also planned.

References

1. Aravantinos, V., Caferra, R., Peltier, N.: A Schemata Calculus For Propositional Logic. In: Giese, M., Waaler, A. (eds.) TABLEAUX 2009. LNCS, vol. 5607, pp. 32–46. Springer, Heidelberg (2009)
2. Aravantinos, V., Caferra, R., Peltier, N.: A Decidable Class of Nested Iterated Schemata (extended version). Technical report, Laboratory of Informatics of Grenoble (2010), `http://arxiv.org/abs/1001.4251`
3. Boyer, R.S., Moore, J.S.: A Computational Logic. Academic Press, New York (1979)
4. Bradfield, J., Stirling, C.: Modal Mu-Calculi. In: Blackburn, P., van Benthem, J.F.A.K., Wolter, F. (eds.) Handbook of Modal Logic. Studies in Logic and Practical Reasoning, vol. 3. Elsevier Science Inc., New York (2007)
5. Bundy, A.: The Automation of Proof by Mathematical Induction. In: [13], pp. 845–911
6. Comon, H.: Inductionless induction. In: [13], ch. 14
7. Cooper, D.: Theorem proving in arithmetic without multiplication. In: Meltzer, B., Michie, D. (eds.) Machine Intelligence, vol. 7. Edinburgh University Press (1972)

8. Corcoran, J.: Schemata: the concept of schema in the history of logic. The Bulletin of Symbolic Logic 12(2), 219–240 (2006)
9. Davis, M., Logemann, G., Loveland, D.: A Machine Program for Theorem Proving. Communication of the ACM 5, 394–397 (1962)
10. Fagin, R.: Finite-Model Theory - A Personal Perspective. Theoretical Computer Science 116, 3–31 (1993)
11. Kapur, D., Subramaniam, M.: Extending Decision Procedures with Induction Schemes. In: McAllester, D. (ed.) CADE 2000. LNCS, vol. 1831, pp. 324–345. Springer, Heidelberg (2000)
12. Orevkov, V.P.: Proof schemata in Hilbert-type axiomatic theories. Journal of Mathematical Sciences 55(2), 1610–1620 (1991)
13. Robinson, J.A., Voronkov, A. (eds.): Handbook of Automated Reasoning (in 2 volumes). Elsevier, Amsterdam (2001)

RegSTAB: A SAT Solver for Propositional Schemata[⋆]

Vincent Aravantinos, Ricardo Caferra, and Nicolas Peltier

Grenoble University (LIG/CNRS)

Abstract. We describe the system RegStab (for **reg**ular **s**chemata **tab**leau) that solves the satisfiability problem for a class of propositional schemata. Our formalism extends propositional logic by considering *indexed propositions* (such as $P_1, P_{\mathtt{i}}, P_{\mathtt{j}+1}, \ldots$) and *iterated connectives* (e.g. $\bigvee_{i=\mathtt{i}}^{\mathtt{n}} \phi$). The indices and bounds are linear arithmetic expressions (possibly containing variables, interpreted as integers). Our system allows one to check the satisfiability of sequences of formulae such as $(\bigvee_{\mathtt{i}=1}^{\mathtt{n}} P_{\mathtt{i}}) \wedge (\bigwedge_{\mathtt{i}=1}^{\mathtt{n}} \neg P_{\mathtt{i}})$.

1 Introduction

Propositional logic is widely used in many domains such as artificial intelligence, automated reasoning, program verification or circuit design. Very efficient systems (SAT solvers) have been developed for deciding satisfiability (see e.g. [1] for a survey). Frequently, propositional specifications are parameterized by a natural number that encodes for instance the size of the data. As a typical example, consider the following formula, implementing a n-bits adder. $\oplus$ denotes the exclusive OR, A, B denote the operands, S denotes the result, C the vector of carries and X_i the i-th bit of X.

$$\neg C_1 \wedge \bigwedge_{\mathtt{i}=1}^{\mathtt{n}} (C_{\mathtt{i}+1} \Leftrightarrow [(A_{\mathtt{i}} \wedge B_{\mathtt{i}}) \vee (A_{\mathtt{i}} \wedge C_{\mathtt{i}}) \vee (B_{\mathtt{i}} \wedge C_{\mathtt{i}})])$$
$$\bigwedge_{\mathtt{i}=1}^{\mathtt{n}} (S_{\mathtt{i}} \Leftrightarrow (A_{\mathtt{i}} \oplus B_{\mathtt{i}}) \oplus C_{\mathtt{i}})$$

Various properties can be verified on this specification, for instance the following formula checks that $A + 0 = A$: $(\bigwedge_{\mathtt{i}=1}^{\mathtt{n}} \neg B_{\mathtt{i}}) \Rightarrow \bigwedge_{\mathtt{i}=1}^{\mathtt{n}} (A_{\mathtt{i}} \Leftrightarrow S_{\mathtt{i}})$.

The usual way of handling such a specification is by giving a value to $\mathtt{n}$ (e.g. $\mathtt{n} = 8, 16, 64, \ldots$). Then the iterated connectives $\bigwedge_{\mathtt{i}=1}^{\mathtt{n}}$ may be expanded into standard ones ($\wedge$), yielding a propositional formula that can be handled by a SAT solver (if the value of $\mathtt{n}$ is not too big). In this paper we describe a system called RegStab (standing for **reg**ular **s**chemata **tab**leau) that is able to reason directly on such schemata to prove that they are valid or unsatisfiable *for every value of* $\mathtt{n} \in \mathbb{N}$ (or $\mathtt{n} \in \mathbb{Z}$). This problem obviously requires some form of inductive reasoning (on $\mathtt{n}$). It is actually undecidable if arbitrary indices and iterations are considered [2] but it is decidable for a (reasonably expressive) subclass of schemata, that we call *regular*. The prover uses a tableaux-based decision procedure. It is fully automatic (no human guidance is needed) and has been optimized to be reasonably efficient.

[⋆] This work has been partly funded by the project ASAP of the French *Agence Nationale de la Recherche* (ANR-09-BLAN-0407-01).

2 Regular Schemata: Formal Definitions

For the sake of conciseness, we only give the formal definitions for the specific class of schemata handled by RegStab (general definitions can be found in [2]).

Let $\mathcal{P}$ be a set of *propositional symbols* and let $\mathcal{V}$ be a set of *arithmetic variables.*

Definition 1. *An* atom *is an expression of the form P_a or $P_{\mathtt{n}+a}$ where $P \in \mathcal{P}$, $\mathtt{n} \in \mathcal{V}$ and $a \in \mathbb{Z}$. The set of atoms of the form P_a (resp. $P_{\mathtt{n}+a}$) is denoted by $\mathfrak{A}$ (resp. $\mathfrak{A}(\mathtt{n})$). An* $\mathtt{n}$-iteration body *is a propositional formula built on the set of atoms $\mathfrak{A}(\mathtt{n})$ and on the set of logical connectives $\vee, \wedge, \Rightarrow, \Leftrightarrow, \neg$.*

Let $\mathtt{n} \in \mathcal{V}$, $a, b \in \mathbb{Z}$. The set of regular schemata of bounds $a, \mathtt{n} - b$ *(or simply* schemata*) is written $\mathfrak{R}_a^{\mathtt{n}-b}$ and inductively defined as follows:*

- $\mathfrak{A} \cup \mathfrak{A}(\mathtt{n}) \cup \{\top, \bot\} \subseteq \mathfrak{R}_a^{\mathtt{n}-b}$.
- *If $\phi, \psi \in \mathfrak{R}_a^{\mathtt{n}-b}$ then $(\phi \vee \psi), (\phi \wedge \psi), (\phi \Rightarrow \psi), (\phi \Leftrightarrow \psi) \in \mathfrak{R}_a^{\mathtt{n}-b}$.*
- *If $\phi \in \mathfrak{R}_a^{\mathtt{n}-b}$ then $(\neg\phi) \in \mathfrak{R}_a^{\mathtt{n}-b}$.*
- *If $\mathtt{i} \in \mathcal{V}$, ϕ is an $\mathtt{i}$-iteration body then $\bigvee_{\mathtt{i}=a}^{\mathtt{n}-b} \phi, \bigwedge_{\mathtt{i}=a}^{\mathtt{n}-b} \phi \in \mathfrak{R}_a^{\mathtt{n}-b}$.*

$\mathtt{n}$ *is called the* parameter *of the schema.*

Example 1. Let $\phi_1 \overset{\text{def}}{=} [P_{\mathtt{n}+1} \wedge \neg P_0]$, $\phi_2 \overset{\text{def}}{=} [P_1 \wedge (\bigwedge_{\mathtt{i}=1}^{\mathtt{n}} P_{\mathtt{i}} \Leftrightarrow P_{\mathtt{i}+2}) \wedge \neg P_{\mathtt{n}}]$ and $\phi_3 \overset{\text{def}}{=} [\neg P_{\mathtt{n}+2} \wedge \bigvee_{\mathtt{i}=0}^{\mathtt{n}+1} P_{\mathtt{i}+1} \wedge \bigwedge_{\mathtt{j}=0}^{\mathtt{n}+1} \neg P_{\mathtt{j}}]$. $\phi_1 \in \mathfrak{R}_a^{\mathtt{n}-b}$ for every $a, b \in \mathbb{Z}$. ϕ_1 is an $\mathtt{n}$-iteration body. $\phi_2 \in \mathfrak{R}_1^{\mathtt{n}-0}$ and $\phi_3 \in \mathfrak{R}_0^{\mathtt{n}-(-1)}$. The adder defined in the introduction belongs to $\mathfrak{R}_1^{n-0}$.

$\bigvee_{\mathtt{i}=1}^{\mathtt{n}} P_{\mathtt{i}} \wedge \bigwedge_{\mathtt{i}=0}^{\mathtt{n}-1} \neg P_{\mathtt{i}+1}$ and $\bigvee_{\mathtt{i}=1}^{\mathtt{n}} P_{\mathtt{i}+\mathtt{n}}$ are not regular schemata: in the former case the two iterations have distinct bounds and in the latter the index is not of the form $\mathtt{i} + a$ for $a \in \mathbb{Z}$.

We forbid arithmetic expressions of the form e.g. $\mathtt{n} + \mathtt{i}$ or $\mathtt{i} + \mathtt{i}$. This restriction is essential for decidability. However, the fact that the iterations have the same bounds is not so important since in many cases this property can be easily ensured by "unfolding" the iterated connectives.

Schemata are interpreted by giving a value to the parameter n and mapping every ground index proposition P_k to a truth value. Formally:

Definition 2. *An* interpretation I *of a formula $\phi \in \mathfrak{R}_a^{\mathtt{n}-b}$ is a function mapping $\mathtt{n}$ to an integer and mapping every indexed proposition P_c where $P \in \mathcal{P}$, $c \in \mathbb{Z}$ to a truth value T or F.*

Let $c \in \mathbb{Z}$. If ϕ is an $\mathtt{i}$-iteration body then $\phi[c]$ denotes the formula obtained by replacing every atom $P_{\mathtt{i}+d}$ occurring in ϕ by P_{c+d}. If $\phi \in \mathfrak{R}_a^{\mathtt{n}-b}$ then $\phi[c]$ denotes the formula obtained by replacing every atom $P_{\mathtt{n}+d}$ by P_{c+d} and every $\mathtt{i}$-iteration $\bigvee_{\mathtt{i}=a}^{\mathtt{n}-b} \psi$ (resp. $\bigwedge_{\mathtt{i}=a}^{\mathtt{n}-b} \psi$) by the (standard) disjunction (resp. conjunction) $\psi[a] \vee \ldots \vee \psi[c-b]$ (resp. $\psi[a] \wedge \ldots \wedge \psi[c-b]$). Notice that the disjunction or conjunction is empty if $c - b < a$, in which case it is equivalent to $\bot$ or to $\top$, by convention. If $c - b = a$ then the disjunction or conjunction is equivalent to $\psi[a]$. Obviously $\phi[c]$ is a standard propositional formula built on $\mathfrak{A}$, thus every interpretation I of ϕ is also an interpretation of $\phi[c]$ (except that the mapping of $\mathtt{n}$ is useless for $\phi[c]$). Propositional formulae are interpreted as usual.

Definition 3. *Let $\phi \in \mathfrak{R}_a^{\mathtt{n}-b}$. An interpretation I is a* model *of ϕ (written $I \models \phi$) iff $I \models \phi[I(\mathtt{n})]$. ϕ is* satisfiable *iff it has a model.*

Consider the formulae ϕ_1, ϕ_2, ϕ_3 defined in Example 1. ϕ_1 and ϕ_2 are satisfiable but not valid (if $I \models \phi_1$ then $I(n) \neq -1$ and if $I \models \phi_2$ then $I(n)$ must be even). ϕ_3 is unsatisfiable.

In many cases, it is useful to add arithmetic constraints to the schema, for instance $\bigwedge_{\mathtt{i}=1}^{\mathtt{n}} (P_{\mathtt{i}} \wedge \neg P_{\mathtt{i}})$ is valid if $\mathtt{n} < 1$ and unsatisfiable if $\mathtt{n} \geq 1$. Thus we actually consider pairs $\phi \mid \mathcal{C}$ where ϕ is a schema and $\mathcal{C}$ is an arithmetic atom of the form $\mathtt{n} \leq a$, $\mathtt{n} \geq a$, $\mathtt{n} > a$, $\mathtt{n} < a$ or $\mathtt{n} = a$ where $a \in \mathbb{Z}$. We write $I \models \phi \mid \mathcal{C}$ iff $I \models \phi$ and $I \models \mathcal{C}$ (arithmetic constraints are interpreted as usual).

An obvious method to check whether a (constrained) schema is satisfiable or not is to enumerate all possible interpretations. This naive procedure does not terminate when the schema is unsatisfiable. The reader can refer to [2,3,4] for informal comparisons with existing languages and discussions.

3 Proof Procedure

The proof procedure (called Stab) used by RegStab is introduced in [2]. Stab handles *general* schemata (not only regular ones). It is sound and complete for satisfiability (i.e. it always returns a model if the considered schema is satisfiable) and it terminates on every regular schema (in double exponential time [3]). It is based on an extension of the tableaux method for propositional logic.

For the sake of efficiency RegStab does not implement the full procedure Stab: it only handles the regular schemata defined in section 2, using a strategy specifically tuned for this class (sketched in Section 5). This restriction permits to define an always terminating, fully automatic program. Several optimizations have been incorporated in order to improve efficiency.

We briefly review the proof procedure. The nodes in the tableaux are labeled by sets of *formulae* that are either schemata or arithmetic constraints. All the schemata have the same parameter $\mathtt{n}$ that is also the only (free) variable occurring in the constraints. The notion of models and satisfiability extend straightforwardly to sets of formulae. Initially, the root is labeled by $\{\phi, \mathcal{C}\}$ where $\phi \mid \mathcal{C}$ is the considered constrained schemata (as defined in Section 2). The tableaux are constructed by *extension rules* of the form $\frac{\phi_1,\ldots,\phi_a}{\psi_1|\ldots|\psi_b}$ meaning that a leaf node labeled by $\Phi \cup \{\phi_1, \ldots, \phi_a\}$ can be extended by b new child nodes labeled by $\Phi \cup \{\psi_c\}$ (where $1 \leq c \leq b$).

To describe these rules, we need to introduce two additional definitions. A schema ψ occurs *positively* in ϕ iff it occurs in the scope of an even number of negation symbols (taking into account "hidden" negations, i.e. P occurs positively in $Q \Rightarrow P$ or in $\neg P \Rightarrow Q$ but not in $P \Rightarrow Q$). A literal l is *pure* in a set of formulae Φ of parameter $\mathtt{n}$ iff for every model I of Φ and for every formula $\phi \in \Phi$, the complement of l does not occur positively in $\phi[I(\mathtt{n})]$.

If Φ, Ψ are two sets of formulae of parameter $\mathtt{n}$, we say that Φ *loops on* Ψ iff for every model I of Φ there exists a model J of Ψ such that $J(\mathtt{n}) < I(\mathtt{n})$. A leaf in the tree is *looping* iff its label loops on the label of the previous node in the branch. The previous relation is actually undecidable, thus a stronger criterion is used: we check that Ψ can be obtained from Φ by replacing the parameter $\mathtt{n}$ by $\mathtt{n} - a$, where $a > 0$[1]. The *extension rules* of STAB are defined as follows.

- The usual rules of propositional tableaux: $\dfrac{\phi \wedge \psi}{\phi \quad \psi}$ $\quad$ $\dfrac{\phi \vee \psi}{\phi \mid \psi}$
- Rules proper to schemata ("iteration rules")[2]:

$$\frac{\bigwedge_{\mathtt{i}=a}^{\mathtt{n}-b} \phi}{\begin{array}{c} \mathtt{n} - b \geq a \\ \bigwedge_{\mathtt{i}=a}^{\mathtt{n}-b-1} \phi \wedge \phi[\mathtt{n} - b] \end{array} \;\Bigg|\; \mathtt{n} - b < a} \qquad \frac{\bigvee_{\mathtt{i}=a}^{\mathtt{n}-b} \phi}{\begin{array}{c} \mathtt{n} - b \geq a \\ \bigvee_{\mathtt{i}=a}^{\mathtt{n}-b-1} \phi \vee \phi[\mathtt{n} - b] \end{array}}$$

- The closure rule adds disunification constraints to the branch (using the same idea as in e.g. [5] or [6]): $\dfrac{P_a \quad \neg P_b}{P_a, \neg P_b, a \neq b}$
- The purity rule removes pure literals from the branch[3] and the looping rule closes looping leaves.

A branch is *closed* iff it contains $\bot$, if the constraints in it are unsatisfiable (modulo arithmetic) or if it is looping. Intuitively, the previous rules try to construct in a symbolic way all possible tableaux starting from the root, depending on the different possible values for $\mathtt{n}$. This is done in a lazy way, i.e. the instantiation of $\mathtt{n}$ is postponed until it is necessary to apply a rule (e.g. to decide whether an iteration is empty or not or whether a branch contains two complementary literals). The looping rule allows one to discard infinite derivations by detecting cycles in the proof search.

4 The System

RegSTAB is available at `http://regstab.forge.ocamlcore.org/`. It is written in OCaml and was successfully tested on MacOSX (10.5), Win32 (Windows XP SP3) and GNU Linux (Ubuntu 9.04) x86 platforms. The system comes with a (short) manual including installation and usage instructions and a description of the input syntax. Functions can be defined to make the input file more readable (see `Sum(i)` and `Carry(i)` below). Here is an input file for the adder example in the Introduction.

```
// A+0=A
let Sum(i)      := S_i <-> (A_i (+) B_i (+) C_i) in
let Carry(i)    := C_i+1 <-> (A_i /\ B_i \/ C_i /\ A_i \/ C_i /\ B_i) in
let Adder       := /\i=1..n (Sum(i) /\ Carry(i)) /\ ~C_1 in
```

[1] A much more comprehensive criterion is proposed in [2] but the previous one is simpler to implement and sufficient to ensure termination.

[2] The right branch in the conclusion of the Iterated $\wedge$-rule is required, e.g. to detect that $\bigwedge_{i=1}^{n} \bot$ is satisfiable with $n = 0$.

[3] Only top-level literals are removed: simplifying positive occurrences is not straightforward since these occurrences are not necessarily explicit (e.g. p_2 "occurs" in $\bigvee_{i=1}^{n} p_i$ if $n > 1$). The interest reader can refer to [4] for more details about this problem.

```
let NullB      := /\i=1..n ~B_i in
let Conclusion := \/i=1..n (A_i (+) S_i) in

Adder() /\ NullB() /\ Conclusion()
```

The software simply prints the status of the schema (satisfiable or unsatisfiable). Options are provided to get more information about the search space (number of inference rules, depth of unfolding etc.). See the manual for details. An additional tool is offered to expand the schema into a propositional formula in DIMACS format (by fixing the value of n). Several examples are available in the distribution (encoding various properties of n-bits adders, automata inclusion and Presburger arithmetic formulae). For instance, the associativity of the ripple-carry adder runs in about 2s with about 240000 applications of the propositional inference rules and 422 applications of the iteration rules.

5 Implementing the Proof Procedure

The Strategy. [2] describes a simple strategy for ensuring termination on regular schemata:

- The propositional extension, looping, closure and purity rules are applied with the highest priority.
- The iteration rules are applied only on iterations of maximal length. More precisely, an iteration $\Pi_{\mathtt{i}=a}^{\mathtt{n}-b}$ is eligible only if $\mathtt{n}-b-a$ is maximal w.r.t. the usual ordering between arithmetic expressions. For instance, if the current branch contains both $\bigvee_{\mathtt{i}=1}^{\mathtt{n}} p_{\mathtt{i}}$ and $\bigwedge_{\mathtt{i}=1}^{\mathtt{n}-1} q_{\mathtt{i}}$ then the decomposition rules do not apply on $\bigwedge_{\mathtt{i}=1}^{\mathtt{n}-1} q_{\mathtt{i}}$.

To improve efficiency, we need to strengthen the application conditions of the rules. The heaviest operation is of course the looping rule: we have to store all the previously generated sets into a database and to check at each step whether the current set occurs in the database (up to a shift on the parameter). From a practical point of view, the database must be kept as small as possible. Fortunately, from the proof of termination in [2] we know we can ensure termination by checking looping *just after* the purity rule. As a consequence this is also the best place to *insert* a node in the database.

The purity rule is also costly because one has to go over every literal in the node. By chance, it is only needed for applying the looping rule. Thus we can apply the purity rule only once; and the ideal place for this is just after all propositional extension rules. So we get the following strategy:

- Propositional extension and closure rules are applied until saturation.
- The purity literal, looping and iteration rules are applied, in that order.
- Back to the beginning.

Handling Indices in the Closure Rule. The purity and closure rules require the handling of arithmetic constraints, which is easy from a theoretical point of view but drastically reduces the efficiency. For instance $\neg P_3$ is not pure in $\bigwedge_{\mathtt{i}=1}^{\mathtt{n}} P_{\mathtt{i}} \Rightarrow P_{\mathtt{i}+1}$ in general, but it is pure if $\mathtt{n} < 2$. The indices must be compared w.r.t. the arithmetic constraints occurring in the branch. This problem is well known to be very complex in general. Furthermore, this prevents us from using efficient data-structures such as search trees or hash-tables. However, due to the restrictions on regular schemata and to the fact that the rules do not need the whole expressive power of linear arithmetic, we can easily optimize the handling of those rules. We only detail the case of the closure rule (the purity rule is presented in [3]).

First, notice that the constraints that are added to the nodes always have the form $\mathtt{n} - b \geq a$ or $\mathtt{n} - b < a$ (where $\mathtt{n}$ denotes the parameter and $a, b \in \mathbb{Z}$). By definition, the initial constraints (those occurring at the root level) are of the form $\mathtt{n} \geq c$ or $\mathtt{n} < c$. So it is easily seen that any conjunction of such constraints can be normalized to one of the forms $\top$, $\bot$, $\mathtt{n} = c$, $\mathtt{n} \geq c$, $\mathtt{n} < c$, or $\mathtt{n} \geq c_1 \wedge \mathtt{n} < c_2$ (with $c_1 \neq c_2 - 1$). Among those forms only $\mathtt{n} = c$ (and $\bot$) can entail $a = b$ for two syntactically distinct indices a, b. As a consequence either the constraint is not of the form $\mathtt{n} = c$ in which case syntactic equality can be used, or it is of the form $\mathtt{n} = c$ in which case we just substitute $\mathtt{n}$ with c and eliminate the arithmetic parameter. Then we can construct a search tree and look for the negation of the literal. Of course constructing this tree each time we look for a single literal is absurd. To enable the reuse of this tree, we observe that when the closure rule applies, the only other rules that can apply are propositional extension rules, which do not affect the constraints in the branch, thus we do not need to reconstruct the tree.

Looping. Of course the heaviest task is the maintenance and look up of the database. Once again because of the arithmetic constraints we cannot directly use classical data structures. But again we can define a normal form for which syntactic equality is sufficient. First of all, notice that the labels do not need to be sets: lists are sufficient. This does not affect termination: the number of possible nodes is still finite, though much bigger. Complexity is transferred from the looping rule to the whole algorithm. It seems we did not gain anything but actually switching from sets to lists has many advantages: a total order can easily be defined between lists, thus a set of labels may be represented through a search tree. Similarly a hash function compatible with equality is much easier to define and can be computed much faster than with sets. Finally, sets of lists are canonically handled by a usual data structure: tries [7]. The membership test is linear in the size of the label and independent of the size of the set[4].

In practice, it can be observed that switching from sets to lists does not really increase the length of the branch (i.e. no cycle is "missed"). Indeed when a set Φ loops on Ψ, each schema in Φ loops on an element of Ψ. Often the elements that loop on each other have the same "role", i.e. they come from the same subterm of the original schema. As the order of the rules is deterministic in our implementation these elements will often occur in the same position in the list.

6 Conclusion

We described a theorem prover for iterated schemata of propositional formulae. This system is able to check (for a large class of schemata) that a sequence

[4] Our implementation is actually not as efficient, but is sufficient in practice.

of propositional formulae depending on a parameter n is valid or unsatisfiable for every value of $n \in \mathbb{Z}$. This obviously requires some form of mathematical induction, which is achieved in our context by a cycle detection rule (the *looping* rule). We are not aware of any existing system offering similar features (in order to use a SAT solver, n must be instantiated). The problems we consider could be encoded into general purpose higher-order systems, but such systems are usually not suitable for fully automated, efficient, theorem proving.

Future work includes the extension of RegStab to non regular schemata, which can be achieved in many cases by translating the considered problem into a regular one. Some useful features will be added, for instance the program could return models of satisfiable formulae[5]. *Refutation schemata* could be constructed in case the formula is unsatisfiable. It should be noticed that currently, as it can be expected, our system is not very efficient when restricted to purely propositional logic (because it uses a very straightforward decision procedure). Thus combining it with more efficient SAT solvers that could be in charge of the propositional part is a very natural idea. However this is not straightforward since such systems cannot be simply used as "black boxes", mainly due to the fact that a *partial* evaluation is needed to propagate the values into the iterations. The implementation of the DPLL-based procedure in [8] will also be considered. This will have the advantage that some optimizations that have been proposed in order to improve the efficiency of the DPLL procedure could be incorporated into our system.

References

1. Bordeaux, L., Hamadi, Y., Zhang, L.: Propositional satisfiability and constraint programming: A comparative survey. ACM Comput. Surv. 38(4) (2006)
2. Aravantinos, V., Caferra, R., Peltier, N.: A schemata calculus for propositional logic. In: Giese, M., Waaler, A. (eds.) TABLEAUX 2009. LNCS, vol. 5607, pp. 32–46. Springer, Heidelberg (2009)
3. Aravantinos, V., Caferra, R., Peltier, N.: Complexity of the satisfiability problem for a class of propositional schemata. In: LATA 2010 (Language, Automata Theory and Applications). LNCS. Springer, Heidelberg (2010)
4. Aravantinos, V., Caferra, R., Peltier, N.: A Decidable Class of Nested Iterated Schemata. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS (LNAI), vol. 6173, pp. 293–308. Springer, Heidelberg (2010)
5. Giese, M.: Simplification rules for constrained formula tableaux. In: Cialdea Mayer, M., Pirri, F. (eds.) TABLEAUX 2003. LNCS, vol. 2796, pp. 65–80. Springer, Heidelberg (2003)
6. Caferra, R., Zabel, N.: Building models by using tableaux extended by equational problems. Journal of Logic and Computation 3, 3–25 (1993)
7. Fredkin, E.: Trie memory. ACM Commun. 3(9), 490–499 (1960)
8. Aravantinos, V., Caferra, R., Peltier, N.: A DPLL proof procedure for propositional iterated schemata. In: Workshop "Structures and Deduction 2009" (ESSLI), pp. 24–38 (2009)

[5] The extraction of the model from irreducible open branches is straightforward but it has been avoided in the current implementation for efficiency reasons.

Linear Quantifier Elimination as an Abstract Decision Procedure

Nikolaj Bjørner

Microsoft Research, One Microsoft Way, Redmond, WA, 98074, USA
nbjorner@microsoft.com

Abstract. This paper develops abstract quantifier elimination procedures for linear arithmetic over the reals and integers. They are formulated as theory solvers in the context of an abstract DPLL-based search. The resulting procedures allow the solvers to maintain integral control of the search process. We also evaluate this procedure and compare it with several alternatives. So far, the evaluation indicates that the proposed approach has some compelling advantages.

1 Introduction

Quantifier elimination for linear arithmetic remains a classical but timely challenge for automated deduction tools. Applications ranging from program analyzers based on separation logic [4], [6] to predicate abstraction [18] rely on tools for performing quantifier elimination.

Our current work draws motivation from two main sources. The first motivation was exploiting the benefits of efficient DPLL search and decision procedures based on integer linear programming for quantifier-elimination. The work in [13] addresses linear arithmetic over the reals. It uses an All-SMT loop to enumerate satisfiable monomes (conjunctions of literals), and applies quantifier elimination procedures on the monomes. A second motivation came from the application of quantifier elimination for predicate abstraction [18] where a large repository of challenging benchmarks have been recently made available. These benchmarks showed to us some limitations of the All-SMT approach: the number of satisfiable monomes to a formula can potentially be much larger than the case analysis inherent in the quantifier elimination procedures. This prompted developing a procedure where case analysis is bounded by the minimum branching of All-SMT *and* the quantifier-elimination procedure. The solver still enjoys the benefits of using ground decision procedures. Another takeaway from our approach is that substitutions are encoded into constraints and therefore represented implicitly during quantifier elimination. We found that this reduces the overhead of creating often useless terms during search.

To give an idea of a challenge in practical quantifier elimination, consider the following formula extracted from [18]. The predicates $p_0, \ldots, p_{23}$ correspond to the abstracted program state and the arithmetical sub-formulas encode a transition relation without abstraction. Quantifier-elimination produces the most precise abstraction for the transition relation.

$$\begin{array}{lll}
\exists x_1, x_2, x_3, x_4, x_5, x_6, x_7 & & \\
\quad (p_0 \equiv (x_4 < x_2)) & \wedge (p_1 \equiv (x_1 > x_4)) & \wedge (p_2 \equiv (x_4 > x_2)) \\
\wedge (p_3 \equiv (x_5 > x_2 + 17)) & \wedge (p_{23} \equiv x_4 \simeq 0) & \wedge (p_5 \equiv (x_6 > x_2)) \\
\wedge (p_7 \equiv (x_1 < x_4)) & \wedge (p_6 \equiv (x_3 > x_2 + 16)) & \wedge (p_9 \equiv (x_1 < x_5)) \\
\wedge (p_{10} \equiv (x_1 < x_6)) & \wedge (p_{19} \equiv (x_7 < x_6)) & \wedge (p_{11} \equiv (x_3 > x_1 + 16)) \\
\wedge (p_{12} \equiv (x_5 < x_2 + 1)) & \wedge (p_{13} \equiv (x_1 > x_5)) & \wedge (p_{14} \equiv (x_6 < x_2)) \\
\wedge (p_{17} \equiv (x_5 > x_7 + 1)) & \wedge (p_{15} \equiv (x_1 > x_6)) & \wedge (p_{16} \equiv (x_7 < x_4)) \\
\wedge (p_{18} \equiv (x_5 > x_7)) & \wedge (p_8 \equiv (x_1 < x_5 + 16)) & \wedge (p_{20} \equiv (x_3 < x_2)) \\
\wedge (p_{21} \equiv (x_3 < x_1)) & \wedge (p_{22} \equiv (x_3 < x_1)) & \wedge (p_4 \equiv (x_5 > x_2 + 1))
\end{array}$$

An approach based on enumerating satisfiable monomes could potentially produce 2^{24} satisfying cases. On the other hand, there are only 24 relevant atoms using 7 variables; a case analysis extracted from using Cooper's method has a much lower bound.

We also develop a quantifier-elimination procedure for Presburger arithmetic that combines features from the Omega test and Cooper's algorithm. We believe it is new and our experiments indicate that it offers some advantages to either of the known approaches.

1.1 Related Work

Linear quantifier elimination has received so much attention that we can only hope here to give a very partial list of contributions. The Omega tool [16] is well-known from program analysis. It uses an adaptation of the Fourier-Motzkin elimination procedure for integer linear programming. It relies on auxiliary procedures for converting formulas into disjunctive normal form. LIRA [3] implements procedures for mixed integer/real linear programming using automata-theoretic methods [7]. Redlog [10] contains several quantifier elimination procedures for different variants of arithmetic, such as the P-adics. Isabelle [15] contains a set of verified algorithms for linear quantifier elimination. These algorithms share a common skeleton and only differ in auxiliary routines. Mjollnir [13] implements decision procedures for linear real arithmetic. It integrates with the SMT solver Yices to efficiently extract satisfiable monomes using an All-SMT loop. An important insight is that quantifier elimination can often work much more efficiently on such small conjunctions as opposed to globally on a formula. To make the All-SMT loop more effective, the SMT solver is used to also minimize the monomes that are used for quantifier elimination. Our own experiments confirmed the importance of this step, but also indicated that it could also account for significant portion of the resulting run-time. Mjollnir's quantifier elimination procedure also interacts to make the All-SMT loop more efficient: the result of eliminating quantifiers from a monome is negated and added as a constraint. Recently, the LDD package [18] for linear differential diagrams (for UTVPI, which is integer arithmetic where inequalities are of the form $\pm x \pm y \leq k$, where x, y are variables and k is an integer constant) integrates a Fourier-Motzkin elimination procedure tightly with the BDD package CUDD.

The effective integration of arithmetic theories into first-order saturation and instantiation-based provers has also been a long-running enterprise, starting with theory resolution and continuing with specialized integration of arithmetic [9], [2], [17]. A tool based on [1] uses an older version of the quantifier-elimination procedures from Z3 [5].

The rest of the paper is organized as follows: After notation has been fixed, Section 3 presents the main idea behind our quantifier-elimination approach in the context of reals. This gets refined into an abstract decision procedure in Section 4. An analogous elimination procedure for integers is given in Section 5. Sections 6 and 7 are devoted to some of the practical problems we are faced with for scalability. Section 8 contains experimental feedback.

2 Preliminaries

The underlying domain for numerals is the set of integers, $\mathcal{Z}$, for integer linear arithmetic, and the set of reals, $\mathcal{R}$ for the theory of linear arithmetic over reals. We use letters $a, b, c, d, e, a_1, a_2, ..$ to represent constant numerals, and $x, y, z, x_1, x_2, ..$ for variables. The letters s, t range over terms, that are assumed in summation normal form $t ::= a_1x_1 + a_2x_2 + \cdots + a_nx_n + c$.

It will be convenient to highlight a distinguished variable x in each atomic formula. For integer linear arithmetic, atoms are inequalities and (negated) divisibility constraints:

$$\mathit{atom} ::= ax \leq t \quad | \quad bx \geq s \quad | \quad c|(dx+t) \quad | \quad \neg c|(dx+t)$$

We will write the atoms so that $a, b, c, d > 0$ and x does not appear in s, t. Equalities $x \simeq t$ will be treated as shorthand for $x \leq t \wedge x \geq t$. Atoms for linear reals are normalized so that x uses the unit coefficient, 1.

$$\mathit{atom} ::= x \simeq t \quad | \quad x > s \quad | \quad x < t$$

Non-strict inequality $x \geq s$ is just a shorthand for $x > s \vee x \simeq s$.

Our algorithms work on formulas without negations, except for negative divisibility,

$$\varphi ::= \mathit{true} \quad | \quad \mathit{false} \quad | \quad \mathit{atom} \quad | \quad \varphi \wedge \varphi \quad | \quad \varphi \vee \varphi \quad | \quad \exists x \,.\, \varphi \quad | \quad \forall x \,.\, \varphi$$

We will nevertheless use normal logical connectives, such as negation $\neg$ and implication $\rightarrow$ to write formulas, and appeal to a straight-forward negation normal form conversion.

An example valid (integer) linear arithmetic formula[1] is

$$\exists \ell \,.\, \forall x \,.\, ((x \leq \ell) \;\vee\; \exists y, z \,.\, (y \geq 0 \wedge z \geq 0 \wedge x \simeq 7y + 8z))$$

It encodes an instance of the well-known stamp problem, that asks whether 7 and 8 cent stamps can be combined to cover every cent after a bound ℓ.

[1] apparently attributed to Bob Constable

It will be useful to introduce notation for finite range quantifiers, that can be kept as is, or expanded to finite disjunctions.

$$(\exists z \in [0 \dots a])\varphi \quad \equiv \quad \exists z \,.\, 0 \leq z \leq a \wedge \varphi \quad \equiv \quad \bigvee_{z=0}^{a} \varphi$$

We write $\varphi[x]$ to refer to all occurrences of x in φ. The notation extends to terms and formulas. So, $\varphi[\psi]$ refers to all occurrences of a sub-formula ψ inside of φ. We write $\varphi[t/x]$ for the formula where x is replaced by the term t.

Our procedures will work on formulas $\exists x \,.\, \varphi$, where φ is quantifier-free. This means that quantifier elimination works bottom-up in the formula. Universal quantifiers ($\forall$) are handled using the de-Morgan duals ($\neg\exists\neg$).

We use terminology from DPLL(T) [14] to describe our algorithms. DPLL(T) works with a partial propositional assignment Γ and a set of clauses F. Forcing $\Gamma \Vdash F$ is used when F evaluates to true under the partial assignment Γ. We will not go into much background on Satisfiability Modulo Theories solvers here; the main feature that is central to our approach is that they include efficient solvers for checking satisfiability of quantifier-free formulas over linear arithmetic.

3 Linear Real Arithmetic

Let us first recollect Loos-Weispfenning's [11] method for quantifier elimination. It will be used to derive the abstract theory solver version. Given a formula $\exists x \,.\, \varphi$, where φ is quantifier free and such that all occurrences of x have been isolated, we refer to E as the set of atoms in φ of the form $x \simeq t$, L is the set of atoms of the form $x > s$ and U is the set of atoms $x < t$. The notation is borrowed from [15]. The Loos-Weispfenning method then amounts to the following equivalence:

$$\exists x \,.\, \varphi \equiv \varphi[\infty/x] \vee \bigvee_{(x \simeq t) \in E} \varphi[t/x] \vee \bigvee_{(x < t) \in U} \varphi[t - \epsilon/x] \qquad (1)$$

There is also an equivalence using $-\infty$ and L instead of ∞ and U. It is advantageous to use this when $|L| < |U|$ to avoid case analysis, but we will work with just this version to simplify the presentation. The infinitesimals can be eliminated after substitution: $(\infty \simeq t) = (\infty < t) = \mathit{false}$, $(\infty > s) = \mathit{true}$, $(s - \epsilon > t) = (s > t)$, $(s - \epsilon < t) = (s \leq t)$.

We will use this equivalence as a starting point to a formulation that is suitable as an abstract decision procedure in the context of DPLL(T). The crux of the method is quite simple, but essential: *instead of applying substitutions, encode the effect of substitutions in a new formula.* A satisfying assignment to the new formula will then respect the substitution. The first step is to introduce a finite domain quantifier to encode the finite set of substitutions that are required for the Loos-Weispfenning method. The second step requires the substitution to preserve truth monotonically. The construction uses references to the elements in the set $\{\infty\} \cup E \cup U$, so we will resort to subscripts to make this feasible.

So set $U = \{(x < t_1), \ldots, (x < t_m)\}$ and $E = \{(x \simeq t_{m+1}), \ldots, (x \simeq t_n)\}$. We now will introduce predicates pivot_i, for $i = 0, \ldots, n$ to cover a case analysis for the values of x that satisfy φ. They correspond to the disjunctions in (1) and they encode the conditions where the respective disjunction holds. Thus, pivot_0 encodes $\varphi[\infty/x]$, and for $(x < t_i)$, pivot_i encodes $\varphi[t_i - \epsilon/x]$, and for $(x \simeq t_i)$, pivot_i encodes $\varphi[t_i/x]$.

Definition 1 (Real Arithmetic Pivoting)

$$\mathit{pivot}_0 = \bigwedge_{atm \in E \cup U} \neg atm \wedge \bigwedge_{atm \in L} atm$$

$$\mathit{pivot}_{1 \le i \le m} = x < t_i \bigwedge_{atm \in E} \neg atm \bigwedge_{(x<t) \in U} (x < t \rightarrow t_i \le t) \bigwedge_{(x>s) \in L} (x > s \rightarrow t_i > s)$$

$$\mathit{pivot}_{m < i \le n} =$$
$$x \simeq t_i \bigwedge_{(x \simeq t) \in E} (x \simeq t \rightarrow t_i \simeq t) \bigwedge_{(x<t) \in U} (x < t \rightarrow t_i < t) \bigwedge_{(x>s) \in L} (x > s \rightarrow t_i > s)$$

Proposition 1. *Let* $\mathit{pivot}(z) = \bigwedge_{i=0}^{n} (z \simeq i \rightarrow \mathit{pivot}_i)$, *then*

$$\exists x \;.\; \varphi \equiv \exists x \;.\; (\exists z \in [0 \ldots n]) \;.\; \varphi \wedge \mathit{pivot}(z)$$

It is easy to argue informally for the lemma: Any interpretation $\mathcal{M}$ that satisfies φ assigns some value to x. There are three main cases for this value: (1) $x^{\mathcal{M}}$ is equal to some $t^{\mathcal{M}}$, where $(x \simeq t) \in E$, the case is encoded in $\mathit{pivot}_{m<i\le n}$. The resulting model will also have to satisfy all implications. Case (2), $x^{\mathcal{M}}$ is bigger or equal to every $t^{\mathcal{M}}$, where $(x < t) \in U$, but then an interpretation where $x^{\mathcal{M}}$ is also bigger than every $s^{\mathcal{M}}$ will satisfy φ, so pivot_0 must hold (in the modified model). Case (3), $x^{\mathcal{M}}$ is smaller than some $t^{\mathcal{M}}$ where $(x < t) \in U$, then one of these is the least. Setting $x^{\mathcal{M}}$ arbitrary close to this least upper bound lets us satisfy $\mathit{pivot}_{1\le i \le m}$. The converse direction is immediate because φ is already in the formula [2]

We can now derive a quantifier elimination procedure based on a ground satisfiability procedure. Every satisfying assignment to $0 \le z \le n \wedge \varphi \wedge \mathit{pivot}(z)$ assigns z to some value i between 0 and n, and for each of these values, pivot_i constrains the value of x with respect to the constraints E, L, U. At this point, apply the substitution on φ that corresponds to each literal or right hand side of each implication in pivot_i and obtain the formula without x.

Our construction can also be iterated as a procedure for multiple bound variables. To simulate the effect of substitution, the new atomic formulas $t_i < t, t_i \simeq t, t_i \le t, t_i > s$, that appear on the right side of implications must be included as part of the sets E, L and U that are used to eliminate the next variable in the new formula.

[2] There is an alternative definition of pivot_i that allows replacing $z \simeq i \rightarrow \mathit{pivot}_i$ by $z \simeq i \equiv \mathit{pivot}_i$. This version is obtained by breaking symmetries: add $\bigwedge_{(x<t_j)\in U, j<i} (x < t_j \rightarrow t_i < t_j)$ to $\mathit{pivot}_{q \le i \le m}$, and add $\bigwedge_{(x \simeq t_j) \in E, j<i} x \not\simeq t_j$ to $\mathit{pivot}_{m<i\le n}$.

4 An Abstract Quantifier Elimination Procedure

Expanding φ up front increases the size of it unnecessarily. There is instead an advantage to integrate pivoting as a decision procedure in the DPLL(T) framework. Some of the main practical advantages include that the framework allows for inter-operating with other decision procedures, inspecting the current state to avoid introducing redundant clauses, and it allows for garbage collecting clauses once they become redundant.

Figure 1 summarizes the theory-solver based elimination procedure for the single variable case. It works over a formula $\exists x\ .\ \varphi$. We follow the style of the abstract DPLL(T) presentation and write the state of the current search as a combination of a stack of assigned literals, Γ and a formula F (in clausal normal form). To track the current partial result of quantifier elimination the state also includes a formula R(esult).

Only four rules, additional to the usual DPLL(T) rules are required. In practice (in our implementation) the existing way that theory solvers are plugged in can directly accommodate these rules without extending the DPLL(T) core. The first rule Init provides the initial state of the quantifier elimination. It contains the empty assignment ϵ, and the formulas φ and a constraint that z ranges from 0 to n. This forces the underlying propositional and theory solvers to choose a value for z. The Pivot rule applies when some value is taken. Pivoting introduces the set of assumptions corresponding to the disjunction encoded by $z \simeq i$. All these pivots are conditioned on $z \simeq i$. Once the context Γ forces $z \not\simeq i$, existing mechanisms for garbage collecting clauses can take effect. The Elim rule applies when Γ is an assignment that satisfies F. The rule leaves some room for flexibility: while our own implementation always checks Γ for consistency with respect to the theory of linear arithmetic, it is not required by the rule. In this state, we apply the substitution corresponding to the i'th pivot to φ, add the disjunction to R, and block this branch. The rules cannot be applied forever: They ensure that eventually F contains $z \not\simeq i$ for each $i = 0, .., n$, which is inconsistent.

$$\textsf{Init}\ \frac{}{\Longrightarrow\ \epsilon \parallel \varphi, \bigvee_{i=0}^{n} z \simeq i \parallel \mathit{false}}$$

$$\textsf{Pivot}\ \frac{\Gamma \Vdash (z \simeq i);\ \text{but not}\ \Gamma \Vdash \mathit{pivot}_i.}{\Gamma \parallel F \parallel R\ \Longrightarrow\ \Gamma \parallel F, (z \simeq i \rightarrow \mathit{pivot}_i) \parallel R}$$

$$\textsf{Elim}\ \frac{\Gamma \Vdash F, z \simeq i, \mathit{pivot}_i}{\Gamma \parallel F \parallel R\ \Longrightarrow\ \Gamma \parallel F, (z \not\simeq i) \parallel R \vee \varphi[\mathit{pivot}_i]}$$

$$\textsf{Return}\ \frac{\Gamma \Vdash \mathit{false}\ \text{(or } F \text{ contains the empty clause)}}{\Gamma \parallel F \parallel R\ \Longrightarrow\ R}$$

Fig. 1. An Abstract Quantifier Elimination Procedure for Reals

Section 6 describes at a high level the elimination of multiple variables using depth-first or breadth-first strategies. The choice of strategy can have a significant impact on the search space.

5 Linear Integer Arithmetic

In this section we show how to lift the quantifier-elimination method for the reals to the case of integers. The result is an algorithm that combines elements from Cooper's method and the Omega test. In a nutshell, we utilize the Omega test method for resolving integer inequalities in order to avoid taking least common multiples with coefficients that come from inequalities. The resulting algorithm allows creating smaller intermediary disjunctions during quantifier elimination and is to our knowledge new.

After a review of Cooper's method and the integer resolution method from the Omega test, we discuss in Section 5.2 discusses how divisibility constraints can be combined. Then Section 5.3 shows how the pieces can be combined.

5.1 Cooper's Algorithm and the Omega Test

Before proceeding with the algorithm, let us briefly review Cooper's elimination procedure and how the Omega test can be adapted to eliminating quantifiers. We will be eliminating x from φ. The set U consists of atoms of the form $ax \leq t$, L of $bx \geq s$, and D are the atoms $c|(dx+u)$. Let δ be the least common multiplier of the coefficients to x from U and the divisors in D. (later we will see that our procedure just needs the lcm from D). Cooper's procedure is based on the equivalence (see also [7]):

$$(\exists x \, . \, \varphi) \equiv \bigvee_{i=0}^{\delta-1} \left(\varphi[\infty - i/x] \;\vee \bigvee_{ax \leq t \in U} a|(t-i) \wedge \varphi[(t-i)/ax] \right)$$

As customary, set $(bx \geq s)[t/ax] = (bt \geq as)$, and $c|(bx+s)[t/ax] = ac|(bt+as)$. The formula $\varphi[\infty - i/x]$ is defined in a similar way by using the replacements $s \leq bx \leftarrow \mathit{true}$ and $cx \leq t \leftarrow \mathit{false}$ and $c|(dx+t) \leftarrow c|(-di+t)$.

The core of the Omega test is an integer variant of the Fourier-Motzkin resolution principle. The presentation in [16] suggests to use resolution to conjunctive constraints $\bigwedge_i a_i x \leq t_i \wedge \bigwedge_j b_j x \geq s_j$, creating $|L| \times |U|$ resolvents. While resolution can be generalized to constraints in DNF, it does not reduce the multiplicative expansion. Working with conjunctive constraints requires also a phase that converts the formula into DNF, which is one significant bottleneck in the Omega package. It will here suffice to use just the binary variant of the Omega test, which we formulate here as:

$$\begin{aligned} &\mathit{resolve}(ax \leq t, bx \geq s) = \\ &\quad as + (a-1)(b-1) \leq bt \;\vee \\ &\quad a \geq b \wedge as \leq bt \wedge (\exists z \in [0 \ldots b-2])(b|(s+z) \wedge a(s+z) \leq bt) \;\vee \\ &\quad b > a \wedge as \leq bt \wedge (\exists z \in [0 \ldots a-2])(a|(t-z) \wedge as \leq b(t-z)) \end{aligned} \tag{2}$$

Resolution can be used to eliminate x:

Lemma 1 (Integer Resolution)

$$(\exists x \ . \ ax \leq t \wedge bx \geq s) \equiv \mathit{resolve}(ax \leq t, bx \geq s) \tag{3}$$

Our formulation of the binary resolution rule is close to, but not quite identical to the version in [16]. In particular, the terms used in the bounded disjunctions have been simplified. Notice that when either $a = 1$ or $b = 1$, then the equivalence simplifies to:

$$(\exists x \ . \ ax \leq t \wedge bx \geq s) \equiv as \leq bt$$

There are two components to the new algorithm. The first component is to extract and process divisibility constraints. The second component is close to the Loos-Weispfenning formulation, but here applied to integers. It selects a least upper bound (or greatest lower bound) for the variable x and either collects the bound constraints or applies the linear integer resolution principle.

5.2 Divisibility Constraints

Divisibility constraints can be handled at the cost of introducing one auxiliary bounded variable. Let δ be the least common multiple of the divisors in D, then for each divisor term in D we have the equivalence:

$$c|(ax + t) \equiv c|(a(x \ \mathit{mod} \ \delta) + t)$$

Therefore, we can introduce an auxiliary variable u and define the predicate

$$\mathit{div\_elim} := \delta|(x - u) \bigwedge_{c|(dx+t) \in D} (c|(dx + t) \leftrightarrow c|(du + t)) \tag{4}$$

Then the following is a tautology (true for all free variables in φ):

$$(\exists u \in [0 \ldots \delta - 1]) \ . \ \mathit{div\_elim}.$$

The last equivalences in *div_elim* encode replacement of the original divisibility constraints with atoms where x is eliminated. If we wish to eliminate the remaining divisibility constraint we can use the equivalence:

$$\exists x \ . \ \delta|(x - u) \wedge \varphi \equiv \exists x, y \ . \ \delta y \simeq x - u \wedge \varphi \quad \equiv \quad \exists x \ . \ \varphi[\delta x + u/x] \tag{5}$$

We will make use of this equivalence in the next section, but in a lazy way.

Remark: In fact, there is a more general, Euclidean inspired, way to eliminate arbitrary divisibility constraints by using the additional equivalences:

$$\begin{aligned}
c|(ax + t) &\equiv c|((a \ \mathit{mod} \ c)x + t) \\
\exists x \ . \ 1|(ax + t) \wedge \varphi &\equiv \exists x \ . \ \varphi \\
\exists x \ . \ c|(ax + t) \wedge \varphi &\equiv \exists x \ .a|((c \ \mathit{mod} \ a)x - t) \wedge \varphi[(cx - t)/ax] \quad \text{for } c > a > 1
\end{aligned}$$

The equivalences produce substitutions. These can be combined to a single substitution $[bx - t/ax]$ because when $R(x, y)$ ranges over the relations $a'x \leq y$ and $b'x \geq y$, then

$$R(x, y)[bx - t/ax][dx - s/cx] = R(bx - t, ay)[dx - s/cx] = \\ R(bdx - bs - ct, acy) = R(x, y)[bdx - ct - bs/acx]$$

In summary, a formula that existentially binds x, possibly using divisibility constraints on x, can be equivalently represented as

$$\exists x \ . \ c|(dx + s) \wedge \varphi \equiv \exists x \ . \ \varphi[bx - t/ax] \tag{6}$$

where $[bx - t/ax]$ is the substitution obtained from $c|(dx + s)$, and φ does not contain divisibility constraints. □

5.3 Integer Pivoting

In the following we will assume we are given the formula $\exists x \ . \ \varphi$ and that *div_elim* is obtained using (4). Let $\theta := [\delta x + u/x]$ be the substitution corresponding to the divisibility constraint in (5). The identifiers in the substitution are primed so they are not confused with the identifiers used elsewhere. If c is 1, then θ is just the identity substitution $[x/x]$. Then in the same spirit as definition 1 we define shorthands for pivoting. Recall, that for integers, we only need to consider non-strict inequalities, so let $U = \{(a_1 x \leq t_1), \ldots, (a_m x \leq t_m)\}$ and define:

Definition 2 (Integer pivoting)

$$\begin{aligned}
\mathit{pivot}_0 &:= \bigwedge_{atm \in U} \neg atm \bigwedge_{atm \in L} atm \\
\mathit{pivot}_{1 \leq i \leq m} &:= a_i x \leq t_i \\
&\quad \bigwedge_{(ax \leq t) \in U} (ax \leq t \rightarrow at_i \leq a_i t) \\
&\quad \bigwedge_{(bx \geq s) \in L} (bx \geq s \rightarrow \mathit{resolve}((bx \geq s)[\theta], (a_i x \leq t_i)[\theta]))
\end{aligned}$$

and set

$$\mathit{pivot} := \mathit{div\_elim} \wedge \bigwedge_{i=0}^{n} (z \simeq i \rightarrow \mathit{pivot}_i)$$

Analogously to the case for linear real arithmetic we now have:

Proposition 2 (Elimination based on Integer Pivoting). *Assume φ is a quantifier free formula whose occurrences of x are summarized using the sets D, L and U*

$$\exists x \ . \ \varphi \equiv \exists x, (\exists u \in [0 \ldots \delta - 1]), (\exists z \in [0 \ldots n]) \ . \ \varphi \wedge \mathit{pivot}$$

Correctness of the proposition is by an argument similar to the correctness for Proposition 1.

Proof: The direction from right to left is immediate because the right side already contains the left.

For the direction from left to right, assume we have a model $\mathcal{M} \models \varphi$. If $x^{\mathcal{M}}$ is such that every upper bound in U is false, we can choose $x^{\mathcal{M}}$ arbitrary large so that every lower bound in L is true. The case $z \simeq 0 \wedge \mathit{pivot}_0$ is satisfied in this model. On the other hand, let $(a_i x \leq t_i) \in U$ be the least upper bound with respect to $\mathcal{M}$. That is, for every other $(ax \leq t) \in U$ where $\mathcal{M} \models ax \leq t$ it is the case that it holds that $(t_i/a_i)^{\mathcal{M}} \leq (t/a)^{\mathcal{M}}$ (using rational division), which is the same as $\mathcal{M} \models at_i \leq a_i t$ (over the integers). This establishes the second conjunct of pivot_i. If there is a $bx \geq s \in L$, such that $\mathcal{M} \models bx \geq s$, there is a greatest lower bound in L with respect to $\mathcal{M}$. For this inequality the left side implies

$$\begin{aligned} \exists x \,.\, \delta|(x-u) \wedge bx \geq s \wedge a_i x \leq t_i &\equiv \exists x \,.\, (bx \geq s)[\theta] \wedge (a_i x \leq t_i)[\theta] \\ &\equiv \mathit{resolve}((bx \geq s)[\theta], (a_i x \leq t_i)[\theta])) \end{aligned}$$

The resolvents of the other lower bounds for x are weaker, so they are implied. $\square$

The abstract decision procedure that implements quantifier elimination for integers uses essentially the same rules as the one given for the reals in Figure 1.

$$\textsf{Init}\ \frac{}{\Longrightarrow\ \epsilon \parallel \varphi, \mathit{div\_elim}, \bigvee_{i=0}^{n} z \simeq i, \bigvee_{j=0}^{\delta-1} u \simeq j \parallel \mathit{false}}$$

$$\textsf{Pivot}\ \frac{\Gamma \Vdash (z \simeq i);\ \text{but not } \Gamma \Vdash \mathit{pivot}_i.}{\Gamma \parallel F \parallel R\ \Longrightarrow\ \Gamma \parallel F, (z \simeq i \rightarrow \mathit{pivot}_i) \parallel R}$$

$$\textsf{Elim}\ \frac{\Gamma \Vdash F, z \simeq i, u \simeq j, \mathit{pivot}_i}{\Gamma \parallel F \parallel R\ \Longrightarrow\ \Gamma \parallel F, (z \not\simeq i \vee u \not\simeq j) \parallel R \vee \varphi[\mathit{pivot}_i][j/u]}$$

$$\textsf{Return}\ \frac{\Gamma \Vdash \mathit{false}\ \text{(or } F \text{ contains the empty clause)}}{\Gamma \parallel F \parallel R\ \Longrightarrow\ R}$$

Fig. 2. An Abstract Quantifier Elimination Procedure for integers

There is one important exception though: resolution and the divisibility constraints introduce finite range (existential) variables in the state. We would like to find the set of instances from these values that satisfy the formula. This is in principle obtained by enumerating the finite set of satisfying assignments to these integers. In the context of the SMT solver Z3, and we believe the experience applies generally to other contexts, we found that a bit-wise encoding of the finite range variables offered dramatic performance gains. We outline the encoding in Section 7.

6 Elimination of Multiple Variables

The introductory example formula illustrated that we should in general expect to eliminate several variables. The order in which the variables are eliminated has a significant impact on search. A possible heuristic known from QBF solvers is to count the number of occurrences of each variable and each polarity. The variable with the least number of occurrences subject to a polarity is chosen first. Similarly, an obvious heuristic is to select the variable whose set $L \cup E$ or $U \cup E$ is the smallest (for reals). For integers minimizing for the least common multiple of the coefficients and divisors to x has a significant impact on the branching.

Another factor that can have a significant impact on performance is whether to eliminate variables together (depth-first) or independently (breadth-first). So in breadth-first elimination for the formula $\exists x, y \,.\, \varphi$, one computes first $\varphi' \equiv \exists x \,.\, \varphi$, where x is eliminated, then y is eliminated from φ'. In depth-first elimination, y gets eliminated from φ every time a substitution for x has been chosen. A combination of depth-first and breadth-first search seems desired. For example, given the formula $\exists y, z, x, u \,.\, \varphi[x, u, y] \wedge \psi[x, u, z]$ it is an advantage to eliminate y and z independently and combine the results instead of eliminating z in every disjunct generated when eliminating y. On the other hand, the dependencies between x and u could make a depth-first exploration more efficient.

The approach taken in our implementation is to use variable disjoint partitions of top-level conjunctions for breadth-first search. It allows using a simplistic algorithm. It splits φ into conjuncts $\varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_n$. Then, for each variable v, that occurs in fewer than $n/2$ conjuncts, create an equivalence class of the formulas where v occurs. Union-find is used to merge and maintain the classes. If more than one class remains, then the variables from each class are eliminated independently. The variables that occur in more than $n/2$ conjuncts are handled jointly as well.

7 From Bounded Integers to Bit-Vectors

We skipped one part of the integer linear arithmetic elimination procedure, namely elimination of finite range variables. The formula produced from quantifier elimination is of the form $(\exists z_1 \in [0 \ldots u_1 - 1]) \ldots (\exists z_n \in [0 \ldots u_n - 1])\psi$. These quantifiers can directly be treated as disjunctions, but the size of the resulting formulas can easily grow to be exponential in the number of bound finite range variables $(u_1 \cdot u_2 \cdot \cdots \cdot u_n \cdot |\psi|)$.

When only a few disjuncts are satisfied in the resulting formula, it pays to use an All-SMT loop. We found that encoding finite range integers as bit-vectors offered superior performance in contrast to an integer encoding; the intermediary conflict clauses produced for bit-vectors remain more precise than what is currently possible (at least to our experience) for the theory of integers.

Of course ψ can contain free integer variables that are not among the finite range $z_1, \ldots, z_n$. So we need to handle constraints that mix integers and bit-vectors. We introduce functions *bv2int* and $int2bv_{[n]}$ in order to convert from n-bit bit-vectors to integers and from integers to n-bit integers. The interpretation of these functions on any bit-vector term x, and integer term y are determined fully by adding constraints:

$$bv2int(x_{[n]}) = \sum_{i=0}^{n-1} ite(x_i, 2^i, 0) \qquad bv2int(int2bv_{[n]}(y)) \ = y \ mod \ 2^n$$

Let us define $bw(a) = \lceil \log_2(a) \rceil$, then the formula becomes

$$\exists y_1 : bv[bw(u_1)] \ . \ bvule(y_1, bv2int(u_1)) \wedge \ldots \exists y_n : bv[bw(u_n)] \ . \\ bvule(y_n, bv2int(u_n)) \wedge \psi[y_1/z_1, \ldots, y_n/z_n]$$

Nested occurrences of *bv2int* can now be propagated upwards by using transformations that propagate it over the the arithmetical functions $+, \cdot$ and relation $\leq$. Two examples illustrate the transformation: $bv2int(x_{[n]}) + bv2int(y_{[n]})$ becomes $bv2int((0_{[1]} \oplus x_{[n]}) +_{bv} (0_{[1]} \oplus y_{[n]}))$, where $\oplus$ is bit-vector concatenation. Also, $bv2int(x_{[n]}) \leq bv2int(y_{[n]})$ becomes $bvule(x_{[n]}, y_{[n]})$. The relation $bvule(x, y)$ performs unsigned comparisons on bit-vectors x and y.

8 Implementations and Experimentation

We implemented several variants of quantifier elimination procedures for reals and integers as part of Z3. In all our variants bounded quantifiers are eliminated using an All-SMT loop. Another important feature present in all variants is that quantifier elimination is avoided completely and replaced by a cheap satisfiability check when the only variables and uninterpreted symbols in φ are those to be eliminated. The variants we tried out are:

FM/Ω-SMT. Fourier-Motzkin/Omega test algorithms that work on monomes produced by an All-SMT loop. For the case of reals, this corresponds to algorithms implemented in Mjollnir.

LW/C-SMT. Loos-Weispfenning/Cooper's algorithm that works on monomes produced by an All-SMT loop. This is also an equivalent to the loop used by Mjollnir.

LW/C-Plain. Loos-Weispfenning/Cooper's algorithm that works directly on the negation normal form formula.

Mix-Model. Our new quantifier elimination algorithm that that uses the current interpretation $\mathcal{M}$ to select the substitutions to explore for each variable. In other words, if $\mathcal{M} \models ax \leq t$, where $(ax \leq t) \in U$, and $\mathcal{M}$ selects t/a as the least upper bound for x, then eliminate x based on this substitution. This method is different from the All-SMT strategy since it is guaranteed to only explore branches corresponding to the number of virtual substitutions for x.

Mix-SMT. The theory-solver based integration presented in this paper.

The first three variants are available in the current release v2.4 of Z3.

8.1 Benchmarking

In spite of long-running attention, we are aware of only a limited set of readily available benchmarks for linear quantifier elimination. We will here report on selected and somewhat circumstantial experiments. Admittedly, the experience is currently somewhat sporadic and not as established as with other SMT divisions.

Table 1. Experimental samples

Source	Type	TO	#	*FM*/*Ω*-SMT		LW/C-SMT		LW/C-Plain		Mix-Model		Mix-SMT	
HOL-l	LIA	∞	72	0.3s	0	3.1s	0	4.9s	0	0.3s	0	0.3s	0
LRA/m	LRA	∞	150	19.7s	0	18.0s	0	8.6s	0	10.3s	0	9.2s	0
LRA/R	LRA	100s	100	61s	15	240s	15	206s	11	270	14	405s	20
LDD	IDL	4s	100	54s	29	52s	29	28s	37	53s	29	66s	17
LDD	IDL	8s	100	135s	18	122s	19	91s	24	121s	19	86s	13

The examples are taken from different sources. We list the source, fragment of arithmetic (IDL is short for integer difference logic), chosen timeout, number of benchmarks in the set, and solver setting. The table fields contain split columns containing *cumulative time* and #*timeouts*, respectively. The HOL-light examples come from the HOL-light distribution[3]. The LRA benchmarks come from SMT-LIB (`http://www.smtlib.org`). They are contributed by Scholl et.al. whose LinAIG tool uses interpolants to simplify formulas. The 150 benchmarks from LRA/model are extracted from a model-checking tool. They are all easy, especially for LW/C-Plain and Mix-SMT. The All-SMT loop seems like a disadvantage here. The situation is different on the random benchmarks where using the SMT solver only slows solving down. This can be expected when most case splits are hard to satisfy. It is of course tempting to dismiss random benchmarks because their relationship with applications is dubious and can lead optimization efforts astray. We selected randomly 100 (but not random) benchmarks from more than 5000 LDD files kindly supplied by Aarie Gurfinkel, and we chose low timeouts of 4 and 8 seconds to get timely, but informative, feedback. In this case, the new DPLL-based integration is the clear winner and solves more benchmarks within 4s than any of the other methods can do in 8s. This is particularly true with the formula sketched in the introduction. Each of the methods *FM*/*Ω*-SMT, LW/C-SMT and Mix-Model spend around 80 seconds on such a small formula, LW/C-Plain can get away with spending 30 seconds, but Mix-SMT spends less than 2 seconds. In conclusion, the experimental feedback suggests that the Mix-SMT approach is generally robust and therefore should be the preferred method.

It will be very interesting in future work to compare with the LDD elimination procedures more carefully. The LDD benchmarks also contain quantifiers over Boolean variables. Currently, we don't handle such quantified variables with any sophistication. A fundamental difference is also how LDD algorithms can rely on

[3] `http://www.cl.cam.ac.uk/~jrh13/hol-light`

dynamic programming techniques, memoizing and re-using partial results built bottom-up. This is not as easy in the DPLL(T) framework where contexts Γ are built top-down.

One resident motivation for the work was the SLAyer [6] tool. It relies on quantifier elimination by Z3. Currently, all queries comprise of very small formulas that can be discharged in micro-seconds. Quantifier elimination is indispensable, but not a bottleneck so far. The SLAyer application also indicates a potentially useful extension to special case quantifier elimination for arrays (it is impossible in general). For example, SLAyer requires Z3 to understand that $\exists i, C\ .\ (A \simeq \mathit{store}(C, \mathit{flink}, i))$ is equivalent to *true*. The formula says that the array A is equal to some array C where i is stored at index *flink*.

9 Concluding Remarks

We have presented abstract quantifier elimination procedures for linear arithmetic. These integrate directly as theory solvers in the DPLL(T) framework. Case analysis is bounded by the elimination algorithms and All-SMT and they leverage efficient procedures for ground feasibility. Our procedure for Presburger arithmetic also has the potential to produce fewer top-level disjunctions than Cooper's classical algorithm: to handle divisibility constraints it requires the lcm of the divisors in D, disjuncts from the coefficients to x in U (or L) are distributed over resolution steps so they can be handled separately.

In our experience with Z3 users, we are increasingly finding requests for additional exotic functionality. Quantifier elimination for a suitable fragment of the array theory is one example. Quantifier elimination for inductive data-types could be used in model-based design tools that use Z3 for their Prolog-inspired specification language.

Recently, several works [8,12] and Scott Cotton's thesis have investigated a variant of the Fourier-Motzkin elimination procedure that uses partial models for guiding the use of resolution. It is not hard to establish how one can use pivoting combined with the resolution procedures, including the integer version, for these approaches, and we speculate if this can provide a performance benefit. The approaches still need to handle auxiliary atoms in the search space, but pivoting may indicate how these can be turned on and off.

We would also like to investigate whether the abstract quantifier elimination procedure for integers presents any advantages to integer linear programming based on Simplex augmented with Gomory cuts and branch and bound. The potential advantage is that elimination can complement cuts and branches.

Thanks to the reviewers for their detailed and highly constructive feedback.

References

1. Althaus, E., Kruglov, E., Weidenbach, C.: Superposition modulo linear arithmetic SUP(LA). In: Ghilardi, S., Sebastiani, R. (eds.) FroCoS 2009. LNCS, vol. 5749, pp. 84–99. Springer, Heidelberg (2009)

2. Baumgartner, P., Fuchs, A., Tinelli, C.: ME(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)
3. Becker, B., Dax, C., Eisinger, J., Klaedtke, F.: Lira: Handling constraints of linear arithmetics over the integers and the reals. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 307–310. Springer, Heidelberg (2007)
4. Chin, W.N., David, C., Nguyen, H.H., Qin, S.: Enhancing modular oo verification with separation logic. In: POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 87–99. ACM, New York (2008)
5. 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)
6. `http://research.microsoft.com/enus/um/cambridge/projects/slayer/`
7. Klaedtke, F.: On the automata size for presburger arithmetic. In: LICS, pp. 110–119. IEEE Computer Society, Los Alamitos (2004)
8. Korovin, K., Tsiskaridze, N., Voronkov, A.: Conflict resolution. In: Gent, I.P. (ed.) CP 2009. LNCS, vol. 5732, pp. 509–523. Springer, Heidelberg (2009)
9. 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)
10. Lasaruk, A., Sturm, T.: Effective quantifier elimination for presburger arithmetic with infinity. In: Gerdt, V.P., Mayr, E.W., Vorozhtsov, E.V. (eds.) CASC 2009. LNCS, vol. 5743, pp. 195–212. Springer, Heidelberg (2009)
11. Loos, R., Weispfenning, V.: Applying linear quantifier elimination. Comput. J. 36(5), 450–462 (1993)
12. McMillan, K.L., Kuehlmann, A., Sagiv, M.: Generalizing dpll to richer logics. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 462–476. Springer, Heidelberg (2009)
13. Monniaux, D.: A quantifier elimination algorithm for linear real arithmetic. In: Cervesato, I., Veith, H., Voronkov, A. (eds.) LPAR 2008. LNCS (LNAI), vol. 5330, pp. 243–257. Springer, Heidelberg (2008)
14. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). J. ACM 53(6) (2006)
15. Nipkow, T.: Linear quantifier elimination. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 18–33. Springer, Heidelberg (2008)
16. Pugh, W.: A practical algorithm for exact array dependence analysis. ACM Commun. 35(8), 102–114 (1992)
17. 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)
18. Chaki, S., Gurfinkel, A., Strichman, O.: Decision Diagrams for Linear Arithmetic. In: FMCAD (2009)

A Decision Procedure for CTL* Based on Tableaux and Automata

Oliver Friedmann[1], Markus Latte[1], and Martin Lange[2]

[1] Dept. of Computer Science, University of Munich, Germany
[2] Dept. of Electrical Engineering and Computer Science, University of Kassel, Germany

Abstract. We present a decision procedure for the full branching-time logic CTL* which is based on tableaux with global conditions on infinite branches. These conditions can be checked using automata-theoretic machinery. The decision procedure then consists of a doubly exponential reduction to the problem of solving a parity game. This has advantages over existing decision procedures for CTL*, in particular the automata-theoretic ones: the underlying tableaux only work on subformulas of the input formula. The relationship between the structure of such tableaux and the input formula is given by very intuitive tableau rules. Furthermore, runtime experiments with an implementation of this procedure in the MLSolver tool show the practicality of this approach within the limits of the problem's computational complexity of being 2EXPTIME-complete.

1 Introduction

The full branching-time temporal logic CTL* is an important tool for the specification and verification of reactive systems [8] and of agent-based systems [11], for program synthesis [14], etc. Emerson and Halpern have introduced CTL* [3] as a formalism which supersedes both the branching-time logic CTL and the linear-time logic LTL. As much as this has led to an easy unification of CTL and LTL, it has also proved to be quite a difficulty in obtaining decision procedures for this logic. The first was automata-theoretic [5], requiring the determinisation of ω-word automata resulting from linear-time formulas. A series of improvements in this part has eventually led to Emerson and Jutla's automata-theoretic decision procedure [4] whose asymptotic worst-case running time is optimal, namely doubly exponential [20]. Other procedures have been given some time later, namely Reynolds' proof system [15], Gabbay and Pnueli's proof system [8], and most recently Reynolds' tableaux [16].

In this paper we present a characterisation of CTL* satisfiability. It is formulated as a calculus of infinite tableaux with natural rules and with global conditions on their branches. The non-termination of the tableaux raises the question after an effective decision procedure based on this calculus, and it is only here that we use automata-theoretic machinery. Branches violating the global condition are recognisable by nondeterministic Büchi automata, and we can then use determinisation and complementation to reduce the question of existence of a tableau to the problem of solving a doubly exponentially large parity game.

This also yields an asymptotically optimal decision procedure which has two distinct advantages over some of the existing ones. First, the tableaux only use subformulas of the input formula, while automata are only used on top of the tableaux in order to check the global conditions. Second, the reduction is implemented in the modal fixpoint solver MLSolver [6] which uses the high-performance parity game solver PGSolver [7] as a backend. The work presented here is therefore—to the best of our knowledge—the first serious attempt at creating a practical decision procedure for CTL*.

The rest of the paper is organised as follows. Sect. 2 recalls CTL*. Sect. 3 introduces the tableau calculus. Soundness and completeness are technically non-trivial to prove but still proceed along standard lines. The detailed proofs are omitted for lack of space and are given in an extended version of this paper. Sect. 4 presents a decision procedure which uses automata-theory in order to reduce the satisfiability problem to the problem of solving a parity game. Sect. 5 highlights the advantages of this approach in comparison to existing others. Sect. 6 reports on experimental results.

2 CTL*

Let $\mathcal{P}$ be a countably infinite set of propositional constants. A transition system is a tuple $\mathcal{T} = (\mathcal{S}, s^*, \to, \lambda)$ with $(\mathcal{S}, \to)$ being a directed graph, $s^* \in \mathcal{S}$ being a designated starting state and $\lambda : \mathcal{S} \to 2^{\mathcal{P}}$ is a labeling function. We assume transition systems to be total, i.e. every state has at least one successor. A *path* π in $\mathcal{T}$ is an infinite sequence of states $s_0, s_1, \dots$ s.t. $s_i \to s_{i+1}$ for all i. With π^k we denote the suffix of π starting with state s_k, and $\pi(k)$ denotes s_k in this case.

Branching-time temporal formulas are given by the following grammar.

$$\varphi \quad ::= \quad q \mid \neg\varphi \mid \varphi \wedge \varphi \mid \mathtt{X}\varphi \mid \varphi\mathtt{U}\varphi \mid \mathtt{E}\varphi$$

where $q \in \mathcal{P}$. Formulas of the form q or $\neg q$ are called *literals*. We use $\bar{\ell}$ to denote the complement of a literal ℓ, i.e. $\bar{\ell} = \neg q$ if $\ell = q$ and $\bar{\ell} = q$ if $\ell = \neg q$.

Other constructs like $\mathtt{tt}, \mathtt{ff}, \vee, \to$ are derived as usual, and so are the temporal ones $\varphi\mathtt{R}\psi := \neg(\neg\varphi\mathtt{U}\neg\psi)$, $\mathtt{G}\varphi = \mathtt{ff}\mathtt{R}\varphi$, $\mathtt{F}\varphi = \mathtt{tt}\mathtt{U}\varphi$, and $\mathtt{A}\varphi := \neg\mathtt{E}\neg\varphi$. A formula of this extended syntax is in *positive normal form* if $\neg$ only occurs in front of a propositional constant. The set of *subformulas* of φ is denoted by $Sub(\varphi)$ and defined as usual by setting $Sub(\varphi \circ \psi) := \{\varphi \circ \psi, \mathtt{X}(\varphi \circ \psi)\} \cup Sub(\varphi) \cup Sub(\psi)$ for $\circ$ being $\mathtt{U}$ or $\mathtt{R}$. The notation is extended to formula sets in the usual way. The size $|\varphi|$ of a formula φ is number of its subformulas. A *quantifier-bound formula block* is an $\mathtt{E}$- or $\mathtt{A}$-labeled set of formulas. We omit the braces for singleton sets. Formulas are interpreted over paths π of a transition systems $\mathcal{T} = (\mathcal{S}, s^*, \to, \lambda)$.

$$\begin{array}{lll}
\mathcal{T}, \pi \models q & \text{iff} & q \in \lambda(\pi(0)) \\
\mathcal{T}, \pi \models \neg\varphi & \text{iff} & \mathcal{T}, \pi \not\models \varphi \\
\mathcal{T}, \pi \models \varphi \wedge \psi & \text{iff} & \mathcal{T}, \pi \models \varphi \text{ and } \mathcal{T}, \pi \models \psi \\
\mathcal{T}, \pi \models \mathtt{X}\varphi & \text{iff} & \mathcal{T}, \pi^1 \models \varphi \\
\mathcal{T}, \pi \models \varphi\mathtt{U}\psi & \text{iff} & \exists k \in \mathbb{N}, \mathcal{T}, \pi^k \models \psi \text{ and } \forall j < k : \mathcal{T}, \pi^j \models \varphi \\
\mathcal{T}, \pi \models \mathtt{E}\varphi & \text{iff} & \exists \pi', \text{ s.t. } \pi'(0) = \pi(0) \text{ and } \mathcal{T}, \pi' \models \varphi
\end{array}$$

$$(\mathtt{A}\wedge)\ \frac{\mathtt{A}(\varphi,\Sigma),\mathtt{A}(\psi,\Sigma),\Phi}{\mathtt{A}(\varphi\wedge\psi,\Sigma),\Phi} \qquad (\mathtt{A}\vee)\ \frac{\mathtt{A}(\varphi,\psi,\Sigma),\Phi}{\mathtt{A}(\varphi\vee\psi,\Sigma),\Phi} \qquad (\mathtt{Al})\ \frac{\ell,\Phi \quad | \quad \mathtt{A}\Sigma,\Phi}{\mathtt{A}(\ell,\Sigma),\Phi}$$

$$(\mathtt{AU})\ \frac{\mathtt{A}(\psi,\varphi,\Sigma),\mathtt{A}(\psi,\mathtt{X}(\varphi\mathtt{U}\psi),\Sigma),\Phi}{\mathtt{A}(\varphi\mathtt{U}\psi,\Sigma),\Phi} \qquad (\mathtt{AR})\ \frac{\mathtt{A}(\psi,\Sigma),\mathtt{A}(\varphi,\mathtt{X}(\varphi\mathtt{R}\psi),\Sigma),\Phi}{\mathtt{A}(\varphi\mathtt{R}\psi,\Sigma),\Phi}$$

$$(\mathtt{AA})\ \frac{\mathtt{A}\varphi,\Phi \quad | \quad \mathtt{A}\Sigma,\Phi}{\mathtt{A}(\mathtt{A}\varphi,\Sigma),\Phi} \qquad (\mathtt{AE})\ \frac{\mathtt{E}\varphi,\Phi \quad | \quad \mathtt{A}\Sigma,\Phi}{\mathtt{A}(\mathtt{E}\varphi,\Sigma),\Phi} \qquad (\mathtt{Ett})\ \frac{\Phi}{\mathtt{E}\emptyset,\Phi}$$

$$(\mathtt{E}\vee)\ \frac{\mathtt{E}(\varphi,\Pi),\Phi \quad | \quad \mathtt{E}(\psi,\Pi),\Phi}{\mathtt{E}(\varphi\vee\psi,\Pi),\Phi} \qquad (\mathtt{E}\wedge)\ \frac{\mathtt{E}(\varphi,\psi,\Pi),\Phi}{\mathtt{E}(\varphi\wedge\psi,\Pi),\Phi} \qquad (\mathtt{El})\ \frac{\mathtt{E}\Pi,\ell,\Phi}{\mathtt{E}(\ell,\Pi),\Phi}$$

$$(\mathtt{EU})\ \frac{\mathtt{E}(\psi,\Pi),\Phi \quad | \quad \mathtt{E}(\varphi,\mathtt{X}(\varphi\mathtt{U}\psi),\Pi),\Phi}{\mathtt{E}(\varphi\mathtt{U}\psi,\Pi),\Phi} \qquad (\mathtt{EE})\ \frac{\mathtt{E}\varphi,\mathtt{E}\Pi,\Phi}{\mathtt{E}(\mathtt{E}\varphi,\Pi),\Phi}$$

$$(\mathtt{ER})\ \frac{\mathtt{E}(\psi,\varphi,\Pi),\Phi \quad | \quad \mathtt{E}(\psi,\mathtt{X}(\varphi\mathtt{R}\psi),\Pi),\Phi}{\mathtt{E}(\varphi\mathtt{R}\psi,\Pi),\Phi} \qquad (\mathtt{EA})\ \frac{\mathtt{A}\varphi,\mathtt{E}\Pi,\Phi}{\mathtt{E}(\mathtt{A}\varphi,\Pi),\Phi}$$

$$(\mathtt{X}_0)\ \frac{\mathtt{A}\Sigma_1,\dots,\mathtt{A}\Sigma_m}{\mathtt{AX}\Sigma_1,\dots,\mathtt{AX}\Sigma_m,\Lambda} \qquad (\mathtt{X}_1)\ \frac{\mathtt{E}\Pi_1,\mathtt{A}\Sigma_1,\dots,\mathtt{A}\Sigma_m \quad \dots \quad \mathtt{E}\Pi_n,\mathtt{A}\Sigma_1,\dots,\mathtt{A}\Sigma_m}{\mathtt{EX}\Pi_1,\dots,\mathtt{EX}\Pi_n,\mathtt{AX}\Sigma_1,\dots,\mathtt{AX}\Sigma_m,\Lambda}$$

Fig. 1. The pre-tableau rules for CTL*

Two formulas φ and ψ are equivalent, written $\varphi \equiv \psi$, if for all paths π of all transition systems $\mathcal{T}$: $\mathcal{T},\pi \models \varphi$ iff $\mathcal{T},\pi \models \psi$. It is well-known and easy to see that every formula is equivalent to one in positive normal form. A formula φ is called a *state formula* if for all $\mathcal{T},\pi,\pi'$ with $\pi(0)=\pi'(0)$ we have $\mathcal{T},\pi \models \varphi$ iff $\mathcal{T},\pi' \models \varphi$. Hence, satisfaction of a state formula in a path only depends on the first state of the path. Note that φ is a state formula iff $\varphi \equiv \mathtt{E}\varphi$. For state formulas we also write $\mathcal{T},s \models \varphi$ for $s \in \mathcal{S}$. CTL* is the set of all branching-time formulas which are state formulas. A CTL* formula φ is *satisfiable* if there is a transition system $\mathcal{T}$ with an initial state s^* s.t. $\mathcal{T},s^* \models \varphi$.

3 Tableaux for CTL*

From now on, formulas are assumed to be in positive normal form. We will construct a tableau for a given state formula ϑ. The following notations are used: Σ and Π are finite (possibly empty) sets of formulas with Σ being interpreted as a disjunction of formulas and Π as a conjunction. We write Λ for a set of literals. For a set of formulas Γ let $\mathtt{X}\Gamma := \{\mathtt{X}\psi \mid \psi \in \Gamma\}$. A *goal (for ϑ)* is a non-empty set—the outermost braces are omitted—of the form $\mathtt{A}\Sigma_1,\dots,\mathtt{A}\Sigma_n,\mathtt{E}\Pi_1,\dots,\mathtt{E}\Pi_m,\Lambda$ where $n,m \geq 0$, and $\Sigma_1,\dots,\Sigma_n,\Pi_1,\dots,\Pi_m,\Lambda$ are subsets of $Sub(\vartheta)$. Such a goal stands for the state formula $\bigwedge_{i=1}^{n} \mathtt{A}\big(\bigvee_{\psi\in\Sigma_i}\psi\big) \wedge \bigwedge_{i=1}^{m} \mathtt{E}\big(\bigwedge_{\psi\in\Pi_i}\psi\big) \wedge \bigwedge_{\ell\in\Lambda}\ell$. Goals are denoted by $\mathcal{C}$. We write $Seq(\vartheta)$ for the set of all possible goals for ϑ. Note that this is a finite set of at most doubly exponential size in $|\vartheta|$. A goal $\mathcal{C}$ is *consistent* if there is no $q \in \mathcal{P}$ s.t. $q \in \mathcal{C}$ and $\neg q \in \mathcal{C}$.

Definition 1. *A* pre-tableau *for ϑ is a possibly infinite tree built according to the rules of Fig. 1 whose root is $\mathtt{E}\vartheta$, whose nodes are all consistent and do not contain $\mathtt{A}\emptyset$, and whose leaves consist of literals only.*

We write pre-tableaux as trees growing upwards. Consequently, a rule in Fig. 1 has a goal at the bottom and one or several subgoals at the top. Letter ℓ stands for arbitrary literals. Rule $(\mathtt{X}_1)$ is the only rule with more than one subgoal, rules $(\mathtt{Al})$, $(\mathtt{AA})$, $(\mathtt{AE})$, $(\mathtt{E}\vee)$, $(\mathtt{EU})$, $(\mathtt{ER})$ each have a single subgoal which can be chosen nondeterministically to be of either of two forms.

An occurrence of a formula is called *principal* if it gets transformed by a rule. For example, the occurrence of $\varphi \wedge \psi$ is principal in $(\mathtt{E}\wedge)$. A principal formula has *descendants* in the subgoals. For example, both occurrences of φ and ψ are descendants of the principal $\varphi \wedge \psi$ in rule $(\mathtt{E}\wedge)$.

Note that, in the modal rules $(\mathtt{X}_0)$ and $(\mathtt{X}_1)$, every formula apart from those in the literal part is principal. Literals in the literal part can never be principal, but literals in an $\mathtt{A}$- or $\mathtt{E}$-block are principal in rules $(\mathtt{Al})$ and $(\mathtt{El})$. Finally, any non-principal occurrence of a formula in a goal may have a *copy* in one of the subgoals. The copy is the same formula since it has not been transformed. For instance, any formula in Σ in rule $(\mathtt{Al})$ has a copy in the subgoal if it is of the right form, but does not have a copy if it is of the left form.

A quantifier-bound block $\mathtt{A}\Sigma$ or $\mathtt{E}\Pi$ is called *principal* as well if it contains a principal formula, and possibly has descendants in the subgoal(s). For example, $\mathtt{A}(\varphi \wedge \psi, \Sigma)$ has two descendants $\mathtt{A}(\varphi, \Sigma)$ and $\mathtt{A}(\psi, \Sigma)$ in an application of $(\mathtt{A}\wedge)$.

Definition 2. *Let $\mathcal{C}$ be a goal to which a rule r is applicable and let $\mathcal{C}'$ be one of its subgoals. Furthermore, let $\mathtt{Q}_1\Delta_1$, resp. $\mathtt{Q}_2\Delta_2$ with $\mathtt{Q}_1, \mathtt{Q}_2 \in \{\mathtt{E}, \mathtt{A}\}$ and $\Delta_1, \Delta_2 \subseteq Sub(\vartheta)$ be quantifier-bound blocks occurring in the $\mathtt{A}$- or $\mathtt{E}$-part of $\mathcal{C}$, resp. $\mathcal{C}'$. We say that $\mathtt{Q}_1\Delta_1$ is* connected *to $\mathtt{Q}_2\Delta_2$ in $\mathcal{C}$ and $\mathcal{C}'$, if either*

- *$\mathtt{Q}_1\Delta_1$ is principal in r, and $\mathtt{Q}_2\Delta_2$ is one of its descendants in $\mathcal{C}'$; or*
- *$\mathtt{Q}_1\Delta_1$ is not principal in r and $\mathtt{Q}_2\Delta_2$ is a copy of $\mathtt{Q}_1\Delta_1$ in $\mathcal{C}'$.*

We write this as $(\mathcal{C}, \mathtt{Q}_1\Delta_1) \rightsquigarrow (\mathcal{C}', \mathtt{Q}_2\Delta_2)$. If the rule instance can be inferred from the context we may also simply write $\mathtt{Q}_1\Delta_1 \rightsquigarrow \mathtt{Q}_2\Delta_2$. Additionally, let ψ, resp. ψ' be a formula occurring in Δ_1, resp. Δ_2. We say that ψ is connected *to ψ' in $(\mathcal{C}, \mathtt{Q}_1\Delta_1)$ and $(\mathcal{C}', \mathtt{Q}_2\Delta_2)$, if either*

- *ψ is principal in r, and ψ' is one of its descendants in $\mathcal{C}'$; or*
- *ψ is not principal in r and ψ' is a copy of ψ in $\mathcal{C}'$.*

We write this as $(\mathcal{C}, \mathtt{Q}_1\Delta_1, \psi) \rightsquigarrow (\mathcal{C}', \mathtt{Q}_2\Delta_2, \psi')$. If the rule instance can be inferred from the context we may also simply write $(\mathtt{Q}_1\Delta_1, \psi) \rightsquigarrow (\mathtt{Q}_2\Delta_2, \psi')$. A block connection $(\mathcal{C}_1, \mathtt{Q}_1\Delta_1) \rightsquigarrow (\mathcal{C}_2, \mathtt{Q}_2\Delta_2)$ is called spawning *iff $\mathtt{Q}_2\psi \in \Delta_1$ is principal and $\Delta_2 = \{\psi\}$. The only rules that possibly induce a spawning block connection are $(\mathtt{EE})$, $(\mathtt{EA})$, $(\mathtt{AA})$ and $(\mathtt{AE})$.*

Definition 3. *Let $\mathcal{C}_0, \mathcal{C}_1, \ldots$ be an infinite branch of a pre-tableau t for some Φ. A* trace *Ξ in this branch is an infinite sequence $\mathtt{Q}_0\Delta_0, \mathtt{Q}_1\Delta_1, \ldots$ s.t. for all $i \in \mathbb{N}$:*

$(\mathcal{C}_i, \mathtt{Q}_i\Delta_i) \rightsquigarrow (\mathcal{C}_{i+1}, \mathtt{Q}_{i+1}\Delta_{i+1})$. *A* trace Ξ *is called an* $\mathtt{E}$*-trace, resp.* $\mathtt{A}$*-trace if there is an* $i \in \mathbb{N}$ *s.t.* $\mathtt{Q}_j = \mathtt{E}$*, resp.* $\mathtt{Q}_j = \mathtt{A}$ *for all* $j \geq i$*. We say that a trace is* finitely spawning *if it contains only finitely many spawning block connections.*

Lemma 4. *Every infinite branch of a pre-tableau contains infinitely many applications of rules* $(\mathtt{X}_0)$ *or* $(\mathtt{X}_1)$*.*

Proof. Assume by contradiction that there is an infinite branch $\mathcal{C}_0, \mathcal{C}_1, \ldots$ in a pre-tableau for some Φ that contains only finitely many applications of the modal rules. Note that there must be a trace $\mathtt{Q}_0\Delta_0, \mathtt{Q}_1\Delta_1, \ldots$ in the branch that is principal infinitely often. Now we consider a lexicographic measure on all the Δ_i that counts how many non-modal formulas of a certain depth are contained in the block. Note that every rule application except the modal rule decreases this measure. But this cannot be the case. □

Definition 5. *A* thread t *in a trace* $\Xi = \mathtt{Q}_0\Delta_0, \mathtt{Q}_1\Delta_1, \ldots$ *is an infinite sequence* $\psi_0, \psi_1, \ldots$ *s.t. for all* $i \in \mathbb{N}$*:* $(\mathcal{C}_i, \mathtt{Q}_i\Delta_i, \psi_i) \rightsquigarrow (\mathcal{C}_{i+1}, \mathtt{Q}_{i+1}\Delta_{i+1}, \psi_{i+1})$*. Such a thread* t *is called a* $\mathtt{U}$*-thread, resp.* $\mathtt{R}$*-thread if there is a formula* $\varphi\mathtt{U}\psi \in Sub(\Phi)$*, resp.* $\varphi\mathtt{R}\psi \in Sub(\Phi)$ *s.t.* $\psi_j = \varphi\mathtt{U}\psi$*, resp.* $\psi_j = \varphi\mathtt{R}\psi$ *for infinitely many* j*.*

An $\mathtt{E}$*-trace is called* good *iff it has no* $\mathtt{U}$*-thread; similarly, an* $\mathtt{A}$*-trace is called* good *iff it has an* $\mathtt{R}$*-thread.*

This immediately yields the definition of a *bad* trace: an $\mathtt{E}$-trace is bad if it contains an $\mathtt{U}$-thread, and an $\mathtt{A}$-trace is bad if it contains no $\mathtt{R}$-thread.

Lemma 6. *Every trace in an infinite branch of a pre-tableau is either an* $\mathtt{A}$*-trace or an* $\mathtt{E}$*-trace and only finitely spawning.*

Proof. Let $\Xi = \mathtt{Q}_0\Delta_0, \mathtt{Q}_1\Delta_1, \ldots$ be a trace and assume by contradiction that $\{i \mid \mathtt{Q}_i\Delta_i \rightsquigarrow \mathtt{Q}_{i+1}\Delta_{i+1}$ is spawning$\}$ is infinite. Let $i_0 < i_1 < \ldots$ be the ascending sequence of numbers in this infinite set and let ϕ_{i_j} denote the formula in the singleton set Δ_{i_j+1}. Note that for all j it is the case that $\phi_{i_{j+1}} \in Sub(\phi_{i_j})$ and $\phi_{i_j} \neq \phi_{i_{j+1}}$, hence the set cannot be infinite. Now note that every finitely spawning trace eventually must be either an $\mathtt{A}$- or an $\mathtt{E}$-trace. □

Lemma 7. *Every thread in a trace of an infinite branch of a pre-tableau is either an* $\mathtt{U}$*- or an* $\mathtt{R}$*-thread.*

Proof. Let $t = \psi_0, \psi_1, \ldots$ be a thread. Assume that t is neither an $\mathtt{U}$- nor an $\mathtt{R}$-thread, hence there is a position i^* s.t. ψ_i is neither of the form $\psi'\mathtt{U}\psi''$ nor of the form $\psi'\mathtt{R}\psi''$ for all $i \geq i^*$, hence $\psi_{i+1} \in Sub(\psi_i)$ for all $i \geq i^*$. By Lemma 4 it follows that $\psi_{i+1} \neq \psi_i$ for infinitely many i which cannot be the case, hence t has to be a $\mathtt{U}$- or an $\mathtt{R}$-thread. Finally, assume that t is both an $\mathtt{U}$- and an $\mathtt{R}$-thread, i.e. there are positions $i_0 < i_1 < i_2$ s.t. $\psi_{i_0} = \psi_{i_2} = \psi'\mathtt{R}\psi''$ and $\psi_{i_1} = \varphi'\mathtt{U}\varphi''$. Hence $\psi_{i_1} \in Sub(\psi_{i_0}) \setminus \{\psi_{i_0}\}$ and $\psi_{i_2} \in Sub(\psi_{i_1}) \setminus \{\psi_{i_1}\}$, thus $\psi'\mathtt{R}\psi'' \in Sub(\psi'\mathtt{R}\psi'') \setminus \{\psi'\mathtt{R}\psi''\}$ which cannot be the case. □

Lemma 8. *For every* $\mathtt{U}$*- and every* $\mathtt{R}$*-thread (in a trace of an infinite branch of a pre-tableau)* $\psi_0, \psi_1, \ldots$ *there is an* $i \in \mathbb{N}$ *such that* ψ_i *is an* $\mathtt{U}$*-, or an* $\mathtt{R}$*-formula resp., and* $\psi_j = \psi_i$ *or* $\psi_j = \mathtt{X}\psi_i$ *for all* $j \geq i$*.*

Proof. For all $i \in \mathbb{N}$, it holds that $\psi_{i+1} \in Sub(\psi_i)$, or $\psi_{i+1} = \mathtt{X}\psi_i$ provided that ψ_i is an U- or an R-formula. The map removing from a formula its frontal X converts the thread into a chain which is weakly decreasing with respect to the subformula order. Because this order is well-founded the claim follows. □

Definition 9. *A* tableau *for ϑ is a pre-tableau for ϑ that does not contain a branch which contains a bad trace.*

In other words, all traces in a tableau must be good. Such tableaux exactly characterise satisfiability of CTL* in the sense of the following theorem.

Theorem 10. *For all $\vartheta \in$ CTL*: ϑ is satisfiable iff there is a tableau for ϑ.*

The completeness proof is technically tedious but does not use any heavy machinery once the right invariants etc. are being found. Given a model for ϑ we use this to construct a pre-tableau in a certain way. Then assume that the result is not a tableau and derive a contradiction from it. Soundness can be shown by collapsing a tableau into a tree-like transition system and verifying that it is indeed a model of ϑ.

4 A Decision Procedure for CTL*

4.1 Using Automata to Recognise Tableau Branches

The main difficulty in deciding the existence of a tableau for a formula φ is the global condition on infinite branches being required to be good. We propose to use automata-theory for this. Pre-tableau branches can be represented as infinite words over a certain alphabet, and we will show that the language of good branches is recognisable by a nondeterministic Büchi-automaton (NBA). This is not trivial since a nondeterministic machine cannot easily check for absence of U-threads in E-traces for instance. However, we can use the nondeterminism in order to check for violations, i.e. the presence of U-threads in an E-trace for instance. This then has to be complemented and combined with something checking for the presence of an R-thread in an A-trace in order to have a device recognising exactly the set of good paths of a tableau.

The goal is then to replace the global condition on branches of having good traces by an annotation of the tableau nodes with automaton states and a global condition on these states. For instance, if the resulting automaton was of Büchi type, then a tableau can be seen as a pre-tableau with nodes annotated by the automaton s.t. on every infinite path, infinitely many final states occur.

Now note that the automaton recognising good paths needs to be deterministic: suppose there are two branches uv and uw with a common prefix u s.t. both branches are recognised by $\mathcal{A}$. If $\mathcal{A}$ is nondeterministic then it may have two different accepting runs on uv and uw that differ on the common prefix u already. Remember that an annotation of tableau nodes with a single automaton state is required. However, this is possible if $\mathcal{A}$ is deterministic.

The problem of deciding existence of a tableau then reduces to the problem of solving a game. Its nodes are pre-tableau nodes annotated with states of the deterministic automaton. Nondeterministic choices in the tableau rules translate into choices of the existential player in the game; the branching rules $(\mathtt{X}_0)$ and $(\mathtt{X}_1)$ translate into choices of the universal player. The type of the game is the same as the type of $\mathcal{A}$. For instance, if $\mathcal{A}$ is a deterministic parity automaton then the game is a parity game.

Here we are particularly interested in Büchi and parity automata [9]. An NBA is a tuple $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ with Q being a finite set of *states*, Σ a finite *alphabet*, $q_0 \in Q$ an *initial state*, $\delta \subseteq Q \times \Sigma \times Q$ the *transition relation* and $F \subseteq Q$ a set of *final states*. A *run* of $\mathcal{A}$ on a $a_0 a_1 \ldots \in \Sigma^\omega$ is an infinite sequence $q_0, q_1, \ldots$ s.t. $(q_i, a_i, q_{i+1}) \in \delta$ for all $i \in \mathbb{N}$. It is accepting if $q_i \in F$ for infinitely many i. The *language* of the NBA $\mathcal{A}$ is $L(\mathcal{A}) = \{w \mid \text{there is an accepting run of } \mathcal{A} \text{ on } w\}$. A *co-Büchi automaton* (NcoBA) is syntactically the same as a NBA. However, a run $q_0, q_1, \ldots$ of an NcoBA is accepting if it only contains finitely many non-final states. A *parity automaton* (NPA) is a tuple $\mathcal{A} = (Q, \Sigma, q_0, \delta, \Omega)$ with Q, Σ, q_0, δ as above and $\Omega : Q \to \mathbb{N}$ assigns to each state a *priority*. A run $q_0, q_1, \ldots$ is accepting if $\max\{\Omega(q) \mid q = q_i \text{ for infinitely many } i \in \mathbb{N}\}$ is even. The *index* of an NPA $\mathcal{A}$ is the number of different priorities occurring, i.e. $|\Omega[Q]|$.

An NBA / NcoBA / NPA with transition relation δ is *deterministic* (DBA / DcoBA / DPA) if $|\{q' \mid (q, a, q') \in \delta\}| = 1$ for all $q \in Q$ and $a \in \Sigma$. Determinism and the duality between Büchi and co-Büchi condition as well as the self-duality of the parity acceptance condition makes it easy to complement a DcoBA to a DBA as well as a DPA to a DPA again. The following is a standard and straight-forward result [9, Sec. 1.2] in the theory of ω-word automata.

Lemma 11. *For every DcoBA, resp. DPA, $\mathcal{A}$ there is a DBA, resp. DPA, $\overline{\mathcal{A}}$ with $L(\overline{\mathcal{A}}) = \overline{L(\mathcal{A})}$ and $|\overline{\mathcal{A}}| = |\mathcal{A}|$.*

4.2 Automata for Tableau Branches

We regard rule applications—more precisely: pairs of a goal and one of its subgoals in one of the tableau rules—in a pre-tableau for a formula φ as symbols of a finite alphabet. Naïvely, this would yield an alphabet of doubly exponential size since there are doubly exponentially many different goals. However, note that such a pair is entirely determined by the principal block and the principal formula of the goal and a number specifying the subgoal. This enables a smaller symbolic encoding. For instance, the transition from the goal $\mathtt{A}(\mathtt{E}\varphi, \Sigma), \Phi$ to the subgoal $\mathtt{A}\Sigma, \Phi$ in rule $(\mathtt{AE})$ would be represented by the quadruple $(\mathtt{A}, \{\mathtt{E}\varphi\} \cup \Sigma, \mathtt{E}\varphi, 1)$. The other possible premiss would have index 0 instead. There are three exceptions to this: applications of rules $(\mathtt{Ett})$ and $(\mathtt{X}_0)$ can be represented using a constant name, and the premiss in rule $(\mathtt{X}_1)$ is entirely determined by one of the $\mathtt{E}$-blocks in the subgoal. Hence, let

$$\Sigma^{\mathsf{br}}_\varphi \;:=\; (\{\mathtt{A}, \mathtt{E}\} \times 2^{Sub(\varphi)} \times Sub(\varphi) \times \{0, 1\}) \;\cup\; \{0, 1\} \;\cup\; 2^{Sub(\varphi)}$$

Note that $|\Sigma^{\mathsf{br}}_\varphi| = 2^{\mathcal{O}(|\varphi|)}$.

An infinite branch $\pi = \mathcal{C}_0, \mathcal{C}_1, \ldots$ in a pre-tableau for φ then induces a word $\pi' = r_0, r_1, \ldots \in (\Sigma^{\mathsf{br}}_\varphi)^\omega$ in a straight-forward way: r_i is the symbolic representation of the goal/subgoal pair (C_i, C_{i+1}). We will not distinguish formally between an infinite branch π and its induced ω-word π' over $\Sigma^{\mathsf{br}}_\varphi$.

Remember that we want to define an NBA which accepts exactly those branches which are not good, i.e. which either contain an $\mathtt{E}$-trace with an $\mathtt{U}$-thread or an $\mathtt{A}$-trace with no $\mathtt{R}$-thread. Nondeterminism can be used in order to guess the trace in the branch, and it can also be used in order to guess an $\mathtt{U}$-thread in an $\mathtt{E}$-trace. However, it is not necessarily useful for showing that no $\mathtt{R}$-thread exists in an $\mathtt{A}$-trace. We therefore use complementation for this subproblem again.

An $\mathtt{A}$-trace-marked branch is a pre-tableau branch in which a single $\mathtt{A}$-trace is marked. It can be represented as an infinite word over the alphabet $\Sigma^{\mathsf{tmb}}_\varphi = \Sigma^{\mathsf{br}}_\varphi \times 2^{Sub(\varphi)}$. The second component of the alphabet simply names the set of subformulas which form the current $\mathtt{A}$-block on the marked trace. Then we define a co-Büchi automaton $\mathcal{C}_\varphi$ which recognises exactly those $\mathtt{A}$-trace-marked branches which contain an $\mathtt{R}$-thread in the marked trace. It is $\mathcal{C}_\varphi = (\{\mathtt{W}, \mathtt{F}\} \cup Sub(\varphi), \Sigma^{\mathsf{tmb}}_\varphi, \mathtt{W}, \delta, F)$ with $F = Sub(\varphi)$. We define the transition relation δ by intuitively describing its behaviour. Starting in the waiting state $\mathtt{W}$ it guesses a formula of the form $\psi_1 \mathtt{R} \psi_2$ which occurs in the marked $\mathtt{A}$-trace. It then tracks this formula in its state for as long as it is unfolded with rule ($\mathtt{AR}$) and remains in the marked trace. If it leaves the marked trace then $\mathcal{C}_\varphi$ moves into the failure state $\mathtt{F}$. The following proposition is easily seen to be true.

Lemma 12. *Let $w \in (\Sigma^{\mathsf{tmb}}_\varphi)^\omega$ be an $\mathtt{A}$-trace-marked branch of a pre-tableau for φ. Then $w \in \mathcal{L}(\mathcal{C}_\varphi)$ iff the marked trace of w contains a $\mathtt{R}$-thread.*

Remember that we are interested in branches whose $\mathtt{A}$-traces do not contain $\mathtt{R}$-threads. Hence, we need complementation. Luckily, an NcoBA can be determinised into a DcoBA using the Miyano-Hayashi construction [12] which can easily be complemented into a DBA according to Lemma 11.

Theorem 13 ([12]). *For every NcoBA $\mathcal{A}$ with n states there is a DBA $\overline{\mathcal{A}}$ with at most 3^n states s.t. $L(\overline{\mathcal{A}}) = \overline{L(\mathcal{A})}$.*

Equally, we can define an $\mathtt{E}$-trace-marked branch as a word over $\Sigma^{\mathsf{tmb}}_\varphi$ and an NcoBA $\mathcal{B}_\varphi$ which accepts exactly those which contain an $\mathtt{U}$-thread in the marked $\mathtt{E}$-trace. It is $\mathcal{B}_\varphi := (\{\mathtt{W}, \mathtt{F}\} \cup Sub(\varphi), \Sigma^{\mathsf{tmb}}_\varphi, \mathtt{W}, \delta, F)$ with $F = Sub(\varphi)$. Its behaviour is almost the same as that of $\mathcal{C}_\varphi$ with the difference that it tracks an $\mathtt{U}$-formula in its state component rather than an $\mathtt{R}$-formula.

Lemma 14. *Let $w \in (\Sigma^{\mathsf{tmb}}_\varphi)^\omega$ be an $\mathtt{E}$-trace-marked branch of a pre-tableau for φ. Then $w \in \mathcal{L}(\mathcal{B}_\varphi)$ iff the marked trace of w contains an $\mathtt{U}$-thread.*

Then we can define an NBA $\mathcal{A}_\varphi$ that accepts exactly those branches which contain a bad trace. Let $\overline{\mathcal{C}_\varphi} = (Q^{\mathcal{C}}, \Sigma^{\mathsf{tmb}}_\varphi, q^{\mathcal{C}}_0, \delta^{\mathcal{C}}, F^{\mathcal{C}})$ be the DBA obtained from $\mathcal{C}_\varphi$ using the complementation construction of Thm. 13, and $\mathcal{B}_\varphi = (Q^{\mathcal{B}}, \Sigma^{\mathsf{tmb}}_\varphi, q^{\mathcal{B}}_0, \delta^{\mathcal{B}}, F^{\mathcal{B}})$. Then define $\mathcal{A}_\varphi := (Q, \Sigma^{\mathsf{br}}_\varphi, \mathtt{W}, \delta, F)$ where $Q = \{\mathtt{W}, \mathtt{F}\} \cup 2^{Sub(\varphi)} \times \big(Q^{\mathcal{C}} \mathbin{\dot\cup} Q^{\mathcal{B}}\big)$. Again, we describe its behaviour informally. It starts in the waiting state $\mathtt{W}$. At some point it guesses a block that is contained in the given alphabet symbol and tracks this block in the first component of its state space in order to check that it is a non-spawing trace. Depending on whether or not it is an `A`- or `E`-block it simulates in its second component the automaton $\overline{\mathcal{C}_\varphi}$, resp. $\mathcal{B}_\varphi$ on the letters which are composed of the input letter and the first component. Thus, it effectively guesses a trace and simulates one of the two automata on the branch in which this trace is marked. If the trace disappears using rule (`Ett`) for instance, it moves to the failure state `F`. Its accepting states F are $2^{Sub(\varphi)} \times (F^{\mathcal{C}} \dot{\cup} F^{\mathcal{B}})$. The following is not too difficult to see using Lem. 12 and 14 as well as Thm. 13.

Lemma 15. *Let $w \in (\Sigma_\varphi^{\mathsf{br}})^\omega$ be a branch of a pre-tableau for φ. Then $w \in \mathcal{L}(\mathcal{A}_\varphi)$ iff w contains a trace which is not good.*

Furthermore, a close inspection of the constructions together which Thm. 13 yields the following estimation on the size of $\mathcal{A}_\varphi$. Note that the initial waiting states of $\mathcal{C}_\varphi$ and $\mathcal{B}_\varphi$ are redundant since waiting is also done in the initial state of $\mathcal{A}_\varphi$.

Proposition 16. *The number of states of $\mathcal{A}_\varphi$ is bounded by $2 + 2^{|\varphi|} \cdot (3^{|\varphi|+2} \cdot (|\varphi| + 2)) \le 2^{\mathcal{O}(|\varphi|)}$.*

Finally, remember that we need a deterministic automaton recognising the complement of the language recognised by $\mathcal{A}_\varphi$. Luckily, there are determinisation constructions for Büchi automata. We are particularly interested in those that yield parity automata [13,10,18].

Theorem 17 ([13]). *For every NBA with n states there is an equivalent DPA with at most n^{2n+2} states and index at most $2n - 1$.*

Together with Lemma 11 we obtain a DPA $\overline{\mathcal{A}}_\varphi$ which accepts exactly those branches containing good traces only, and has size $2^{2^{\mathcal{O}(|\varphi|)}}$ and index $2^{\mathcal{O}(|\varphi|)}$.

4.3 The Reduction to Parity Games

The problem of deciding the existence of a tableau can easily be phrased as a game: starting with the initial goal $\mathtt{E}\varphi$, the *proponent* chooses a rule instance that can be applied to the current goal, and the *opponent* chooses a subgoal whenever the instance is a branching rule. Note this is only the case for the modal rules. This yields a pre-tableau branch in the limit. The proponent wins iff all traces on this branch are good, otherwise the opponent wins. Clearly, there is a tableau for φ iff the proponent has a winning strategy in this game. We will now use the automata-theoretic machinery of the previous subsection in order to formalise this game and present a reduction of the satisfiability problem for CTL* to the problem of solving a parity game.

A *parity game* is a $\mathcal{G} = (V, V_0, V_1, v_0, E, \Omega)$ s.t. (V, E) is a finite, directed graph with total edge relation E, V_0, V_1 is a partition of the node set V into nodes owned by player 0 and 1, resp., $v_0 \in V$ is a designated starting node, and $\Omega : V \to \mathbb{N}$ assigns priorities to the nodes. A *play* is an infinite sequence $v_0, v_1, \ldots$ starting in v_0 s.t. $(v_i, v_{i+1}) \in E$ for all $i \in \mathbb{N}$. It is won by player 0 if $\max\{\Omega(v) \mid$

$v = v_i$ for infinitely many $i\}$ is even. A *(non-positional) strategy* for player i is a function $\sigma : V^*V_i \to V$, s.t. for all sequences $v_0 \ldots v_n$ with $(v_j, v_{j+1}) \in E$ for all $j = 0, \ldots, n-1$, and all $v_n \in V_i$ we have: $(v_n, \sigma(v_0 \ldots v_n)) \in E$. A play $v_0 v_1 \ldots$ *conforms* to a strategy σ for player i if for all $j \in \mathbb{N}$ we have: if $v_j \in V_i$ then $v_{j+1} = \sigma(v_0 \ldots v_j)$. A strategy σ for player i is a *winning strategy* in node v if player i wins every play that begins in v and conforms to σ. A *(positional) strategy* for player i is a strategy σ for player i s.t. for all $v_0 \ldots v_n \in V^*V_i$ and all $w_0 \ldots w_m \in V^*V_i$ we have: if $v_n = w_m$ then $\sigma(v_0 \ldots v_n) = \sigma(w_0 \ldots w_m)$. Hence, we can identify positional strategies with $\sigma : V_i \to V$. It is a well-known fact that for every node $v \in V$, there is a winning strategy for either player 0 or player 1 for node v. In fact, parity games enjoy positional determinancy meaning that there is even a positional winning strategy for node v for one of the two player [1]. The problem of *solving* a parity game is to determine which player has a winning strategy for v_0. It is solvable [17] in time polynomial in $|V|$ and exponential in $|\Omega[V]|$.

Definition 18. *Let φ be a state formula and $\overline{\mathcal{A}}_\varphi = (Q, \Sigma^{\mathsf{br}}_\varphi, q_0, \delta, \Omega)$ be the DPA according to the previous subsection which recognises good branches in pre-tableaux for φ. The* satisfiability game *for φ is a parity game $\mathcal{G}_\varphi = (V, V_0, V_1, v_0, E, \Omega')$ defined as follows.*

- $V := \mathit{Seq}(\varphi) \times Q$
- $V_1 := \{(C, q) \in V \mid \textit{rule } (\mathtt{X}_0) \textit{ or } (\mathtt{X}_1) \textit{ applies to } C\}$
- $V_0 := V \setminus V_1$
- $v_0 := (\mathtt{E}\varphi, q_0)$
- $((C, q), (C', q')) \in E$ *iff (C, C') is an instance of a rule application which is symbolically represented by $r \in \Sigma^{\mathsf{br}}_\varphi$ and $q' = \delta(q, r)$, or no rule is applicable to C and $C = C'$ and $q = q'$,*
- $\Omega'(C, q) := \begin{cases} 0 & \textit{if } C \textit{ is a consistent set of literals} \\ \Omega(q) & \textit{if there is a rule applicable to } C \\ 1 & \textit{otherwise} \end{cases}$

The following theorem states correctness of this construction. It is not difficult to prove. In fact, a winning strategy for player 0 is basically a finite representation of an infinite tableau.

Theorem 19. *Player* 0 *wins $\mathcal{G}_\varphi$ iff there is a tableau for φ.*

Proof. Assume that player 0 wins $\mathcal{G}_\varphi$ with a positional winning strategy σ. Unfolding the game $\mathcal{G}_\varphi$ starting with v^* and conforming to σ results in a possibly infinite tree that can be easily transformed into a pre-tableau P for φ by removing all annotations of the branch-checking automaton and by replacing all consistent-set-loops with consistent-set-leafs. Note that it is impossible that a finite branch does not end in a consistent set with player 0 winning from v^*. Given an arbitrary infinite branch π in P, it holds that $\pi \in \mathcal{L}(\overline{\mathcal{A}}_\varphi)$, hence by Lemma 15 it follows that π contains no bad trace. Consequently, P is a tableau.

For the other direction, let P be a tableau for φ. Starting with v^*, every goal in P can be labeled with the corresponding game state; then, every player 0 position of $\mathcal{G}_\varphi$ corresponding to a node in the labeled version of P can be used as a non-positional strategy decision for player 0. The player 0 strategy obtained in that manner is indeed a winning strategy for player 0 starting in v^*: let π be an arbitrary play conforming to the strategy; if π is finite, it must correspond to a branch in P that ends in a consistent set of literals, hence it is won by player 0, otherwise the branch corresponding to π contains only good traces, and hence by Lemma 15 it follows that π is won by player 0. $\square$

Corollary 20. *Deciding existence of a tableau for some φ is in 2EXPTIME.*

Corollary 21. *Any satisfiable* CTL* *formula φ has a model of size at most $2^{2^{\mathcal{O}(|\varphi|)}}$ and branching-width at most $2^{|\varphi|}$.*

5 Comparison with Existing Methods

We briefly compare the tableau/automata-based reduction to parity games with existing decision procedures for CTL*, namely Emerson/Jutla's tree automata [4], Reynolds' proof system [15], Gabbay/Pnueli's proof system [8], and Reynolds' tableaux [16].

Emerson/Jutla's procedure transforms a CTL* φ formula in some normal form into a tree-automaton recognising exactly the tree-unfoldings of fixed branching-width of all models of φ. This uses a translation of linear-time formulas into Büchi automata and then into deterministic (Rabin) automata for the same reasons as outlined above. This has a drawback, as Emerson [2, Sec. 6.5] notes himself: "*... due to the delicate combinatorial constructions involved, there is usually no clear relationship between the structure of the automaton and the candidate formula.*"

Note that our approach does not use tree-automata as such—even though one may argue that the constructed parity games represent tree automata. However, the crucial difference is the separation between the use of tableau-machinery for the characterisation of satisfiability in CTL* and the use of automata-machinery only in order to obtain a decision procedure. In particular, we do not need translations of linear-time temporal formula into ω-word automata. The relationship between input formula and resulting structure (here: game) is given by the tableau rules. Furthermore, this separation makes a huge difference in practice, as we believe, because it allows the branching-width of models of φ to be flexible. Note that this is given by the number of premisses of rule $(\mathtt{X}_1)$, whereas in Emerson/Jutla's approach it is fixed a priori to a number which is linear in the size of the input formula. While this does not increase the asymptotic worst-case complexity, it does have an effect on the efficiency in practice. Not surprisingly, we do not know of any attempt to implement the tree-automata approach.

Reynolds' proof system is an approach at giving a sound and complete finite axiomatisation for CTL*. Its proof of correctness is rather intricate and the system itself is useless for practical purposes since it uses a second-order rule and it is therefore not even clear how a decision procedure, i.e. proof search could be done. In comparison, our calculus has the subformula property and comes with an implementable decision procedure. The only price to pay for this is the characterisation of satisfiability through infinite objects instead.

Gabbay/Pnueli's proof system is a unifying approach to compositional model checking and validity checking. Their work focuses on obtaining a sound and complete system. It is not clear whether this could be used in practice and we also do not know of any implementation based on that calculus. Also, the system is only complete for a special model of reactive systems in which path quantifiers are implicitly relativised.

Reynold's recent tableau system shares some similarities with our tableau system. He also uses sets of sets of formulas as well as traces (which he calls threads), etc. Even though his tableaux are finite, the difference in this respect is marginal. Finiteness is obtained through looping back, i.e. those branches might be called infinite as well. One of the real differences between the two systems lies in the way that the semantics of the CTL* operators shows up. In Reynolds' system it translates into technical requirements on nodes in the tableaux, whereas our system comes with relatively straight-forward tableau rules. The other main difference is the loop-check. Reynolds says that "*... we are only able to give some preliminary results on mechanisms for tackling repetition.* [...] *The task of making a quick and more generally usable repetition checker will be left to be advanced and presented at a later date.*" Our method comes with a non-trivial repetition checker: it is given by the annotated automata. Finally, Reynolds reports of a prototype implementation of his tableau decision procedure [16]. This implementation is, however, not publicly available, and tests are only performed on single short formulas such that no asymptotic behaviour can be inferred from those results. We strongly believe that this implementation is greatly outperformed by ours. For example, the formula $\mathtt{AG}(\mathtt{EX}p \wedge \mathtt{EX}\neg p) \wedge \mathtt{AG}(\mathtt{G}p \vee (\neg r)\mathtt{U}(r \wedge \neg p))$ apparently cannot be checked for satisfiability by Reynolds' implementation anymore whereas ours takes 0.04s for this task.

6 An Implementation

We report on practical aspects of the decision procedure described above. As said in the introduction, it is implemented in the MLSolver tool, a framework for solving satisfiability and validity problems of modal fixpoint logics. It reduces such problems to parity games and then uses PGSolver, a high-performance solver for parity games. Both tools are publicly available[1].

Optimisations. (1) It is possible to partially *determinise the proponent's strategy* without compromising on soundness or completeness: except for the modal rules $(\mathtt{X}_0)$ and $(\mathtt{X}_1)$, it is not important which rule is to be applied next. Instead of allowing the proponent to choose the rule we use a function which determines for each goal the rule that has to be applied to it next. This leaves the proponent with the

[1] `http://www.tcs.ifi.lmu.de/{mlsolver,pgsolver}`

choices of the disjuncts to be preserved in the current goal in rules $(\mathtt{Al})$, $(\mathtt{AA})$, $(\mathtt{AE})$, $(\mathtt{E}\vee)$, $(\mathtt{EU})$, and $(\mathtt{ER})$ and reduces the out-degree of the resulting parity games.

(2) MLSolver allows parity games to be generated in *compact mode*. This means that not every pair of pre-tableau goal and automaton state is present in the game. Instead, the game only contains those pairs in which rule $(\mathtt{X}_0)$ or $(\mathtt{X}_1)$ applies to the pre-tableau goal. This is possible because CTL* formulas are guarded in the sense that every infinite pre-tableau path must contain infinitely many applications of one of these rules. Compact mode does not only create much smaller games, they are also often generated faster because loop-checks do not need to be performed for every newly generated pair.

(3) MLSolver is able to perform *literal propagation* in each step of the creation of a pre-tableau. This means that whenever a literal becomes top-level in a goal, its other occurrences which are not under the scope of a temporal operator are replaced by $\mathtt{tt}$. Equally, all such occurrences of the complement literal are replaced by $\mathtt{ff}$. The resulting goal can be simplified according to the usual rules for boolean operations and then provide less disjunctive choices or allow to detect inconsistencies earlier.

(4) MLSolver prefers large formulas as principals. This scheduling reduces the branching width to linear—in contrast to the general case, cf. Cor. 21.

(5) Finally, PGSolver contains implementations of basically all known algorithms for solving parity games. While some of them are consistently bad in practice, there are some which perform quite well even though none of them is always best. These are furthermore aided by modules performing simplifications on the parity games which speed up the solving. The running times reported below are obtained using the solving algorithm which is best on the respective instance—usually the one by Stevens and Stirling [19].

Benchmarks. (1) We consider two simple families of formulas that feature deep nestings of modal operators. Let $\alpha_0 := q$, $\alpha_{n+1} := \mathtt{AFG}\alpha_n$, $\beta_0 := q$, $\beta_{n+1} := \mathtt{AFAG}\beta_n$, $\psi_n := \alpha_n \rightarrow \beta_n$, and $\varphi_n := \beta_n \rightarrow \alpha_n$ for $n \geq 0$. Both families are checked for validity, but note that ψ_n is falsifiable whereas φ_n is valid. Thus, there is a tableau for $\neg\psi_n$ but none for $\neg\varphi_n$.

(2) We consider $n+1$ programs $0, \ldots, n$. A proposition p_i states whether or not the program i is running. A scheduler is assumed which guarantees that at any time at least one program is running, and that each program runs infinitely often. Then for any execution sequence and at any time, if program 0 is running then the programs 1 to n will run in this order but possibly interrupted by others. The hole setting is given by the formula $(\mathtt{AG}(\bigvee_i p_i) \wedge \bigwedge_i \mathtt{AGF}p_i) \;\rightarrow\; \mathtt{AG}(p_0 \rightarrow \tau_1)$ where $\tau_i = \mathtt{F}(p_i \wedge \tau_{i+1})$ and $\tau_{n+1} = \mathtt{tt}$.

(3) The formula $\lambda(\varphi, \psi, \varphi', \psi') := (\varphi \wedge \mathtt{AG}(\varphi \rightarrow \mathtt{EX}(\psi \mathtt{U} \varphi))) \;\rightarrow\; \mathtt{EG}(\psi' \mathtt{U} \varphi')$ is a variation of the limit closure property, and is a tautology if φ' is a consequence of φ and ψ' of ψ. Its iterations serve as benchmarks, that are

$$\alpha_0 := q \rightarrow q \text{ and } \alpha_{n+1} := \lambda(p, \varphi, p, \varphi') \text{ where } \varphi \rightarrow \varphi' = \alpha_n\text{, and}$$
$$\beta_0 := p \rightarrow p \text{ and } \beta_{n+1} := \lambda(\varphi, q, \varphi', q) \text{ where } \varphi \rightarrow \varphi' = \beta_n.$$

Family		n	Explored Game Size	Time
Nested Modal Operators	ψ_n	14	102,200	727.55s
		15	123,774	1,315.55s
		16	148,213	1,663.43s
		17	175,697	3,173.65s
		18	?	†
	φ_n	2	400	0.25s
		3	5,581	10.18s
		4	?	†

Family		n	Explored Game Size	Time
Scheduler		1	81	0.08s
		2	852	1.21s
		3	12,320	30.82s
		4	?	†
Limit Closure	α_n	1	49	0.12s
		2	7,213	328.15s
		3	?	†
	β_n	1	49	0.12s
		2	?	†

Fig. 2. Runtime results

Experimental Results. All tests have been carried out on a 64-bit machine with four quad-core Opteron$^{\text{TM}}$ CPUs and 128 GB RAM space. The implementation does not (yet) support parallel computations, hence, each test runs on one core only and needed less than 4 GB RAM. We only present instances of non-negligible running times. On the other hand, the solving of larger instances not presented in Fig. 2 anymore has experienced time-outs after one hour, marked †.

7 Further Work

The results of the previous section show that the tableau/automata approach to deciding CTL* is reasonably viable in practice. Note that the implementation so far only features optimisations on one of three fronts: it uses the latest and optimised technology for solving the resulting games. However, there are two more fronts for optimisations which have not been exploited so far. The main advantage of this approach is—as we believe—the combination of tableau-, automata- and game-machinery and therefore the possible benefit from optimisation techniques in any of these areas. It remains to be seen for instance whether the automaton determinisation procedure can be improved or replaced by a better one. Also, the tableau community has been extremely successful in speeding up tableau-based procedures using various optimisations. It also remains to be seen how those can be incorporated in the combined method.

Furthermore, it remains to expand this work to extensions of CTL*, for example CTL* with past operators, multi-agent logics based on CTL*, etc.

References

1. Emerson, E., Jutla, C.: Tree automata, μ-calculus and determinacy. In: Proc. 32nd Symp. on Foundations of Computer Science, San Juan, pp. 368–377. IEEE, Los Alamitos (1991)
2. Emerson, E.A.: Temporal and modal logic. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, ch. 16. Formal Models and Semantics, vol. B, pp. 996–1072. Elsevier and MIT Press, New York (1990)

3. Emerson, E.A., Halpern, J.Y.: "Sometimes" and "not never" revisited: On branching versus linear time temporal logic. J. of the ACM 33(1), 151–178 (1986)
4. Emerson, E.A., Jutla, C.S.: The complexity of tree automata and logics of programs. SIAM Journal on Computing 29(1), 132–158 (2000)
5. Emerson, E.A., Sistla, A.P.: Deciding full branching time logic. Information and Control 61(3), 175–201 (1984)
6. Friedmann, O., Lange, M.: A solver for modal fixpoint logics. In: Proc. 6th Workshop on Methods for Modalities, M4M-6 (2009)
7. Friedmann, O., Lange, M.: Solving parity games in practice. In: Liu, Z., Ravn, A.P. (eds.) ATVA 2009. LNCS, vol. 5799, pp. 182–196. Springer, Heidelberg (2009)
8. Gabbay, D.M., Pnueli, A.: A sound and complete deductive system for CTL* verification. Logic Journal of the IGPL 16(6), 499–536 (2008)
9. Grädel, E., Thomas, W., Wilke, T. (eds.): Automata, Logics, and Infinite Games. LNCS, vol. 2500. Springer, Heidelberg (2002)
10. Kähler, D., Wilke, T.: Complementation, disambiguation, and determinization of Büchi automata unified. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part I. LNCS, vol. 5125, pp. 724–735. Springer, Heidelberg (2008)
11. Luo, X., Su, K., Sattar, A., Chen, Q., Lv, G.: Bounded model checking knowledge and branching time in synchronous multi-agent systems. In: Proc. 4th Int. Conf. on Auton. Agents and Multiagent Syst., AAMAS'05, pp. 1129–1130. ACM, New York (2005)
12. Miyano, S., Hayashi, T.: Alternating finite automata on omega-words. TCS 32(3), 321–330 (1984)
13. Piterman, N.: From nondeterministic Büchi and Streett automata to deterministic parity automata. In: Proc. 21st Symp. on Logic in Computer Science, LICS'06, pp. 255–264. IEEE Computer Society, Los Alamitos (2006)
14. Pnueli, A., Rosner, R.: A framework for the synthesis of reactive modules. In: Vogt, F.H. (ed.) CONCURRENCY 1988. LNCS, vol. 335, pp. 4–17. Springer, Heidelberg (1988)
15. Reynolds, M.: An axiomatization of full computation tree logic. Journal of Symbolic Logic 66(3), 1011–1057 (2001)
16. Reynolds, M.: A tableau for CTL*. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 403–418. Springer, Heidelberg (2009); Long version availabe as technical report of the University of Western Australia
17. Schewe, S.: Solving parity games in big steps. In: Arvind, V., Prasad, S. (eds.) FSTTCS 2007. LNCS, vol. 4855, pp. 449–460. Springer, Heidelberg (2007)
18. Schewe, S.: Tighter bounds for the determinisation of Büchi automata. In: de Alfaro, L. (ed.) FOSSACS 2009. LNCS, vol. 5504, pp. 167–181. Springer, Heidelberg (2009)
19. Stevens, P., Stirling, C.: Practical model-checking using games. In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, pp. 85–101. Springer, Heidelberg (1998)
20. Vardi, M.Y., Stockmeyer, L.: Improved upper and lower bounds for modal logics of programs. In: Proc. 17th Symp. on Theory of Computing, STOC'85, Baltimore, USA, pp. 240–251. ACM, New York (1985)

URBiVA: Uniform Reduction to Bit-Vector Arithmetic

Filip Marić and Predrag Janičić

Faculty of Mathematics, Studentski trg 16, 11000 Belgrade, Serbia
{filip,janicic}@matf.bg.ac.rs

Abstract. We describe a system URBiVA for specifying and solving a range of problems by uniformly reducing them to bit-vector arithmetic (BVA). A problem description is given in a C-like specification language and this high-level specification is transformed to a BVA formula by symbolic execution. The formula is passed to a BVA solver and, if it is satisfiable, its models give solutions of the problem. The system can be used for efficient modelling (specifying and solving) of a wide class of problems. Several state-of-the-art solvers for BVA are currently used (Boolector, MathSAT, Yices) and additional solvers can be easily included. Hence, the system can be used not only as a specification and solving tool, but also as a platform for evaluation and comparison between BVA solvers.

1 Introduction

In recent years, propositional satisfiability (SAT) and satisfiability modulo theory (SMT) testing have successfully been applied for solving different problems. Huge advances have been made, and state-of-the art SAT and SMT solvers can quickly solve huge problem instances coming from various industrial applications. The progress in this community is strengthen by standardization initiatives like SMT-lib[1] and by annual competitions like SAT-Comp[2] and SMT-Comp.[3] One of the SMT theories that has been extensively used in software and hardware verification lately is the theory of *bit-vector arithmetic (*BVA*)* [4]. Informally, bit-vectors represent fixed-length vectors of bits over which operations are performed as over (finite-precision) integers (either unsigned or two's complement encoded signed). Syntactically, the quantifier-free fragment of the first-order theory of bit-vector arithmetic includes arithmetic operators (`+`, `*`, `-`, `/`, `%`), relational operations (`==`, `!=`, `<`, `>`, `<=`, `>=`), bit-wise operators (`&`, `|`, `^`, `<<`, `>>`), logical operators (`&&`,`||`, `!`), operators for bit-extraction and concatenation, etc. All arithmetic operators are finite-precision and are applied only over bit-vectors of the same width. The semantics of BVA is introduced in a straightforward manner [1]. The satisfiability problem for the quantifier-free fragment of BVA is defined as usual: for a given formula F, check whether there is a variable assignment

[1] http://combination.cs.uiowa.edu/smtlib/
[2] http://www.satcompetition.org/
[3] http://www.smtcomp.org

which makes F true. Although arithmetic over arbitrary precision integers with addition and multiplication is undecidable, BVA is decidable thanks to the finite domain.[4] Additional operators can be defined (in a reasonable manner) without compromising decidability. It can be simply proved that this decidability problem is NP-complete. BVA is suitable for representing and reasoning about programs' properties because it operates on the word-level, in compliance with standard hardware and software operations over integers.

Most (if not all) reported applications of BVA are in the domain of software and hardware verification. We argue that potentials for using BVA solvers are much wider. In this paper, we describe a specification language and a tool URBiVA that can be used for solving not just verification problems, but a much wider range of problems (including, for instance, many classes of constraint satisfaction problems). The approach combines features of declarative and imperative programming. It automatically transforms problem specifications to BVA and solves generated formulae by one of underlying BVA solvers. This general reduction approach can be beneficial since hand-crafting BVA formulae that encode specific problems is typically error-prone. As we are aware of, there are still no other tools that uniformly reduce problem specifications to BVA.

2 Problem Specification

The class of problems that are considered are problems of the general form: *find (if it exists) an assignment S which satisfies some given constraints* (variations can require only checking if such an assignment exists, finding all assignments that meet the given conditions, etc). Constraints can be specified by an imperative test that checks whether S (assuming that S is given in advance) is indeed a solution. Therefore, the approach is declarative: only this test is given (instead of a solving procedure). It is often much easier to write such a test, than to write an efficient program that checks satisfiability.

The specification language is C-like and provides all standard arithmetic, bitwise, logical and relational operators, and the ternary conditional operator (`?:`). Two types of variables are supported — numerical (with identifiers starting with `n`) and Boolean (with identifiers starting with `b`). For simplicity, variables are not declared, but introduced dynamically. All operators can be applied to variables of both types (implicit type conversions are performed whenever necessary). User-defined functions are also supported. The `assert` statement specifies a condition (a Boolean expression) that must be satisfied. It triggers the underlying solver to search for an assignment to the unknowns which satisfies the assertion. The `assert_all` statement searches for all such assignments.

Let us illustrate the specification language by considering the problem of finding all Gray codes of given length, i.e., finding all permutations of integers from 0 to $dim - 1$ such that each successive pair of integers differs exactly in one bit in their binary representation. A possible specification of this problem is:

[4] Indeed, for a given formula F to be tested for satisfiability, one can test all possible assignments to variables that appear in F and check if any of them satisfies F.

```
nDim = 8;

bDomain = true;
for (ni = 0; ni < nDim; ni++)
   bDomain &&= 0 <= na[ni] && na[ni] < nDim;

bAllDiff = true;
for (ni = 0; ni < nDim-1; ni++)
   for (nj = ni+1; nj < nDim; nj++)
     bAllDiff &&= na[ni] != na[nj];

bGray = true;
for (ni = 0; ni < nDim - 1; ni++) {
   nDiff = na[ni] ^ na[ni+1];
   bGray &&= !(nDiff & (nDiff - 1)) && (nDiff != 0);
}

assert_all(bDomain && bAllDiff && bGray);
```

The vector `na` is assumed to contain the required permutation (and all such vectors should be found). The code checks if all required conditions are met. The auxiliary variable `bDomain` encodes that all elements of `na` are between 0 and *dim*, `bAllDiff` encodes that all elements of `na` are different, and `bGray` encodes that all successive pairs differ in exactly one bit.

Notice that specifications (implicitly) contain the information on the variables that are *unknown* and have to be assigned so that the given constraints are satisfied. Those are the variables that appear within expressions/statements (not on the left-hand side of the assignment operator) before they were defined (in this example, `na[0]`, ..., `na[7]`). So, the above code is a full and precise specification of the problem, up to the domains of the variables and the semantics of operators (discussed in the next section).

There are certain restrictions of the specification language: conditions in the `if`, `while` and `for` statements and indices for accessing array elements must be ground and not symbolic values. The restriction for `if` is relaxed by the presence of the conditional operator that can take symbolic arguments, while the restriction for arrays and loops cannot be removed (as it would require e.g., undefinite loop unrolling).

3 Problem Solving

Specifications given in the language outlined above are used as a starting point in problem solving. Namely, a problem specification is symbolically executed (for a given fixed bit-width) in order to build a BVA encoding of the problem. The unknowns are represented by BVA variables and results of operations are represented by BVA formulae. Finally, an assertion generates a BVA formula for which a satisfying assignment is to be found. Any satisfying valuation (if it exists) for that formula yields (ground) values for the unknowns that meet the specification, i.e., a solution to the problem.

The semantics of specification language is not equal, but rather parallel to the standard semantics of imperative programming languages. Namely, in the standard semantics, expressions (numerical and Boolean) are always evaluated to ground values and variables must be defined before they are accessed. In the proposed semantics, expressions may be evaluated to ground or symbolic values (BVA formulae) and accessing undefined variables is allowed. In the URBiVA tool, the standard semantics of unsigned BVA is assumed. The domain of Boolean variables is $\{\text{false}, \text{true}\}$ and the domain of numerical variables are finite precision unsigned integers from a domain $[0, 2^l - 1]$, for a given l (so, arithmetic modulo 2^l is assumed).[5] Constant expressions are always evaluated to ground values (for example, after the statement `nA = 3 + 2*5;`, the variable `nA` is assigned the ground value `13`, instead of a BVA formula). Note that even expressions involving symbolic values need not necessarily be evaluated to symbolic values (for example, after the statement `bX = bY && false;`, the variable `bX` can be assigned the ground value *false*, even if the variable `bY` had symbolic value).

Let us illustrate the solving process on the following specification:

```
nB = nA + 3;
nB = 2 * nB;
assert(nA + nB == 12);
```

Since, in the first line of the specification, the variable `nA` was accessed before it was defined, it is associated with a fresh bit-vector variable `A`. In the same line, the formula `A+3` is assigned to `nB`. Similarly, in the second line, the variable `nB` is assigned the symbolic value `2 * (A+3)`. Finally, the `assert` command asserts that `nB + nA == 12` is true, which gives a BVA formula `A + 2*(A+3) == 12` which is tested for satisfiability. It is true if `A` is assigned the value `2`, so a solution to the given problem is `nA == 2` (notice that the variable `nA` was the only unknown in the specification, i.e., only its value is required).

4 Implementation

The system URBiVA[6] is implemented in the programming language C++. The whole system has a flexible architecture and is relatively small. An input specification is parsed into an abstract syntax tree (AST). The interpreter traverses the AST, performs type checking and conversions and executes statements, while keeping a list of unknown variables, and a symbol table containing current variable values. Variable values are represented using a specialized data structure: ground values (BVA constants) are represented by finite length bit-arrays (implemented as byte arrays) and symbolic values (BVA formulae) are represented by term-sharing data structures (DAGs). DAG data structures for representing symbolic values can be either our custom structures, or the ones offered by an

[5] The system can be also applied for any finite-precision signed or unsigned, integer or real numbers, as long as the underlying BVA solver provides support for these types.

[6] The source code with example specifications (but without third-party solvers, due to specific licensing) is available online from: `http://argo.matf.bg.ac.rs/software/`

underlying solver's API. Using underlying solvers' native data structures helps the integration of the system (and avoids using of external files and textual formats, e.g., SMT-lib). The direct communication via API also facilitates the search for all models: once a model is found, a corresponding blocking clause is constructed and passed to the solver via API and the search for the next model can be (incrementally) started. Currently supported underlying solvers are: our custom solver based on bit-blasting [5] that uses our SAT solver ArgoSAT [6], Boolector[7] [2], Yices[8] and MathSAT[9] [3].

5 Examples and Experimental Results

In this section we give several examples that illustrate the problem modelling and problem solving within the URBiVA system and that we used for a small comparison between the underlying solvers (as yet, larger specifications related to real-world applications have not been considered). We consider one number-theory problem, two combinatorial problems, and one problem from software verification.

Fermat's triples modulo m. By the Fermat's last theorem, there are no natural numbers a, b, c such that $a^n + b^n = c^n$ and $n > 2$. However, this does not hold in arithmetic modulo m. The problem of determining the number of solutions of the given equation can be simply stated in our specification language (for a concrete n, say 3) as follows:

```
function nPower(nx, np) {
  nPower=1;
  for(ni = 0; ni < np; ni++)
    nPower *= nx;
}
assert_all(nPower(na,3) + nPower(nb,3) == nPower(nc,3));
```

This specification can solved by the URBiVA system using k-bit representation when arithmetic modulo 2^k is considered.

Gray Codes Problem. The Gray codes problem is described and its specification is given in Section 2. The parameter of the problem is dim and it can be solved for different bit-widths (sufficient for storing values from 0 to $dim - 1$).

Magic Square Problem. A magic square of order n is a $n \times n$ matrix containing the numbers from 1 to n^2, with each row, column and both diagonals equal the same sum. The problem is to find one (or all, unique up to rotations and reflections) magic square(s) of order n.[10]

[7] http://fmv.jku.at/boolector/

[8] http://yices.csl.sri.com

[9] http://mathsat.itc.it/

[10] For lack of space, we do not give a specification of this problem here, but it is available within the URBiVA distribution.

Software Verification Example — Bit Counting. Bit count (or population count) is the problem of counting all set bits of an integer. It can be implemented in a number of ways, two of which are given here for 16-bit integers, specified in our language (almost in verbatim as in the C programming language). The URBiVA tool can be used to show that these two specifications agree on all inputs, i.e., the asserted expression is unsatisfiable.

```
function nBC1(nX) {
  nBC1 = 0;
  for (nI = 0; nI < 16; nI++)
    nBC1 +=  nX & (1 << nI) ? 1 : 0;
}
function nBC2(nX) {
  nBC2 = nX;
  nBC2 = (nBC2 & 0x5555) + (nBC2>>1 & 0x5555);
  nBC2 = (nBC2 & 0x3333) + (nBC2>>2 & 0x3333);
  nBC2 = (nBC2 & 0x0077) + (nBC2>>4 & 0x0077);
  nBC2 = (nBC2 & 0x000F) + (nBC2>>8 & 0x000F);
}
assert(nBC1(nX) != nBC2(nX));
```

Experimental Results. Table 1 shows results of experimental comparison of the four underlying solvers applied on some instances of the four described problems. We solved other instances of these problems and relative performance of the solvers is rather consistent across instance sizes for one problem. We also used different bit-widths for one problem instance and longer bit-widths do not necessarily lead to longer solving times, contrary to what one might expect.

It is interesting to notice that there is no solver superior to others, and that some sorts of problems seem more suited to some solvers (even if the translation mechanism is fixed). This confirms that a system such as URBiVA should take advantage of having several different solvers supported.

Table 1. Results of experimental comparison between four underlying solvers ("bw" denotes bit-width used; all times are given in seconds; best times are given in bold face, worst times are given in italic). All experiments were performed on a PC computer, with Intel Pentium Dual-Core 2.00GHz processor and 2GB RAM.

Problem	Fermat's triples $n = 3$, bw=6	Gray codes dim=12, bw=4	Magic square $n = 4$, bw=6	Bit Count bw=32
number of solutions	10240	1168	880	0
Boolector	**3.22**	9.37	197.28	**1.20**
MathSAT	98.43	9.72	309.09	*>600.00*
Yices	*144.64*	**2.66**	**76.15**	560.67
bit-blasting	27.18	*12.23*	*461.81*	7.26

6 Conclusions, Related Tools, and Further Work

We have described a system that can be used for efficient modelling (specifying and solving) of a wide class of problems by reducing them to BVA and using the power of state-of-the art BVA solvers. The system can also serve as a testing and evaluation platform for BVA solvers. The approach is most suitable for problems for which it is easy to check whether some values satisfy the problems, but it is hard to construct such values. Such problems are, for instance, NP-problems and one-way functions. More generally, the approach can be used for calculating values $\boldsymbol{x}$, such that $f(\boldsymbol{x}) = \boldsymbol{y}$, where f is a function expressible in the described specification language. Still, the approach has two limitations. The first is a finite-precision representation used, and the second is that not all computable functions are expressible since conditional statements and array indices in the specification can involve only expressions that evaluate to ground values.

There is a number of general modelling systems using specific underlying theories and techniques (e.g., CLP(FD) systems, answer set programming (ASP) systems, ILOG OPL, DLV, etc.). All these systems use purely declarative languages (e.g, MiniZinc) and, in contrast to URBiVA, do not have features of imperative programming languages (e.g., destructive assignments). These features, however, make URBiVA directly applicable to wider class of problems (e.g., verification problems). URBiVA is also related to tools for software verification based on symbolic execution (e.g., Java Pathfinder, Pex, SAGE, SmartFuzz, FORTE). Some of these tools use SMT solvers, but they are focused on finding (single) models that lead to bugs (rather than on enumerating all solutions of a given problem). Also, they typically handle only machine data-types (and not arbitrary bit-widths).

URBiVA reduces problems to bit-vector arithmetic. However, the same methodology can be applied, in some cases, for reducing to other SMT problems. We are currently developing a wider system URSA MAJOR, that will use various SMT solvers for various theories, yielding a powerful general problem solving tool. We are planning to consider a wide range of combinatorial and verification problems from various domains and to explore the practical applicability of the approach.

References

1. Brinkmann, R., Drechsler, R.: Rtl-datapath verification using integer linear programming. In: Proceedings of the VLSI Design 2002. IEEE Computer Society, Los Alamitos (2002)
2. Brummayer, R., Biere, A.: Boolector: An efficient smt solver for bit-vectors and arrays. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 178–181. Springer, Heidelberg (2009)
3. Bruttomesso, R., Cimatti, A., Franzén, A., Griggio, A., Sebastiani, R.: The MATHSAT 4 SMT solver. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 299–303. Springer, Heidelberg (2008)
4. Bryant, R.E., Kroening, D., Ouaknine, J., Seshia, S.A., Strichman, O., Brady, B.A.: An abstraction-based decision procedure for bit-vector arithmetic. STTT 11(2) (2009)
5. Janičić, P.: Uniform Reduction to SAT (manuscript submitted) (2010)
6. Marić, F.: Formalization and Implementation of Modern SAT Solvers. Journal of Automated Reasoning 43(1) (2009)

Induction, Invariants, and Abstraction[⋆]

Deepak Kapur

University of New Mexico, Albuquerque, NM, USA
kapur@cs.unm.edu

Given that loops in imperative programs can be represented as tail-recursive programs, it is perhaps possible to gain insights into automatically generating loop invariants by analyzing inductive reasoning about tail-recursive programs. Intermediate lemmas are typically needed in a proof by induction, and methods have been investigated to speculate such lemmas from proof attempts.

The abstract interpretation framework has been successfully used to generate loop invariants using widening operators suitably designed for various abstract domains, e.g., interval domain, octagon domain, congruences, linear constraints, etc. Most of these abstract domains can be expressed in the quantifier-free theory of Presburger arithmetic. Intermediate lemma generation while reasoning about tail-recursive representations of such loops can be exploited to generate such loop invariants expressed in various fragments of Presburger arithmetic.

Interpolants have been found a useful tool for performing postcondition computation in program analysis and are related to program invariants. There is also a close relationship between interpolant computation and quantifier-elimination methods. The reduction approach to generating decision procedures for quantifier-free theories for collections such as finite sets, finite multisets(bags), finite lists, and finite arrays, can be exploited to generate interpolants for formulas in these theories, leading to automatic generation of loop invariants of programs operating on such data structures. Relationships among invariant generation, intermediate lemma speculation in inductive reasoning, interpolation, and quantifier-elimination methods are investigated.

References

1. Kapur, D.: A Quantifier Elimination based Heuristic for Automatically Generating Inductive Assertions for Programs. J. of Systems Science and Complexity 19(3), 307–330 (2006)
2. Kapur, D., Majumdar, R., Zarba, C.: Interpolation for Data Structures. In: Proc. 14th Symp. on Foundations of Software Engineering (November 2006)
3. Kapur, D., Sakhanenko, N.A.: Automatic generation of generalization lemmas for proving properties of tail-recursive definitions. In: Basin, D., Wolff, B. (eds.) TPHOLs 2003. LNCS, vol. 2758, pp. 136–154. Springer, Heidelberg (2003)
4. Kapur, D., Subramaniam, M.: Lemma discovery in automating induction. In: McRobbie, M.A., Slaney, J.K. (eds.) CADE 1996. LNCS, vol. 1104, pp. 538–552. Springer, Heidelberg (1996)
5. Kapur, D., Subramaniam, M.: Automatic generation of simple lemmas from recursive definitions using decision procedures–preliminary report. In: Saraswat, V.A. (ed.) ASIAN 2003. LNCS, vol. 2896, pp. 125–145. Springer, Heidelberg (2003)
6. Kapur, D., Zarba, C.: A Reduction Approach to Decision Procedures. TR-CS-2005-44, Dept. of Computer Science, University of New Mexico (November 2004)

⋆ Partially supported by NSF grants CCF-0729097, CNS-0831462, and CNS-0905222.

A Single-Significant-Digit Calculus for Semi-Automated Guesstimation*

Jonathan A. Abourbih, Luke Blaney, Alan Bundy, and Fiona McNeill

School of Informatics, University of Edinburgh
jabourbih@acm.org, L.Blaney@sms.ed.ac.uk,
{A.Bundy,f.j.mcneill}@ed.ac.uk

Abstract. We describe a single-significant-digit calculus for estimating approximate solutions to guesstimation problems. The calculus is formalised as a collection of proof methods, which are combined into proof plans. These proof methods have been implemented as rewrite rules and successfully evaluated in an interactive system, GORT, which forms a customised proof plan for each problem and then executes the plan to obtain a solution.

1 Introduction

The benefits of the huge amount of information available via the internet will not be fully realised until we can automatically combine it in novel ways. Unfortunately, most of the interest in information retrieval focuses around the extraction of isolated facts. Interest in question answering, which *did* try to combine such facts via inference, has waned recently (but see §6).

This paper tries to return to that earlier vision. We are interested in solving problems by using inference to combine information from a variety of sources, including linked data, natural language, etc from the internet, and knowledge already known to the user. Our research programme is as follows:

1. Construct a proof plan [3] that can be used to identify the information required to solve the current problem and then solve it.
2. Retrieve this information and use it to construct an initial problem-specific, logic-based ontology from multiple sources. Note that the sources might contain ungrammatical, uncertain and dubious information, and must be parsed, disambiguated, checked, etc and expressed logically. The final logical representation might be more sophisticated than any explicitly present in the original sources.
3. Execute the proof plan in the customised ontology to provide the required solution.

* This paper is based on the MSc project of Jonathan A. Abourbih and the undergraduate project of Luke Blaney. Dr McNeill was funded by ONR project N000140910467. We would like to thank 3 anonymous IJCAR referees for their constructive comments, and Aparna Ghagre for information on the QUARK system.

4. Sanity check the answer and the information used in its construction, e.g., ensure that the customised ontology is consistent. If a problem is diagnosed, use ontology evolution techniques to repair it, extracting new information, if necessary, then re-execute the plan.

1.1 Guesstimation

Guesstimation is the task of finding a single-significant-digit estimate to a quantitative problem based on a combination of intuition, facts, and reasoning. The types of problems that are suited to guesstimation are those for which a precise answer cannot be known or easily found [13,10]. An example guesstimation problem is:

How many golf balls would it take to circle the Earth at the equator?

This can be answered by finding the diameter of a typical golf ball and the circumference of the Earth and dividing the latter by the former. More examples can be found in Table 1 in §5. Guesstimation requires a combination of facts, planning, reasoning, and guesswork. We have implemented a semi-automated guesstimator in the system GORT (Guesstimation with Ontologies and Reasoning Techniques) using SWI-Prolog, which was chosen for its excellent linked-data toolkit.

Since common patterns of reasoning can be identified and formalised as proof plans, and the information from a variety of diverse sources must be combined in unexpected ways, guesstimation is an ideal *initial* vehicle for realising the programme outlined above. We emphasise "initial" because the work described in this paper does not yet realise all aspects of our programme. For instance, we have not yet tackled fault diagnosis or ontology repair, although these are the topic of other research projects in our research group [8,4]. Nor have we used natural language processing techniques.

1.2 The SingSigDig Calculus

According to [13], the normal form for guesstimation answers is a number in single significant digit form, $d \times 10^i$, in SI units, where d is a digit from $1, \ldots, 9$ and i is an integer. Where quantities are not originally in this normal form, numeric values must be approximated to the form $d \times 10^i$ and non-SI units must be converted to SI[1].

We now define a calculus for reasoning about numbers approximated to a single significant digit, which we will call the SingSigDig Calculus. This calculus is expressed as a set of rewrite rules between first-order terms that evaluate to numbers in this normal form, i.e., function values are expressed only approximately. These rewrite rules are based on an equality that is modulo this approximation, which we represent as $s \rightsquigarrow t$.

[1] In practice, the current GORT implementation does not stick to strictly SI units.

Let $\mathbb{R}^{\sim} = \{d \times 10^i | d \in \{1, \ldots, 9\} \wedge i \in \mathbb{Z}\}$ be the domain of normal form numbers. Let $nf^{\sim} : \mathbb{R} \mapsto \mathbb{R}^{\sim}$ be the function that converts a real number into its nearest single significant digit approximation. Observe that $\mathbb{R}^{\sim}$ is the quotient space $\frac{\mathbb{R}}{nf^{\sim}}$.

Upper-case letters represent sets; lower-case letters represent objects, and the notation $\|\ldots\| : Set(\tau) \mapsto \mathbb{R}^{\sim}$ approximates the number of elements in a set.

Formulae in the SingSigDig Calculus are first-order expressions whose domain of discourse consists of numbers in $\mathbb{R}^{\sim}$ plus everyday objects and sets of such objects. Functions and predicates defined on this domain are abstracted from those defined on $\mathbb{R}$. Suppose, without loss of generality, that all real number arguments of function f (predicate p) are initial, i.e., that $f : \mathbb{R}^m \times \tau^n \mapsto \mathbb{R}$ ($p : \mathbb{R}^m \times \tau^n \mapsto bool$), where τ is the type of any non-$\mathbb{R}$ arguments of f, if any, so $n \geq 0$. For every such function f (predicate p), we define a corresponding $f^{\sim} : \mathbb{R}^{\sim m} \times \tau^n \mapsto \mathbb{R}$ ($p^{\sim} : \mathbb{R}^{\sim m} \times \tau^n \mapsto bool$). In particular, we define the equality predicate $=^{\sim}: \mathbb{R}^{\sim 2} \mapsto bool$. In general, $f^{\sim}$ ($p^{\sim}$) can be defined in terms of f (p) as follows:

$$f^{\sim}(nf^{\sim}(r_1), \ldots, nf^{\sim}(r_m), t_1, \ldots, t_n) ::= nf^{\sim}(nf^{\sim}(r_1), \ldots, nf^{\sim}(r_m), t_1, \ldots, t_n)$$
$$(p^{\sim}(nf^{\sim}(r_1), \ldots, nf^{\sim}(r_m), t_1, \ldots, t_n) ::= nf^{\sim}(nf^{\sim}(r_1), \ldots, nf^{\sim}(r_m), t_1, \ldots, t_n))$$

These definitions ensure that $f^{\sim}$ ($p^{\sim}$) is uniquely defined on its first m numeric arguments[2]. In order to ensure uniqueness for the next n non-numeric arguments, we need to make the following assumption.

Assumption 1. *Similarity Assumption:*
For all functions $f^{\sim}$ (predicates $p^{\sim}$) and sets S, to ensure that $f^{\sim}(\ldots, S, \ldots)$ ($p^{\sim}(\ldots, S, \ldots)$) is uniquely defined then we assume that:

$$\forall s_1, s_2 \in S.\ f^{\sim}(\ldots, s_1, \ldots) \ =^{\sim} \ f^{\sim}(\ldots, s_2, \ldots)$$
$$(\forall s_1, s_2 \in S.\ p^{\sim}(\ldots, s_1, \ldots) \iff p^{\sim}(\ldots, s_2, \ldots))$$

Note that $=^{\sim}$ is actually $=$, but over $\mathbb{R}^{\sim 2}$ rather than $\mathbb{R}^2$. This makes $=^{\sim}$ an appropriate basis for a rewriting calculus, since it inherits from $=$ the properties required of rewriting, namely transitivity, monotonicity and stability. $\rightsquigarrow$ is a directed version of $=^{\sim}$. Where the context makes clear that an approximate function is being used, we will usually drop the $\sim$ superscript.

We will frequently want to specify some typical element of a set. To formalise this we will use Hilbert's ϵ operator, [5]. We will designate ϵS to be a typical representative element of the set S.

We will use polymorphic functions which apply to both objects and sets of those objects. If S is a set, the semantics of $f(S)$ is $f(\epsilon S)$. An exception to this semantic rule is the function $\|S\|$ described above, which returns the approximate number of elements in the set S.

[2] We are indebted to an anonymous referee for suggesting this version of the definition of $f^{\sim}$ ($p^{\sim}$) to ensure this property.

1.3 Hypothesis

The hypothesis that we seek to evaluate in this project is:

> *Proof planning in the* SingSigDig*Calculus can be successfully used to solve guesstimation problems.*

Our evaluation consists of implementing this technology in the system GORT and applying it to a representative sample of guesstimation problems (see §5). 'Success' here is mainly measured by the proportion of test set guesstimation problems to which GORT returns an accurate result. Secondary characteristics of success are GORT's efficiency and its adaptability to the information available to it. In §6 GORT is compared favourably to related systems.

2 Guesstimation Proof Methods

Proof plans consist of a configuration of proof methods. We now describe the proof methods that typically apply to guesstimation problems. These methods have been derived by introspection and examination of the worked problems from [13]. To date it has proven possible to express each of them as a, sometimes conditional, rewrite rule based on the $\rightsquigarrow$ relation. It is unclear whether this will continue to be the case for future methods. The methods described below are not an exhaustive list of all techniques that apply to guesstimation problems, but they do form the basis of the methods currently implemented in GORT.

We have divided the methods into *primary* and *secondary*. Primary methods form the initial part of each guesstimation proof plan and are applied manually using the web interface (see §4.3). Primary methods also often require user input of parameters. Secondary methods are used to complete the proof plan and are applied automatically by GORT (see §4.2). One exception to this classification is the *user interaction* method, which can either be called manually as a primary method or automatically, as a secondary method on backtracking when all other methods have failed.

2.1 Primary Methods

The Total Size Method. The *total size* method is applicable in cases where a guesstimation question requires the total of some physical quantity over a set. The general form of this question asks for the sum total of a quantity for all elements in a set. Thus, this type of question can be expressed as $\sum_{s\in S} f(s)$, which is rewritten as,

$$\sum_{s\in S} f(s) \rightsquigarrow f(S) \times \|S\|, \tag{1}$$

where S is a set of non-numeric objects of type τ and f is a function $f : \tau \mapsto \mathbb{R}^{\sim}$. An example might be, *What area would be required if all humans in the world were put in one place?*. Here, S is the set of all humans and $f(s)$ is the area occupied by s.

The Count Method. The *count* method is applicable in cases where a guesstimation question requires a count, $\|Small\|$, of a set of small objects, $Small$, that would exactly fit a larger object, big. An example might be, *How many golf balls would it take to circle the Earth?*, which is also worked as an example in Section 3.

We assume that the sum of some measurement g over the elements of $Small$ is equal to a measurement f of big. f and g will typically be the volume, length, duration or mass of these objects. The units of the measurements must be the same. We can formalise the *count* method as:

$$g(Small) \neq \emptyset \wedge f(big) =^{\sim} \sum_{s \in Small} g(s) \implies \|Small\| \rightsquigarrow \frac{f(big)}{g(Small)}. \qquad (2)$$

The preconditions of the *count* method are checked by user interaction; the user instantiates f and g to functions for which s/he believes the preconditions to be true.

The Law of Averages Method. The *law of averages*[3] method uses the fact that, on average, the proportion of time an object has a given property is equal to the proportion of objects in a larger population with that property at a given time. This can be formalised as:

$$S \neq \emptyset \wedge T \neq \emptyset \implies \frac{\|t \in T|\phi(\epsilon S, t)\|}{\|T\|} \rightsquigarrow \frac{\|s \in S|\phi(s, \epsilon T)\|}{\|S\|} \qquad (3)$$

For example, the proportion of time an average person spends asleep is equal to proportion of people on Earth asleep at any time, where S is the set of people, T is a finite set of equal time intervals in a day, and $\phi(s,t)$ asserts that person s is asleep during time interval t.

The Distance Method. The *distance* method is a domain-specific technique for calculating the distance between two locations on Earth. It applies in the case of a problem such as, *How much time would it take to drive from London to Manchester?*, where two locations are given and a distance is required. *distance* calculates an exact value using the following formula. For two points $\langle \phi_s, \lambda_s \rangle$ and $\langle \phi_f, \lambda_f \rangle$, where the ϕs represent latitudes and λs represent longitudes, the planar angle between the points is calculated[4] by the formula:

$$\Delta\widehat{\sigma} = 2 \arcsin\left(\sqrt{\sin^2\left(\frac{\Delta\phi}{2}\right) + \cos\phi_s \cos\phi_f \sin^2\left(\frac{\Delta\lambda}{2}\right)}\right).$$

Then the distance along the surface of the Earth is $r \cdot \Delta\widehat{\sigma}$, where the single significant digit approximation of r, the radius of the Earth, is 6×10^4 km.

[3] This name is adopted from [13]. The normal pejorative use of this phrase is not intended.

[4] http://en.wikipedia.org/wiki/Great-circle_distance

2.2 Secondary Methods

The Arbitrary Object Method. The *arbitrary object* method uses Hilbert's ϵ operator to convert the value of some function of a set into the value of that function on a typical member of that set. This can be formalised as:

$$f(S) \rightsquigarrow f(\epsilon S) \tag{4}$$

For example, S might be the set of humans and $f(s)$ the height of the human s.

The Average Value Method. The *average value* method guesstimates a numeric value for some $f(\epsilon S)$ by computing the arithmetic mean of all $f(s), s \in S$:

$$S \neq \emptyset \implies f(\epsilon S) \rightsquigarrow \frac{\sum_{s \in S} f(s)}{\|S\|}. \tag{5}$$

An application of this method could be to find the average runtime of a typical film, based on knowledge about runtimes of particular films. In this case, S would be the set of all films and $f(s)$ the runtime of the film s.

The Aggregation over Parts Method. A guesstimation problem may require a quantity for a large object that is composed of many non-overlapping smaller objects. This is formalised as the *aggregation over parts* method:

$$f(o) \rightsquigarrow \sum_{p \in Parts(o)} f(p), \tag{6}$$

where $Parts(o)$ is a function that returns the set of all non-overlapping parts of o. One such example could be a need for the population of a continent. In that case, the continent could be subdivided into non-overlapping regions, such as countries. o would be the continent and each p a country in that continent. The population of o would be calculated as the sum of the populations of the ps.

The Generalisation Method. The *generalisation* method finds more general information when it isn't available for the typical member of a specific set. By looking at the properties of the typical member of a superset, an approximate value can be found.

$$S \subset T \implies f(\epsilon S) \rightsquigarrow f(\epsilon T) \tag{7}$$

For instance, suppose we cannot discover the thickness of a typical lottery ticket. Knowing that lottery tickets are made from cardboard, we can seek instead the thickness of a typical piece of cardboard. S would be the set of lottery tickets and T the set of cardboard objects.

The Geometry Methods. Guesstimation problems often need to reason about the physical properties of an object, such as its surface area or volume. Where a precise value for the needed measurement is unavailable, it may be possible to calculate the required measurement from other knowledge, for example a sphere's circumference, $Circ(s)$, given its radius, $Radius(s)$. The *geometry* methods require knowledge of the shape of an object and a measurement. An example of a *geometry* method is one that expresses the circumference of a circular object, $Circ(s)$, in terms of its radius, $Radius(s)$:

$$Circ(s) \rightsquigarrow 2\pi Radius(s)$$

Similarly, methods have been implemented for computing the volume and the surface area of both spherical and rectangular prism objects.

The User Interaction Method. The solution to a guesstimation question may rely on information that GORT cannot find on the internet. The *educated guess* method then asks the user for the required value. The user can also elect to use this method as a primary one.

2.3 Towards an Axiomatic Equational SingSigDig Theory

The question naturally arises as to whether we can develop an axiomatic equational SingSigDig theory in which the rewrite rules of this section are theorems. We have begun some experiments towards this end to explore some of the options. The domain-specific rewrite rules generally follow just from the rules of algebra, trigonometry, geometry, etc. We, therefore, restricted our attention to the general-purpose rewrite rules. Firstly, we made unoriented versions of each of them, considered as equations over the $=^{\sim}$ relation. Then we considered which of them could be derived from the others.

If equations based on rewrite rules (4) and (7) are adopted as axioms, then equations based first on (1), and then on (5), (2) and (3), can all be proved as theorems. Equation (6) requires a definition of $Parts(o)$ in terms of o, or could itself be adopted as an axiom.

Note that, as oriented, the rewrite rules for the *arbitrary object* (4), *average value* (5) and *total size* (1) methods have the potential to loop. Such loops are currently prevented by the division of the proof plan into separate primary and secondary phases. The potential loops we have identified all contain both a primary and a secondary method. Since no primary method is allowed to follow a secondary method, such loops do not arise in practice. If, in the future, this restriction is relaxed, or if purely primary or secondary loops are discovered, then a loop checking mechanism will be required.

GORT's proof methods are approximate in two senses. Not only do they return an answer only accurate to within a single significant digit, but they are also fallible. If, for some function $f^{\sim}$ and set S the Similarity Assumption 1 is violated then different ways of evaluating $f^{\sim}(\ldots, S, \ldots)$ can return different values. Currently, it is the responsibility of the user to check that this assumption is met. We would like to automate this check, but it seems inherently resistant to automation.

An anonymous referee wondered whether it might be possible to order the methods by some fallibility measure, e.g., the probability of their truth, and use this measure during search control. Given their interderivability, it seems unlikely that it will be possible to order the proof methods in this way. Rather, fallibility is not due to the particular method used, but rather to the violation of the Similarity Assumption for some function and set.

3 Worked Examples

We now illustrate how GORT can combine the proof methods from §2 into a proof plan to solve a guesstimation problem. Our worked example is taken from [13].

Problem: *How many golf balls would it take to circle the Earth at the equator?*

Solution: We begin by identifying the type of plan that is appropriate for this question. The result of the question must be a count of a set of golf balls; therefore, the appropriate proof plan is the *count* method described in §2.1. The objects being considered are the set of golf balls required, $Golf\_Balls$, and the object, *earth*[5], and the properties under consideration are the diameter and circumference of these objects, respectively.

We start by choosing the *count* method (2). The user instantiates *big*, *Small*, *f* and *g* to *earth*, *Golf_Balls*, *Circ* and *Dia*, respectively. The user also confirms that the preconditions $Dia(Golf\_Balls) \neq 0$ and $Circ(earth) = \sum_{s \in Golf\_Balls} Dia(s)$ are satisfied. This creates the following rewrite rule:

$$\|Golf\_Balls\| \rightsquigarrow \frac{Circ(earth)}{Dia(Golf\_Balls)} \tag{8}$$

whose RHS must be evaluated to provide the required value for the LHS.

Since *Golf_Balls* is a set, the *arbitrary object* method applies and rewrites the RHS of (8), giving:

$$\|Golf\_Balls\| \rightsquigarrow \frac{Circ(earth)}{Dia(\epsilon Golf\_Balls)}. \tag{9}$$

Continuing, we now need values for $Circ(earth)$ and $Dia(\epsilon Golf\_Balls)$. We take an *educated guess* for the diameter of a golf ball:

$$Dia(\epsilon Golf\_Balls) \rightsquigarrow 4.10^0 \text{ cm}$$

Next, we need the circumference of the *earth*. [13] uses background knowledge about flights and time zones to guesstimate the circumference. However, with an information retrieval system at our disposal, we can easily look up the radius of the *earth*, calculate the circumference and rewrite the units to match those of our archetypal golf ball:

$$Radius(earth) \rightsquigarrow 6.10^3 \text{ km}$$
$$Circ(earth) \rightsquigarrow 2\pi \times Radius(earth) \rightsquigarrow 4.10^5 \text{ km} \rightsquigarrow 4.10^9 \text{ cm}$$

[5] Recall the upper/lower case convention for sets/objects given in §1.2.

Finally, we continue the plan from (9) to obtain the result:

$$\frac{Circ(earth)}{Dia(\epsilon Golf\_Balls)} \rightsquigarrow \frac{4.10^9 \text{ cm}}{4.10^0 \text{ cm}} \rightsquigarrow 1.10^9 \qquad \square$$

4 Implementation

GORT is implemented as a collection of modules, each of which is described below.

4.1 Basic Ontology

GORT needs background knowledge to determine the appropriate proof plans that apply at a given point in a guesstimation solution. The basic ontology consists of an upper ontology and a small set of ground facts. The knowledge base encompasses the following bodies of knowledge.

– shapes and geometric properties of objects, such as roundness, whether an object is tangible, etc;
– measurement units and dimensions, to support scale unit conversion and common conversion factors;
– a hierarchy of concepts and entailments that allows reasoning about subsumption relationships between sets (e.g., all *Actor*s are also *Person*s);
– a large range of instances of sets, to allow the user to express questions on a broad number of topics.

4.2 Inference System

The proof plans, composed of the methods described in §2, form the basis of GORT's ability to reason over the facts in the knowledge base. The system implements both general and domain-specific methods, and refers to knowledge in the knowledge base to select an appropriate method. Primary methods are selected by the user and secondary method are selected automatically by exhaustive, depth-first rewriting. The proof planner handles failure by backtracking to attempt other plans when possible. If none of the proof plans achieve a result, then the planner will need to trigger a user interaction to get a needed fact.

4.3 The Web Interface

A prototype web interface has been developed for GORT. It uses AJAX to load the results asynchronously. A drag-and-drop interface (using the BBC's GLOW JAVASCRIPT LIBRARY[6]) allows users to select primary methods and provide the parameters required by these methods. When a proof method receives these parameters, it updates itself and any other methods which rely on its output.

[6] `http://www.bbc.co.uk/glow/`

4.4 Customised Ontology

The Customised Ontology serves two purposes. First, it provides a mechanism for GORT to record intermediate calculation results and facts that it has retrieved from either the knowledge base or some outside source. Second, because the custom ontology records each intermediate calculation and retrieved fact, it also makes explicit the knowledge that GORT uses to solve a guesstimation question.

4.5 Information Retrieval

GORT needs a way to gather data in response to a user query, and combine it with the proof plans and background knowledge already in the knowledge base to arrive at a solution. This module is responsible for gathering appropriate information in response to a user query. It will eventually be adaptable to a range of information sources, such as Semantic Web sources and natural-language knowledge sources.

GORT, currently supports two methods of information retrieval. Firstly it has various pre-stored ontologies in the form of RDF triples, with explicit links into them using `rdfs:subClassOf`, `rdfs:subPropertyOf`, and `owl:sameAs` relations. Secondly, it uses SPARQL to dynamically link to ontologies that have an appropriate endpoint.

SPARQL(SPARQL Protocol and RDF Query Language) is an RDF query language. It allows the querying of RDF datastores, which means that only relevant data is returned. This avoids having to download lots of unnecessary data, which could be a problem for large datastores. There are a range of datastores with SPARQL endpoints[7], including DbPedia[8], the BBC[9] and Edubase[10]. Queries are sent over HTTP to the SPARQL endpoint, which then returns the relevant results. GORT uses parts of the CLIOPATRIA SEMANTIC SEARCH LIBRARY[11] for SWI-Prolog to assist in querying the endpoints.

Another benefit of using SPARQL is that GORT doesn't need to remember URIs for every object. Queries can be sent using the English labels given by the user and numerical results are returned.

5 Evaluation

The results of a run of GORT on 8 example test problems are summarised in Table 1. The tests were run on a Linux-based, 3Ghz HP dc7900 running SWI-Prolog version 5.8.1. Problem 2 was run twice — once without the data for Loch Ness (2*), and again after having loaded it.

The 'Target' results are taken from third party sources, where available, such as [13]. Note that GORT produced results accurate to within a single significant

[7] `http://esw.w3.org/topic/SparqlEndpoints`
[8] `http://dbpedia.org/sparql`
[9] `http://api.talis.com/stores/bbc-backstage/services/sparql`
[10] `http://services.data.gov.uk/education/sparql`
[11] `http://e-culture.multimedian.nl/software/ClioPatria.shtml`

Table 1. Results for all Test Problems

Problem	Answer	Target	User	Time (s)
1. *How many cells are there in the human body?*	2.10^{14}	1.10^{14}	$\surd$	10.4
2*. *How many golf balls would it take to fill Loch Ness?*	*fail*	*fail*	n/a	< 0.1
2. *How many golf balls would it take to fill Loch Ness?*	2.10^{14}	1.10^{14}	$\surd$	< 0.1
3. *If all Europeans were placed head-to-toe, how far would they reach?*	1.10^{9} m	1.10^{9} m	$\times$	11.0
4. *How many people would be needed to form a chain from central London to central Edinburgh?*	3.10^{5}	3.10^{5}	$\times$	10.4
5. *How many Loch Nesses would fit into the Red Sea?*	3.10^{4}	3.10^{4}	$\times$	0.17
6. *How many Hangzhou Bay Bridges would it take to cross the Doppler crater on the moon?*	3.10^{0}	3.10^{0}	$\times$	0.17
7. *If everyone in the crowd at Croke Park drove a Volkswagen New Beetle and parked them in straight line, how long would the line be?*	3.10^{8} mm	3.10^{8} mm	$\times$	0.5
8. *How many Channel Tunnels would it take to stretch from Edinburgh to New York?*	2.10^{2}	1.10^{2}	$\times$	0.4

The Answer *column gives* GORT*'s answer and* Target *gives the target result. A* $\surd$ *in the* User *column shows that user input was required via the* educated guess *method; a* $\times$ *shows it wasn't. The* Time *column shows the average CPU time in seconds, averaged over 10 runs.*

digit for 5 of the 8 problems and within an order-of-magnitude for the other three. The three discrepancies arise from differences in the proof plans used by GORT and human guesstimators, and from the inherent fallibility of GORT's guesstimation methods (see discussion in §2.3).

To estimate the success rate of GORT it was run on all 11 of the 'general questions' from [13][Chap. 3]. These 11 problems were chosen because they were a wide and representative sample from an independent source. GORT was able to solve 6 of these 11 problems. For all 6 successful problems it returned a result identical to that given by Weinstein and Adams. The remaining 5 failed for various reasons. 2 failed because GORT did not have a guesstimation method able to solve rate of change problems (see §7.2 for further discussion). The other 3 failed due to the lack of the required geometric or chemical knowledge.

The adaptability of the system was assessed by its ability to modify its execution plan when new RDF triples appeared in the knowledge base, simulating the appearance of new knowledge in the web of linked data. In problem 2, when the volume data for Loch Ness was missing, GORT queried the user to provide it, but the user declined, so the attempt failed. The missing information was then provided and the problem successfully rerun.

6 Related Work

There are no systems that are directly comparable to GORT. However, there are several systems that share common characteristics. In this section, we compare GORT with five other Semantic Web and knowledge-based systems: PowerAqua, a Semantic Web-based question answering system [7], QUARK, a domain-independent, logic-based, natural-language, question-answering system [12], CS Aktive Space, a system for tracking UK computer science research [11], Cyc, a general-purpose 'common-sense' reasoning system [6], and Wolfram|Alpha, a system that calculates answers to numerical questions on a wide range of topics[12]. The comparisons will be conducted along the four dimensions below, which were proposed in [9] for evaluating next-generation Semantic Web applications. They have been used to compare other Semantic Web systems, so act as an objective set of evaluation criteria.

1. the system's ability to re-use Semantic Web data;
2. whether the system is *single-ontology* or *multi-ontology*;
3. the system's ability to adapt to new Semantic Web resources at the request of the user; and
4. the system's ability to scale.

Data Reuse. To begin, we consider the systems' abilities to reuse data from other Semantic Web systems. The earliest system under consideration is Cyc, which is a large-scale, curated, knowledge base that is not able to directly incorporate knowledge from Semantic Web sources, although there are techniques for mapping Cyc concepts to external concepts. CS Aktive Space (CSA) was designed in the early days of the Semantic Web, and thus little data was available. It is not designed to dynamically adapt to other Semantic Web resources until their contents have been translated into its AKT reference ontology. Wolfram|Alpha is the newest of the systems under consideration. Like Cyc, it uses a closed, hand-curated data set to perform its inferences and does not incorporate a facility for accessing Semantic Web resources. PowerAqua, QUARK and GORT, on the other hand, are each designed to operate with data from other Semantic Web resources. PowerAqua and QUARK can answer questions based on data gathered from a large number of ontologies, and do so dynamically at runtime. GORT is also capable of incorporating data from several Semantic Web data sources.

Single- or Multi-Ontology. Secondly, we consider whether each system is *single-ontology* or *multi-ontology*. This distinction is important: only a *multi-ontology* system assumes that it operates in a larger data ecosystem such as the Semantic Web. As systems with hand-curated, proprietary knowledge bases, both Cyc and Wolfram|Alpha are clearly single-ontology systems. These systems do not appear to be capable of working with more than their own ontology. Although CS Aktive Space incorporates data from multiple sources, all of its

[12] http://www.wolframalpha.com/

data is re-mapped into the AKT reference ontology. As previously mentioned, PowerAqua and QUARK can work with multiple ontologies at the same time. PowerAqua discovers and integrates these at runtime. QUARK is linked to various information interchange sources. This flexibility makes it particularly easy to incorporate new Semantic Web data sources into PowerAqua and QUARK — they have been designed with multiple ontologies in mind. GORT is also capable of working with multiple ontologies. The use of SPARQL makes it fairly easy to add new ontologies by adding their endpoints.

Openness. Thirdly, we consider each system's openness to new semantic resources. Three of the systems are not especially amenable to the incorporation of new semantic resources: none of Cyc, Wolfram|Alpha, or CS Aktive Space can incorporate new RDF content in response to a user query. This is a consequence of each of those systems' single-ontology approach. There are techniques for mapping new ontologies into Cyc, but these ontologies cannot automatically be retrieved and mapped at the user's request. PowerAqua is capable of incorporating new ontologies into its query answering at its user's request, without additional configuration. QUARK can be readily reprogrammed to link it to additional ontologies. GORT is also open to new ontologies via its SPARQL interface. By adding an ontology's SPARQL endpoint, its data becomes available.

Scalability. Finally, we consider each system's ability to scale to large data sets. Although Cyc's knowledge base is large and supports complex inferences, it is small in comparison to the projected size of the Semantic Web. Although Cyc is adaptable to Semantic Web sources, no testing has been done to evaluate its performance on large, non-curated data sets. Similarly, QUARK is linked to some very large ontologies, but we could find no experimental data on its scalability. There is no public estimate of the amount of data stored in Wolfram|Alpha, but the range of problems for which it provides an answer suggests a very large knowledge base, but this is unlikely to approach the magnitude of the Semantic Web. PowerAqua is designed with the large-scale Semantic Web in mind, and its query performance has been tested against large data sets. The PowerAqua system has access to a larger data set, more sophisticated inference algorithms, and is capable of answering a larger variety of question types than GORT, although it does not solve guesstimation problems. The 0–11 second response time of GORT with a database of 3 million triples, suggests that it is scalable up to several million more, still staying within tolerable response times.

7 Conclusion

In this paper we have argued that proof planning in an SingSigDig calculus can be successfully used to solve guesstimation problems. We have implemented this technology in the GORT system and applied it to a representative sample of guesstimation problems §5. GORT has been compared favourably with related systems in §6.

7.1 Discussion

We now consider the success of GORT by the criteria defined in §1.3, namely: the proportion of accurate results returned from its test set; its adaptability; and its efficiency.

In §5, GORT was able to produce an answer to all 8 of the test set of guesstimation problems. On 5 problems, GORT guesstimated the target single significant digit answer. On 3 other problems it came within an order-of-magnitude of the target answer. These discrepancies arose from different choices in the proof plans used to guesstimate the answers, and seem inevitable given the inherent approximate nature of guesstimation. On another run, designed to evaluate its success rate on an independently sourced test set, GORT successfully guesstimated 6/11 problems to within a single significant digit.

GORT was shown to be adaptable via the experiments on problem 2, as discussed in §5. Further such experimentation is desirable.

The timing data shown in Table 1 indicate that each query completed in less than 11 seconds, over a knowledge base of approximately 3 million RDF triples. Profiling shows that the system spends most of its time in tactics that perform list aggregation, such as the *average value* and *aggregation over parts* plans. We have also investigated the computational complexity of GORT's various proof methods. *average value* and *aggregation over parts* both use breadth-first search, and so have complexities $O(b^d)$, where b is the branching rate and d is the depth of the search space. The other secondary methods all have complexity $O(1)$; the complexity of the primary methods depends on the secondary methods they call.

7.2 Further Work

To extend GORT to handle the full range of guesstimation problems found in sources such as [13,10] requires several additional methods. For instance, many problems require examining rates, such as "How long would it take to fill the dome of St. Paul's Cathedral with water from a typical garden hose?" To solve this problem would require a proof method for reasoning about rates of flow.

To make GORT easier to use, the web-service interface needs considerable improvement. In the long term, we plan to build a natural language interface, so that guesstimation problems can be posed as English questions. We also plan to automate the choice of top-level proof method, by analysing the form of the question. This would include, for instance, automating the instantiation and checking of the preconditions of methods such as *count*[13]. We will also continue to explore the development of an axiomatic equational theory for the SingSigDig calculus, as discussed in §2.3.

References

1. Abourbih, J.A.: Method and system for semi-automatic guesstimation. Master's thesis, University of Edinburgh, Edinburgh, Scotland (August 2009)
2. Blaney, L.: Semi-automatic guesstimation. University of Edinburgh, Undergraduate Project Dissertation (2010)

[13] This will be the topic of a new MSc project by Aparna Ghagre.

3. Bundy, A.: A science of reasoning. In: Lassez, J.-L., Plotkin, G. (eds.) Computational Logic: Essays in Honor of Alan Robinson, pp. 178–198. MIT Press, Cambridge (1991)
4. Bundy, A., Chan, M.: Towards ontology evolution in physics. In: Hodges, W., de Queiroz, R. (eds.) WoLLIC 2008. LNCS (LNAI), vol. 5110, pp. 98–110. Springer, Heidelberg (2008)
5. Hilbert, D., Bernays, P.: Die Grundlagen der Mathematik — Zweiter Band. In: Number L in Die Grundlehren der Mathematischen Wissenschaften in Einzeldarstellungen. Springer, Heidelberg (1939)
6. Lenat, D.B.: CYC: a large-scale investment in knowledge infrastructure. ACM Commun. 38(11), 33–38 (1995)
7. Lopez, V., Guidi, D., Motta, E., Peroni, S., d'Aquin, M., Gridinoc, L.: Evaluation of semantic web applications. OpenKnowledge Deliverable D8.5, Knowledge Media Institute, The Open University, Milton Keynes, England (October 2008) (accessed August 10, 2009)
8. McNeill, F., Bundy, A.: Dynamic, automatic, first-order ontology repair by diagnosis of failed plan execution. International Journal On Semantic Web and Information Systems 3(3), 1–35 (2007); Special issue on ontology matching
9. Motta, E., Sabou, M.: Next Generation Semantic Web Applications. In: Mizoguchi, R., Shi, Z.-Z., Giunchiglia, F. (eds.) ASWC 2006. LNCS, vol. 4185, pp. 24–29. Springer, Heidelberg (2006)
10. Santos, A.: How Many Licks: Or, How to Estimate Damn Near Anything. Perseus Books, Cambridge (2009)
11. Shadbolt, N., Gibbins, N., Glaser, H., Harris, S., Schraefel, M.C.: CS AKTive space, or how we learned to stop worrying and love the semantic web. IEEE Intelligent Systems 19(3), 41–47 (2004)
12. Waldinger, R., Hobbs, J., Appelt, D.E., Fry, J., Israel, D.J., Jarvis, P., Martin, D., Riehemann, S., Stickel, M.E., Tyson, M., Dungan, J.L.: Deductive question answering from multiple resources. In: New Directions in Question Answering, pp. 253–262. AAAI Press, Menlo Park (2003)
13. Weinstein, L., Adam, J.A.: Guesstimation: solving the world's problems on the back of a cocktail napkin. Princeton University Press, Princeton (2008)

Perfect Discrimination Graphs: Indexing Terms with Integer Exponents[⋆]

Hicham Bensaid[1,2], Ricardo Caferra[2], and Nicolas Peltier[2]

[1] INPT/LIG, Avenue Allal Al Fassi, Madinat Al Irfane,
Rabat, Morocco
bensaid@inpt.ac.ma
[2] Grenoble University (LIG/CNRS)
Bâtiment IMAG C - 220, rue de la Chimie 38400 Saint Martin d'Hères, France
{Ricardo.Caferra,Nicolas.Peltier}@imag.fr

Abstract. Perfect discrimination trees [12] are used by many efficient resolution and superposition-based theorem provers (e.g. E-prover [17], Waldmeister [10], Logic Reasoner[1], ...) in order to efficiently implement rewriting and unit subsumption. We extend this indexing technique to handle a class of *terms with integer exponents* (or *I-terms*), a schematisation language allowing to capture sequences of iterated patterns [8]. We provide an algorithm to construct the so called *perfect discrimination graphs* from I-terms and to retrieve indexed I-terms from their instances. Our research is essentially motivated (but not restricted to) by some approaches to inductive proofs, for which termination of the proof procedure is capital.

1 Introduction

Inductive definitions are intensively used in mathematics and computer science. For instance natural numbers, lists, trees, ... are all formalised by means of inductive definitions. Automatically proving properties of such objects is more difficult than reasoning in first-order logic because the validity of the logical formulae has to be considered in special interpretations (Herbrand minimal models for example) and not in the set of all possible interpretations. We distinguish two classes of approaches in inductive theorem proving: The first one is based on explicit induction schemes (see [5] for a survey): for example, to prove that a property P on natural numbers is inductively valid, we prove that $P(0) \wedge \forall n(P(n) \Rightarrow P(n+1))$ is valid. The second class of approaches contains all the proof procedures in which the inductive schema is not made explicit, ranging from the *implicit induction* approach [3,4] to the *inductionless induction* method [7]. In particular, in this latter setting, inductive proofs are reduced to first-order consistency verification, which makes possible to use saturation in a first-order theorem prover to check inductive validity. The problem then is that in many cases, saturation

⋆ This work has been partly funded by the project ASAP of the French *Agence Nationale de la Recherche* (ANR-09-BLAN-0407-01).

[1] http://sites.google.com/site/lcastelli2/logicreasoner

(and consequently satisfiability) cannot be detected due to the divergence of the inference process.

To address this divergence problem, one often has to transform the initial formulae into a different equivalent "convergent" statement[2]. This can be done for instance by adding appropriate inductive lemmata or by using reformulation techniques. Such lemmata can be generated automatically by running the prover for a while and by trying to detect regularities in the search space. This approach is used for instance by the divergence critic of [19], which uses a "difference matching" procedure to generate lemmata. Another approach based on generalisation techniques is described in [1]. The inferred formulae (clauses) share in general a common structure and the differences can be described by a rule or a pattern. For example from the input $\{even(0), \forall x(even(x) \Rightarrow even(s(s(x)))\}$ an infinite set of clauses can be produced, i.e. $\{even(0), even(s(s(0))), even(s(s(s(s(0))))), \ldots\}$. Several formalisms have been introduced to grasp such repeating patterns. They allow one to finitely specify infinite sequences of structurally similar terms, obtained by repeated applications of a given context (e.g. $s(s(\ldots)))$ on an initial term (e.g. 0). The infinite set of clauses in the last example can be represented by $even(s^{2N}(0))$ where N is an arithmetic variable. The concept of recurrent terms has been introduced in [6] and then other formalisms have been defined, which gave rise to a hierarchy of term schematisation languages in term of expressiveness and complexity [8,16,9]. The existence of these formalisms suggests that it could be worth implementing the handling of term schematisation in existing first-order provers, in order to help to solve divergence problems, particularly (but not exclusively) in inductive proofs.

There are however two serious constraints: the first one is that a (finitary) unification algorithm is needed for performing inferences on the considered formalism (in order to avoid loosing decidability of inference rules) and the second one is that suitable indexing techniques must be developed to perform efficiently redundancies elimination. The unification problem has been proven to be decidable (and finitary) for the previously mentioned term schematisation languages ([8,16,9]). The present work tackles the second issue, namely the extension of state-of-the-art indexing techniques to such languages (see [14] for a survey on existing term indexing techniques), more precisely of the perfect discrimination trees [12] that are used to implement efficiently subsumption and simplification rules. This point is crucial because such simplification rules are usually essential for termination and using general purpose algorithms [13] for finding subsuming (generalised) terms would be exceedingly time consuming. The matching problem (i.e. finding for a term s, a term t such that there exists a substitution σ verifying $s = t\sigma$) is much more complex than for usual first-order terms because one has to handle unfolding: for instance $s^N(0)$ subsumes both $s(s(s(0)))$ and $s(s^{2N+1}(0))$. The expressive power of term schematisations makes the extension of usual indexing techniques such as discrimination trees to these languages a very difficult – but potentially rewarding – task. We focus on a particular term schematisation language, called *I-terms*, that presents a good tradeoff between

[2] This is obviously not always possible since first-order logic is only semi-decidable.

expressiveness and simplicity. We devise an extension of the perfect discrimination tree indexing algorithm for a sub-class of the I-terms, that we call *eligible*.

The paper is structured as follows. Section 2 introduces the main definitions, in particular the syntax and semantics of terms with integer exponents (I-terms). Section 3 defines the notion of *index graph*, an extension of discrimination trees that is able to deal with I-terms. Section 4 shows how to construct an index graph for a given I-term (or sets of I-terms). Section 5 contains the most important contribution of the present paper, namely the algorithm for retrieving subsuming (indexed) terms for a given term. Section 6 briefly concludes the paper.

2 Definitions and Notations

We consider three disjoint (nonempty) sets of symbols: a set of *ordinary variables* V_X, a set of *function symbols* $\mathcal{F}$ and a set of *arithmetic variables* V_N. We consider two special symbols $\diamond$, $\#$ not occurring in $V_X \cup V_N \cup \mathcal{F}$: $\diamond$ is called the *hole* and is used to define inductive contexts (see Definition 1) and $\#$ is used in Section 3 to encode cycles in index graphs. By convention, arithmetic variables will be denoted by capital letters $N, M, \ldots$ and ordinary variables by $x, y, \ldots$.

We first define the set of terms with integer exponents (I-terms) and the set of terms with one hole (originally introduced in [8]). For the sake of readability, examples are given before formal definitions.

Example 1 (I-terms). $f(a, \diamond, b)^N.g(c)$ is an I-term denoting the infinite set of (standard) terms $\{g(c), f(a, g(c), b), f(a, f(a, g(c), b), b), ...\}$.

E_N denotes the set of arithmetic expressions built as usual on the signature 0, s and V_N (in particular $\mathbb{N} \subseteq E_N$).

Definition 1. *The set of* terms with integer exponents T_I *(or* I-terms*) and the set of* terms with one hole $T_\diamond$ *are the smallest sets verifying:*

- $\diamond \in T_\diamond$ *and* $V_X \subset T_I$.
- *If* $t_1, \ldots, t_k \in T_I$, $f \in \mathcal{F}$ *and* $arity(f) = k$ *then* $f(t_1, \ldots, t_k) \in T_I$.
- *If* $arity(f) = k > 0$, $t_j \in T_I$ *for* $j \in [1, k], j \neq i$ *and* $t_i \in T_\diamond$ *then* $f(t_1, \ldots, t_n) \in T_\diamond$. t_i *is called the* hole holder.
- *If* $t \in T_\diamond$, $t \neq \diamond$, $s \in T_I$ *and* $N \in E_N$ *then* $t^N.s \in T_I$. *An* I*-term of this form is called an* N-term. t *is the* inductive context, N *is the* exponent *and* s *is the* base term. *A* $\mathcal{F}$-term *is an* I*-term that is not an* N*-term.*

If t is a term with one hole, we denote by $t[s]$ the term obtained from t by replacing the (unique) occurrence of the hole (not in an N-term) by s. We denote by $t[s]_p$ the term obtained from t by replacing the subterm at position p by s. The semantics of I-terms are defined by the following rewrite rules:

$$t^0.s \rightarrow s \qquad t^{N+1}.s \rightarrow t[t^N.s]$$

This rewrite system is convergent and together with the rules of Presburger arithmetic, it reduces any I-term not containing arithmetic variables to a standard term. We denote by $t \downarrow$ the normal form of t.

To simplify technicalities, we assume that the exponent of every N-term is a variable (this is easy to ensure by unfolding, for instance $f(\diamond)^{2N+1}.a$ is equivalent to $f(f(f(\diamond))^N.a))$.

It is clear, from the semantics of the I-terms, that the head symbol of the normal form of an N-term $t^n.s$ depends on the value of n. The function $ph()$ returns the set of possible head symbols of a term:

Definition 2. *The function ph:* $(T_I \cup T_\diamond) \to \mathcal{P}(\mathcal{F})$ *is defined by:*

- $ph(x) = \mathcal{F}$ *if* $x \in V_X$.
- $ph(\diamond) = \emptyset$.
- $ph(f(t_1, \dots, t_k)) = \{f\}$.
- *If* $t \in T_\diamond \backslash \{\diamond\}$, $s \in T_I$, $N \in V_N$ *then* $ph(t^N.s) = ph(t) \cup ph(s)$.

A term $t \in T_I \cup T_\diamond$ is *eligible* iff for every N-term $u^N.v$ occurring in t we have $ph(u) \cap ph(v) = \emptyset$. The set of eligible terms in T_I (resp. $T_\diamond$) is denoted by Ξ (resp. $\Xi_\diamond$). Notice that Ξ is closed under substitution (if the variables are replaced by eligible terms), thus restricting a resolution-based prover to eligible I-terms is a straightforward task: it suffices to check that all I-terms occurring in the *initial* set are eligible. Ξ is *not* closed under substitutivity, thus using superposition would require additional restriction to ensure that no non eligible term can be generated[3]. However, the superposition calculus is not complete in presence of I-terms [2]. N-terms of the form $u^n.x$ where $x \in V_X$ are not eligible.

Example 2 (Eligible and non eligible I-terms). $f(a, \diamond, b)^N.g(c)$ and $f(\diamond)^N.g(\diamond)^M.b$ are eligible I-terms, $f(\diamond)^N.g(\diamond)^M.f(a)$ is not because $ph(g(\diamond)^M.f(a)) = \{g\} \cup ph\{f(a)\} = \{g\} \cup \{f\}$ thus $f \in ph(g(\diamond)^M.f(a))$.

The restriction on eligible terms makes the index graph deterministic (i.e. reading the next input symbol is sufficient to decide whether we are in the inductive context or in the base term) and permits to define an efficient retrieving algorithm (without it we would have to consider several paths in parallel).

From now on, we consider *exclusively* eligible I-terms.

3 Index Graphs

An *index graph* is an automaton (see, e.g. [11]), driven by the symbols occurring (from left to right) in the indexed term. We distinguish three kinds of edges: edges labeled with a symbol from the indexed terms or with the symbol #, edges labeled with arithmetic variable symbols (i.e. from the set V_N) and edges labeled with equations of the form $N = 0$ where $N \in V_N$.

Definition 3. *Let* $\mathcal{V} = V_X \cup \mathcal{F} \cup \{\diamond, \#\}$ *(the* vocabulary*) and* $\mathcal{L} = \mathcal{V} \cup V_N \cup V_N^0$ *(the* transition labels*) where* V_N^0 *is the set of equations of the form* $N = 0$ *with* $N \in V_N$. *An* index graph *is a tuple* (S, Σ, δ, F), *where* $S \in \Sigma$ *is the* initial state, Σ *is a set of states,* $F \subseteq \Sigma$ *is the set of* final (accepting) states *and* $\delta : \Sigma \times \mathcal{L} \to \Sigma$ *is a (partial)* transition function[4].

[3] The simplest way to ensure this property is to restrict oneself to N-terms of the form $t^N.a$, where a is a constant symbol, and assume that the ordering is chosen in such a way that a constant cannot be replaced by a complex term.

[4] We write $\delta(p, \mu) = \bot$ iff $\delta(p, \mu)$ is undefined.

If $\delta(p,\mu) = q \neq \bot$ then the graph contains a *transition* from p to q labeled with μ. q is a *successor* of p. A transition labeled by an element of a set $E \subseteq \mathcal{L}$ (resp. by a label $\mu \in \mathcal{L}$) is called an *E-transition* (resp. μ-transition). The language denoted by an index graph is defined as usual [11].

We briefly recall some basic operations on index graphs [11]. If f is a symbol in $\mathcal{V}$ then we also denote by f (when no confusion is possible) an arbitrary graph of the form $(p, \{p,q\}, \{(p,f) \mapsto q\}, \{q\})$ where $p \neq q$ (i.e. a graph accepting $\{f\}$). $IG_1 \bullet \ldots \bullet IG_n$ denotes the sequential composition of the graphs $IG_1, \ldots, IG_n$. Intuitively, the language denoted by the obtained graph corresponds to the concatenation of the ones of $IG_1, ..., IG_n$ in this order. The graphs $IG_1, \ldots, IG_n$ are assumed to be disjoint (the states are renamed if needed). $IG_1|IG_2$ denotes the graph obtained from $IG_1 \cup IG_2$ by merging the initial states of IG_1 and IG_2 (IG_1 and IG_2 are assumed to be disjoint). The language denoted by $IG_1|IG_2$ is the union of the ones denoted by IG_1, IG_2. This operation is well-defined iff there is no label f s.t. both IG_1 and IG_2 contain an f-transition from their initial state. Finally, $IG[p/q]$ denotes the graph obtained from IG by replacing the state p by the state q. Every edge pointing to p is redirected to q. This operation can obviously be extended to multiple replacements of states p_i by states q_i $(1 \leq i \leq n)$. These operations are standard thus the formal definitions are omitted.

4 Construction of the Index Graph

We create for each term or context t an index graph denoted by $IG(t)$ which represents t. $IG(t)$ is inductively defined as follows.

1. If $t = f(t_1, t_2, \ldots, t_n)$ (with possibly $n = 0$ and/or $f \in V_X$) then $IG(t) = f \bullet IG(t_1) \bullet \ldots \bullet IG(t_n)$.
2. Assume that $t = f(t_1, \ldots, t_n)^N.s$. Let $IG_i = IG(t_i) = (S_i, \Sigma_i, \delta_i, F_i)$ for $1 \leq i \leq n$. Let $IG_s = IG(s) = (S_s, \Sigma_s, \delta_s, F_s)$ and let $IG_s^0 = (S_s^0, \Sigma_s^0, \delta_s^0, F_s^0)$ be a copy of IG_s.
   The index graph of t is composed of two branches IG_b^0 and $IG_{N\neq 0}$ corresponding respectively to $N = 0$ and $N \neq 0$. The latter branch is obtained by concatenation of two graphs IG_c and IG_b corresponding respectively to the iterated context and base term. More precisely, $IG_b^0, IG_c, IG_b, IG_{N\neq 0}$ and IG are defined as follows:
   (a) IG_b and IG_b^0 are obtained from IG_s and IG_s^0 respectively by adding for each state $p_s \in F_s$ (resp. $p_s^0 \in F_s^0$) a new state q_s (resp. q_s^0) and a transition from p_s to q_s (resp. from p_s^0 to q_s^0) labeled by N (resp. by $N = 0$). The final states of IG_b and IG_b^0 are the states q_s, q_s^0.
   (b) IG_c is obtained from $IG_1 \bullet \ldots \bullet IG_n$ by adding a new state r^q and a #-transition from q to r^q for every final state q in IG_n. Each #-transition creates a cycle in the graph (encoding iterated unfolding).
   (c) $IG_{N\neq 0}$ is obtained from $IG_c \bullet IG_b$ by adding an f-transition from each r^q to the initial state of IG_c.
   (d) $IG = IG_b^0|(f \bullet IG_{N\neq 0})$.

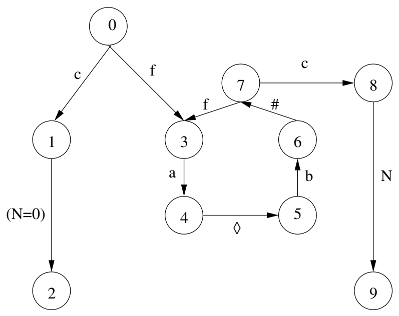


Fig. 1. The index graph of $f(a, \diamond, b)^N.c$

Notice that the merging operation used in the last item is well defined since we consider only eligible I-terms. Considering two distinct copies of the base term IG_s and IG_s^0 is necessary to avoid ambiguities when distinct N-terms share the same base term (see Remark 1 page 380).

Due to space restriction, we only present the construction of the index graph of a single term. This technique may be extended to index *simultaneously* sets of I-terms. The corresponding graph is constructed from an initially empty one by inserting successively the index graph of each I-term in the set. The procedure inserting a new term t into an existing graph IG must ensure as much as possible common prefix sharing in the resulting graph. This may be done for instance by first constructing the graph $IG(t)$ (assumed to be disjoint from IG) and then by replacing iteratively the states in $IG(t)$ (starting from the initial one) by states from IG, until we reach a transition that does not occur in IG. Then the resulting index is simply the union of IG and $IG(t)$. We do not detail the algorithm since it is actually very similar to the one for trees. The retrieval algorithm presented in the next section handles indexes containing multiple indexed terms (see for instance Remark 1 page 380).

For every index graph IG and for every final state p in IG we denote by $\theta_{IG}(p)$ the I-term corresponding to the state p.

5 The Filtering Algorithm

In this section, $IG = (S, \Sigma, \delta, F)$ denotes a given index graph. We describe an algorithm for finding all the (indexed) terms t that filter a candidate term s, i.e. the set $\{t \mid t \in \theta_{IG}(F), \exists \sigma \, t\sigma \downarrow = s\}$ where σ is a substitution. The main difficulty is to handle N-terms in the input term and $(\diamond, \#)$-transitions in the index ($\mathcal{F}$-terms and $V_X \cup \mathcal{F}$-transitions are handled exactly as usual). The following lemma introduces a crucial property of eligible N-terms on which our algorithm is based:

Lemma 1. *Let w and $u^M.v$ be non variable eligible I-terms. If there exists a substitution σ s.t. $w\sigma \downarrow = u^M.v$ then:*

– *w is of the form $t^N.s$.*
– $\exists i \in \mathbb{N}, N\sigma = i.M$.
– *$t^i\sigma \downarrow = u$ and $s\sigma \downarrow = v$, where $t^i = \underbrace{t[t[\ldots[\diamond]_q \ldots]_q]_q}_{i}$.*

Notice that Lemma 1 does not hold if w is non eligible. As a consequence of Lemma 1, if the input term is an N-term $u^M.v$ then the current state in the index graph must also correspond to an N-term $t^N.s$[5]. Furthermore, the matching can be decomposed into two parts, corresponding respectively to the inductive context u and to the base term v. Then the value of the exponent N is $i.M$, where i is the number of unfoldings of t that are necessary to match u.

We now provide an informal description of the retrieving algorithm based on this decomposition. Variables and function symbols are handled exactly as usual. When we read an input term of the form $u^N.v$, we store the base term v and the exponent N into a stack `BT` and we try to match the inductive context u. During the handling of u, a $\diamond$-transition will eventually be reached. Then the I-term corresponding to the current loop in the index graph must be unfolded in order to match the input term (for instance the context $f(\diamond)$ must be unfolded once to match $f(f(\diamond))$). Unfortunately, the context cannot be unfolded at this point since we have not reached the #-transition pointing to its initial state. Consequently, we must use another stack `H` to store the input term, in order to delay its treatment until the remaining part of the context is filtered. If we reach a #-transition, a term is popped from `H`. If this term is distinct from $\diamond$ then the index term is unfolded to match the remaining part of the inductive context. Otherwise the iterated context u has been entirely filtered, thus we start processing the base term (stored in `BT`). After successfully filtering the base term, we apply the N-transition, which allows us to compute the value of the exponent. The base term and the exponent (still) stored in `BT` are then popped from it.

We also need to remind (using another stack `L`) the first state corresponding to the loop of the index term (and the head function symbol), in order to check that the first transition that is reached after matching the inductive context really loops to this state (see Example 5 for more details).

In case two distinct loops share a common prefix, we have to ensure that the same transitions are applied at each iteration. This is done by marking all the encountered states (using a set *mk*). If a successor of the current state is marked then we simply ignore all the other transitions. The usefulness of *mk* is well evidenced by the following example.

Example 3. Consider the input term $f(a, f(a, c, b), d)$ and an index containing the terms $f(a, \diamond, b)^N.c$ and $f(a, \diamond, d)^N.e$. The inductive contexts of these two terms share the prefix $fa\diamond$. Starting from the initial state of the index graph, we read successively f, a and we reach a $\diamond$-transition, pointing to a state p. As explained before, we need to store the term $f(a, c, b)$ in `H`. Next we read the symbol d and we reach the #-transition. The term $f(a, c, b)$ is popped from `H`

[5] Notice that due to the sharing, the current state may actually correspond to *several* terms, including both N-terms and $\mathcal{F}$-terms.

and the inductive context of the index graph is unfolded. We read the symbol f, a and we reach the state p again. The term c is stored and we try to match the term b. However, the inductive context has already been unfolded, and we know that this context is $f(a, \diamond, d)$ and *not* $f(a, \diamond, b)$. Consequently, the transition b (that is not marked) can be ignored and the algorithm fails (no valid transition). Without marking, the b-transition would be applicable which would yield an unsound solution.

In order to find the value of the exponent variables, we need to count the number of times we enter into the corresponding loop in the index graph. This can be done by counting the number of #-transitions. However, a unique counter is not sufficient. Indeed, the unfolding can correspond either to the matching of a standard term (e.g. if $f(f(f(a)))$ is matched by $f(\diamond)^N.a$) or to the one of an inductive context (if $f(f(\diamond))^N.a$ is matched). We use a counter b for the first kind of unfolding and another counter a for the second one. Both counters are stored into a stack ($\mathtt{Cnt}$), together with the corresponding state in the index graph. We also need to store a flag that indicates the kind of terms that is currently handled (I-terms or context). A stack $\mathtt{Ctx}$ is used for this purpose. Notice that the two counters can be used on the same time e.g. if the index contains $t = f(\diamond)^N.c$ and the input term is $s = \underbrace{f(f}(\underbrace{f(f(f(\diamond)))^M}.c))$ which is an instance of t. The b-counter is used to count the number of unfoldings before reaching the nested N-term (2 in this case) and the a-counter is used to count the number of loops needed to match $f(f(f(\diamond)))$ using $f(\diamond)$ (3 in this case). We have $s = t\sigma$ with $\sigma = \{N \mapsto 3.M + 2\}$.

Finally, we use a set of equations E to store the value of the variables occurring in the index graph (both the ordinary variables and the arithmetic ones). The algorithm fails if this set of equations is unsatisfiable (considering variables of the input term as constant).

Formally, the algorithm handles tuples of the form $\mathcal{C} = (q, \mathtt{W}, \mathtt{BT}, \mathtt{H}, \mathtt{Cnt}, \mathtt{Ctx}, \mathtt{L}, E, mk)$ where:

- $q \in \Sigma$ the *current state*, $\mathtt{W} \in (\Xi \cup \Xi_\diamond)^*$ is a stack used to store *remaining terms to handle*. Initially $q = S$ and $\mathtt{W} = s$ where s is the query term.
- $\mathtt{BT} \in (\Xi \times V_N)^*$ is a stack used for storing base terms and exponents of eligible I-terms.
- $\mathtt{H} \in (\Xi \cup \Xi_\diamond)^*$ is a stack used for storing terms occurring in hole positions during filtering.
- $\mathtt{Cnt} \in (\Sigma \times \mathbb{N} \times \mathbb{N})^*$ is a stack used by #-transitions to store couples of integers used as counters. The first component is used to identify the state used in the last #-transition, so that we can determine if it is the first time we visit a #-transition or not.
- $\mathtt{Ctx} \in (\{0, 1\} \times \mathbb{N} \times \mathbb{N})^*$ is a stack used to hold the type of the current open context (the current I-term not entirely filtered): 0 for $\mathcal{F}$-terms, 1 for N-terms. The second and third components are used only for non constant functional terms, they denote respectively the arity of the function symbol and the number of arguments remaining to handle.
- $\mathtt{L} \in (\Sigma \times \mathcal{F})^*$ is a stack used for filtering N-terms.

- E is a set of equations that are either of the form $N = a.M + b$ where $N, M \in V_N$ and $a, b \in \mathbb{N}$ or of the form $x = t$ where $x \in V_X$ and $t \in \Xi$.
- *mk* is a set of *marked* states.

We call such tuples a *configuration*. Initially we have $\mathcal{C} = \mathcal{C}_0 = (S, s, \epsilon, \epsilon, \epsilon, \epsilon, \epsilon, \emptyset, \{S\})$, where ϵ denotes an empty stack. $\mathcal{CF}$ denotes the set of configurations.

Let *empty*, *top*, *pop*, *push* be functions operating on stacks. *empty* and *top* return respectively a boolean indicating whether the stack is empty and the first element of a non empty stack, *pop* returns the first element and removes this element from the stack and *push* adds a new element in the stack.

The next definition states sufficient conditions ensuring that a given transition is valid in a given configuration. Transitions labeled by a function or an ordinary variable are handled as usual, whereas transitions labeled by $\diamond$, $\#$, by an arithmetic variable or by an equation $N = 0$ are treated as ϵ-transitions (no condition on the input). Furthermore, the reached state must be marked *if this is possible.*

Definition 4. *Let* $\mathcal{C} = (q, \mathtt{W}, \mathtt{BT}, \mathtt{H}, \mathtt{Cnt}, \mathtt{Ctx}, \mathtt{L}, E, mk)$ *be a configuration. We say that a transition* $(q, \mu) \mapsto p$ *is* valid *in* $\mathcal{C}$ *if and only if:*

1. $\delta(q, \mu) = p$ *and one of the following conditions holds:*
   - $\mu \in \{\diamond, \#\} \cup V_N \cup V_N^0$.
   - $\mu \in V_X$, $top(\mathtt{W}) = t$ *where* $t \in \Xi$ *and* $t \notin \Xi_\diamond$.
   - $\mu = f$, $top(\mathtt{W}) = t$ *where* $t \in (\Xi \cup \Xi_\diamond)$ *and* t *is of the form* $f(t_1, \ldots, t_n)$ *or* $f(t_1, \ldots, t_n)^N.s$.
2. *Either* $p \in mk$ *or* $\Lambda \cap mk = \emptyset$ *where* Λ *is a set of nodes defined as follows:* Λ *is the set of* all *successors of* q *if* q *does not occur right after a* $\#$*-transition and otherwise, the set of all successors* p' *of* q *s.t.* $(q, \nu) \mapsto p'$ *and* $top(\mathtt{L}) \neq (p', \nu)$ *(hence, we discard the first state of the iterative context loop which has already been marked, and we consider only states belonging to base terms).*

`FindGeneralisation` implements the matching algorithm.

```
Function FindGeneralisation($\mathcal{C}$)
  Input: $\mathcal{C} = (q, \mathtt{W}, \mathtt{BT}, \mathtt{H}, \mathtt{Cnt}, \mathtt{Ctx}, \mathtt{L}, E, mk)$ the configuration for which to find a
         generalisation
  $IG = (S, \Sigma, \delta, F)$ /* the index graph */
  Output: $(\theta_{IG}(p), E)$, a solution of the generalisation finding problem. $\theta_{IG}(p)$ is
          the term corresponding to the final state and $E$ are the constraints on
          the variables in $\theta_{IG}(p)$.
  if ($empty(\mathtt{W})$) and ($q \in F$) then
      /* We are in an accepting state */ ;
      return $(\theta_{IG}(q), E)$ ;
  else if there is no valid transition $(q, \mu) \mapsto p$ in $\mathcal{C}$ then
      fail;
  else
      $\mathcal{C}_n$ = ComputeNextConfiguration($\mathcal{C}$) ;
      FindGeneralization($\mathcal{C}_n$) ;
```

```
Function ComputeNextConfiguration($\mathcal{C}$)
  Input: $\mathcal{C}$: the configuration to handle
  /* $\mathcal{C} = (q, \mathtt{W}, \mathtt{BT}, \mathtt{H}, \mathtt{Cnt}, \mathtt{Ctx}, \mathtt{L}, E, mk)$ */ ;
  Output: $\mathcal{C}_n$: a next configuration
  choose a valid transition $(q, \mu) \mapsto p$ ;
  $t = pop(\mathtt{W})$; /* The current term to handle */ ;
  $q = p$ ;
  $mk = mk \cup \{p\}$ ;
  switch $\mu$ do
      case $\mu \in V_X$
          $E = E \cup \{\mu = t\}$ ;
          DiscardArgOfOpenTerm(Ctx) ;
          if $E$ is not satisfiable then
              fail
      case $\mu = \diamond$
          $push(t, \mathtt{H})$ ;
      case $\mu = \#$
          ManageCounters(Cnt,$q$,Ctx) ;
          $r = pop(\mathtt{H})$ ;
          if $r = \diamond$ then
              $(s, N) = top(\mathtt{BT})$ ;                /* Base term and exponent */
              $(r, f) = pop(\mathtt{L})$ ;
              $push(s, \mathtt{W})$ ;                     /* Next term to handle is $s$ */
              if $\delta(p, f) \neq r$ then
                  fail
          else
              $push(r, \mathtt{W})$ ;
      case $\mu \in V_N$
          $(s, N) = pop(\mathtt{BT})$ ;
          $(r, a, b) = pop(\mathtt{Cnt})$ ;
          $E = E \cup \{\mu = a.N + b\}$ ;
          $pop(\mathtt{Ctx})$ ;
          DiscardArgOfOpenTerm(Ctx) ;
          if $E$ is not satisfiable then
              fail
      case $\mu \in V_N^0$
          $E = E \cup \{\mu = 0\}$ ;
          if $E$ is not satisfiable then
              fail
      case $\mu \in \mathcal{F}$ /* $t = f(t_1, ..., t_k)$ or $t = f^N(t_1, ..., t_k).s$ */
          $push(t_k, \mathtt{W})$; $push(t_{k-1}, \mathtt{W})$; $\cdots$ ; $push(t_1, \mathtt{W})$ ;
          if $t \notin \Xi_\diamond$ then
              PushOverOpenTerm($t$,Ctx);
          if $t = f^N(t_1, ..., t_k).s$ then
              $push((p, f), \mathtt{L})$ ;
              $push((s, N), \mathtt{BT})$ ;
  $\mathcal{C}_n = (q, \mathtt{W}, \mathtt{BT}, \mathtt{H}, \mathtt{Cnt}, \mathtt{Ctx}, \mathtt{L}, E, mk)$ ;
  return $\mathcal{C}_n$ ;
```

```
Procedure ManageCounters(Cnt,$q$,Ctx)
  Input: Cnt the counter stack handling $a$- and $b$-counters; $q$ the current state; Ctx
         the open contexts stack
  Result: Modifies Cnt according to the type of handled terms and wether it is the
          first visit or not to the current state
  if $empty$(Cnt) $or$ ($top$(Cnt) $= (p, a, b)$ $and$ $p \neq q$) then   /* This is the first
  time we visit this state */
      if $top$(Ctx) $= (0, x, y)$ then  /* the current open term is an $\mathcal{F}$-term. We
      must use the $b$-counter */
          $push((q, 0, 1),$ Cnt$)$ ;
      else
          $push((q, 1, 0),$ Cnt$)$ ;
  else
      $(p, a, b) = pop($Cnt$)$ ;
      if  $top$(Ctx) $= (0, x, y)$ then /* the current open term is an $\mathcal{F}$-term. We
      must use the $b$-counter */
          $push((p, a, b + 1),$ Cnt$)$ ;          /* we increment the $b$-counter */
      else
          $push((p, a + 1, b),$ Cnt$)$ ;          /* we increment the $a$-counter */
```

It uses the non deterministic function ComputeNextConfiguration which computes the configurations produced by all the possible valid transitions.

The handling of the stack Cnt is ensured by the procedure ManageCounters. When we read a term (not a context), we push in Ctx its type (0 if it is an $\mathcal{F}$-term, 1 if it is an N-term). If this term is an $\mathcal{F}$-term, we also store the arity of the head symbol and the number of remaining arguments. A $\mathcal{F}$-term is entirely handled when all its arguments are handled. It is then removed from Ctx. An N-term is entirely handled if both the iterated context and the base term are handled. The procedure PushOverOpenTerm is used for managing Ctx.

```
Procedure PushOverOpenTerm(Ctx,$t$)
  Input: Ctx the context stack; $t \in \Xi$ the next term to process
  if  $t = f^N(t_1, ..., t_n).s$ then
      $push((1, 0, 0),$ Ctx$)$ ;
  else
      if $t = f(t_1, ..., t_n)$ $and$ $n = arity(f) > 0$ then
          $push((0, n, n),$ Ctx$)$ ;
      else
          DiscardArgOfOpenTerm(Ctx) ;  /* $t$ is a constant or a variable */
```

Handling N-terms is easy: they are removed from the stack when an N-transition is reached. Handling $\mathcal{F}$-terms is slightly more difficult. When we read a constant or a variable, the counter of remaining arguments on the top of Ctx is decreased. When this counter reaches 0, the term is removed from the stack. This operation is repeated until the stack is empty or until an N-term is reached.

```
Procedure DiscardArgOfOpenTerm(Ctx)
  Input: Ctx the context stack
  /* we iteratively close (discard) imbricated simple open terms having
  only one remaining argument */ ;
  while $\exists n \in \mathbb{N}$, $top(\mathtt{Ctx}) = (0, n, 1)$ do
    $pop(\mathtt{Ctx})$ ;
  if $top(\mathtt{Ctx}) = (0, n, d)$ where $(d > 1)$ then
    $(0, n, d) = pop(\mathtt{Ctx})$ ;
    $push((0, n, d-1), \mathtt{Ctx})$ ;
```

When a term is popped from `Ctx` the new counter of remaining arguments of the top of `Ctx` must be recursively decreased. The procedure `DiscardArgOfOpenTerm` implements this operation.

Example 4. The following example illustrates the usefulness of `Ctx`. We consider the indexed term $t = f(\diamond)^N.a$ and the input term $s = f(f(f(f(\diamond))^M.a))$. The index graph of t contains a loop for the iterated part $f(\diamond)$. The first time we reach the corresponding #-transition, the current term is $f(f(f(f(\diamond))^M.a))$ which is not an N-term. Therefore we must use the b-counter. The current term is unfolded again, yielding the term $f(f(f(\diamond))^M.a)$. The b-counter is used since this term is an $\mathcal{F}$-term. After the next unfolding, when we reach #, the current term is the N-term $f(f(\diamond)))^M.a$, thus the a-counter must be used. The role of `Ctx` is to keep track of the type of the current term in order to determine which counter must be used (a or b). Back to our example, before the loop, we push in `Ctx` the tuple $(0, 1, 1)$ meaning that we are handling an $\mathcal{F}$-term (0), with a head symbol of arity 1 and that 1 argument remains to handle. When we reach #, we find in the top of `Ctx` a tuple $(0, x, y)$ which indicates that we must use the b-counter. The next term to handle is $f(f(f(\diamond))^M.a)$ which is again an $\mathcal{F}$-term. We push in `Ctx` the tuple $(0, 1, 1)$. In the #-transition, we use again the b-counter (for the same reasons of the first loop). The next term to handle is the N-term $f(f(\diamond))^M.a$, thus we push in `Ctx` the tuple $(1, 0, 0)$. When we reach the #-transition, the top of `Ctx` drives us to use the a-counter. The next term to handle is a context $f(\diamond)$, thus no information is pushed in `Ctx` and the top of the stack still contains $(1, 0, 0)$. After a new loop, when we reach the #-transition we use again the a-counter. The next term to handle is $\diamond$ which is discarded from `Ctx`. $\diamond$ means that the iterated context is entirely handled. Next we consider the base term a. It is a constant and hence it is discarded. The top of `Ctx` is not affected because it corresponds to an N-term. Finally we reach an N-transition, `Ctx` is popped and the procedure `DiscardArgOfOpenTerm` is called. $top(\mathtt{Ctx})$ is successively $(0, 1, 1)$ and $(0, 1, 1)$, and then `Ctx` becomes empty.

The next remark explains why we needed to duplicate the index graph of the base term in Section 4.

Remark 1. Suppose that we use only one index graph for the base term (following the notations of Section 4, we use IG_s in both cases instead of IG_s^0) and suppose that we have to index the two terms $h(f(\diamond)^N.a, b)$ and $h(g(\diamond)^M.a, c)$. The resulting index graph is presented in Figure 2.

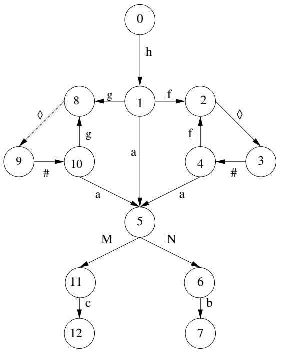


Fig. 2. Indexing $h(f(\diamond)^N.a, b)$ and $h(g(\diamond)^M.a, c)$

Now if we try to match the term $h(g(\diamond)^N.a, b)$ which is neither an instance of $h(f(\diamond)^N.a, b)$ nor of $h(g(\diamond)^M.a, c)$, we will succeed following for example the path $0 \rightarrow 1 \rightarrow 8 \rightarrow 9 \rightarrow 10 \rightarrow 5 \rightarrow 6 \rightarrow 7$. The reason of this undesired behaviour is that if we use a single index graph for the base term, then the index graphs of two N-terms having the same base term and completely different inductive contexts will share a common state (5 in the previous example) which can be reached from two different paths, leading to inconsistencies. This problem is quite similar in essence to the one that leaded us to use marked states, namely the existence of states reachable from many different paths (e.g. the state 5 in the example). However, the chosen solution is different in this case. Actually marks are useful for states occurring inside loops where common states can be visited several times. But if the state is outside a loop (as in this example), marks cannot prevent incoherences since the state will be visited only once. The only solution is to distinguish between the case were the base term is filtered directly (without handling the iterated context) and the case where it is filtered after having handled the iterated context. Thus, even if two N-terms share a common base term, they will share only states belonging to the index graph of the common base term. A possible optimisation is to duplicate the base term index graph only when this is needed, namely when we insert a new term to the index having the same base term as an existing indexed term (this optimisation is straightforward, but it is not considered in the present paper, for the sake of readability).

Finally, we illustrate the usefulness of the stack L with an example.

Example 5. Consider the terms $t = f(g(\diamond)^N.a)$ and $s = f(g(\diamond))^N.a$. Obviously, t is not a generalisation of s. However if we represent the two terms without parenthesis and without the transition corresponding to the cycle, we obtain the same representation, namely $fg \diamond \#aN$ and $fg \diamond \#aN$. To solve this problem, we have to take into account the state pointed by the #-transition. Using this state, the two representations can be distinguished provided the initial state of the loop and the corresponding head symbol are available, which justifies the use of L. In our example, after handling the iterated context $f(g(\diamond))$ we will find on the top of `L` a pair (p, f) but in the index graph we have no valid transition driven by f (the iterated context in the index is $g(\diamond)$ therefore the only valid transition is driven by g) and the matching will fail.

The following theorem states the soundness and completeness of our algorithm.

Theorem 1. *Let IG be an index graph, let t be an eligible term. The two following assertions are equivalent:*

- *There exist a state S in IG and a substitution σ s.t. $\theta_{IG}(S)\sigma\downarrow = t$.*
- *There is a run of FindGeneralisation($\mathcal{C}_0$) that returns $(\theta_{IG}(S), E)$, where σ is a solution of E.*

The proof follows from the previous explanations and from the key lemma [1]. The complexity of the function `FindGeneralisation` is obviously linear w.r.t the size of the input term.

6 Conclusion

The presented work is a natural continuation of [2]. In [2] we described the theorem prover DEI, an extension of the well-known E-prover [17]. The extension allows for compact, more expressive clauses containing the so-called I-terms, i.e. schemata denoting (possibly infinite) sets of standard first-order terms. One of the main motivations for using this formalism is to improve the termination behaviour of the calculus, which permits satisfiability detection and proof by consistency (in inductive theorem proving). This obviously requires simplification and redundancy elimination rules. In order to implement such rules effectively, and to use the implementation for *extensive* experimentation (which are obviously beyond the scope of the present paper), powerful indexing techniques similar to that used for standard terms are needed, e.g. *perfect discrimination trees*. Originally motivated by implementation needs, adapting perfect discrimination trees to deal with I-terms turned out to be a difficult task and appeared to be a research subject per se.

The algorithm is described in details (using pseudocode to make the implementation easier). The main property ensuring its soundness is proven and several examples illustrate the construction of the index graph (called perfect discrimination graphs) allowing to represent schemata and to retrieve subsuming terms of a given (query) term. Other index maintenance techniques (e.g. deletion) are on going work.

Term schematisation languages are expressive formalisms that have been studied so far essentially from a theoretical perspective. Our work is, to the best of our knowledge, the first attempt to devise efficient implementation techniques for such languages. It has to our opinion, at least two positive aspects: it will allow a much more efficient implementation of DEI necessary to *extensive* practical experimentation and, in some sense a partial consequence of this, the experimentation will very probably provide hints to solve a *theoretical problem*, i.e. characterize an interesting subclass of I-terms whose presence in formulae will preserve completeness of the superposition calculus. We are presently working in both directions.

References

1. Bensaid, H., Caferra, R., Peltier, N.: Towards systematic analysis of theorem provers search spaces: First steps. In: Leivant, D., de Queiroz, R. (eds.) WoLLIC 2007. LNCS, vol. 4576, pp. 38–52. Springer, Heidelberg (2007)
2. Bensaid, H., Caferra, R., Peltier, N.: Dei: A theorem prover for terms with integer exponents. In: Schmidt, R.A. (ed.) Automated Deduction – CADE-22. LNCS, vol. 5663, pp. 146–150. Springer, Heidelberg (2009)
3. Bouhoula, A., Kounalis, E., Rusinowitch, M.: Spike, an automatic theorem prover. In: Voronkov (ed.) [18], pp. 460–462 (1992)
4. Bouhoula, A., Rusinowitch, M.: Implicit induction in conditional theories. Journal of Automated Reasoning 14, 14–189 (1995)
5. Bundy, A.: The automation of proof by mathematical induction. In: Robinson, Voronkov (eds.) [15], pp. 845–911 (2001)
6. Chen, H., Hsiang, J., Kong, H.: On finite representations of infinite sequences of terms. In: Okada, M., Kaplan, S. (eds.) CTRS 1990. LNCS, vol. 516, pp. 100–114. Springer, Heidelberg (1991)
7. Comon, H.: Inductionless induction. In: David, R. (ed.) 2nd Int. Conf. in Logic For Computer Science: Automated Deduction. Lecture notes, Chambéry, Univ. de Savoie (1994)
8. Comon, H.: On unification of terms with integer exponents. Mathematical Systems Theory 28(1), 67–88 (1995)
9. Hermann, M., Galbavý, R.: Unification of infinite sets of terms schematized by primal grammars. Theor. Comput. Sci. 176(1-2), 111–158 (1997)
10. Hillenbrand, T., Buch, A., Vogt, R., Löchner, B.: Waldmeister - high-performance equational deduction. J. Autom. Reason. 18(2), 265–270 (1997)
11. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation, 2nd edn. Addison-Wesley, Reading (2000)
12. McCune, W.: Experiments with discrimination-tree indexing and path indexing for term retrieval. J. Autom. Reasoning 9(2), 147–167 (1992)
13. Peltier, N.: Increasing model building capabilities by constraint solving on terms with integer exponents. Journal of Symbolic Computation 24(1), 59–101 (1997)
14. Ramakrishnan, I.V., Sekar, R.C., Voronkov, A.: Term indexing. In: Robinson, Voronkov (eds.) [15], pp. 1853–1964
15. Robinson, J.A., Voronkov, A. (eds.): Handbook of Automated Reasoning (in 2 volumes). Elsevier, Amsterdam (2001)
16. Salzer, G.: The unification of infinite sets of terms and its applications. In: [18], pp. 409–420 (1992)
17. Schulz, S.: System description: E 0.81. In: Basin, D., Rusinowitch, M. (eds.) IJCAR 2004. LNCS (LNAI), vol. 3097, pp. 223–228. Springer, Heidelberg (2004)
18. Voronkov, A. (ed.): LPAR 1992. LNCS, vol. 624. Springer, Heidelberg (1992)
19. Walsh, T.: A divergence critic for inductive proof. Journal of Artificial Intelligence Research 4(1), 209–235 (1996)

An Interpolating Sequent Calculus for Quantifier-Free Presburger Arithmetic*

Angelo Brillout[1], Daniel Kroening[2], Philipp Rümmer[2], and Thomas Wahl[2]

[1] ETH Zurich, Switzerland
[2] Oxford University Computing Laboratory, United Kingdom

Abstract. Craig interpolation has become a versatile tool in formal verification, for instance to generate intermediate assertions for safety analysis of programs. Interpolants are typically determined by annotating the steps of an unsatisfiability proof with *partial interpolants*. In this paper, we consider Craig interpolation for full *quantifier-free Presburger arithmetic* (QFPA), for which currently no efficient interpolation procedures are known. Closing this gap, we introduce an *interpolating sequent calculus* for QFPA and prove it to be sound and complete. We have extended the PRINCESS theorem prover to generate interpolating proofs, and applied it to a large number of publicly available linear integer arithmetic benchmarks. The results indicate the robustness and efficiency of our proof-based interpolation procedure.

1 Introduction

Craig interpolation [3], a principle known to logicians since the 1950s, has recently emerged in formal verification as a practical approximation method. Its applications range from efficient image computations in SAT-based model checking to accelerating convergence of fixpoint calculations for infinite-state systems. Given two formulae A and C such that A implies C, written $A \Rightarrow C$, an interpolant is a formula I such that $A \Rightarrow I$, $I \Rightarrow C$, and I contains only non-logical symbols occurring in both A and C. Interpolants exist for any two first-order formulae A and C such that $A \Rightarrow C$. As is common in formal verification, we also consider interpolation for unsatisfiable conjunctions $A \wedge B$, which corresponds to $C = \neg B$ in the above formulation.

In software verification, interpolation is applied to formulae encoding the transition relation of a model underlying a program. In order to support expressive programming languages, much effort has been invested in the design of algorithms that compute interpolants for formulae of various theories. As a result, efficient interpolation methods are known for propositional logic, linear arithmetic over the reals with uninterpreted functions [10,1,16], datastructures like arrays and sets [7], and other theories. As for integer arithmetic, a theory particularly relevant for software, interpolating solvers have so far been reported only

* Supported by the Engineering and Physical Sciences Research Council (EPSRC) under grant no. EP/G026254/1, by the EU FP7 STREP MOGENTES, and by the EU ARTEMIS CESAR project.

for restricted fragments such as difference-bound logic, and logics with linear equalities and constant-divisibility predicates. For these theories, an interpolant can be derived in time polynomial in the size of the input formulae.

In this paper, we push the boundaries of interpolation-based software model checking by presenting an interpolation method for full quantifier-free *Presburger arithmetic* (QFPA), i.e., linear arithmetic over the integers. This theory has been used, besides others, to model the behavior of infinite-state programs and of hardware designs. Presburger arithmetic was shown to be decidable by quantifier elimination [12]. A brute-force interpolation method is to quantify out the variables not common to the input formulae, and then to eliminate those quantifiers. This approach suffers, however, from the triply-exponential complexity of the elimination procedure and tends to be ineffective in many practical cases.

A more promising approach (that has also been used, e.g., in [10,1,8,5]) is to extract interpolants directly from an unsatisfiability proof for $A \wedge B$. To this end, we first present a sound and complete proof system for QFPA based on a *sequent calculus*. We then augment the proof rules with labeled formulae and *partial interpolants* — proof annotations that, at the root of a closed proof, reduce to interpolants. In practice, the resulting interpolating proof system can be used to extend an existing unsatisfiability proof to one that interpolates. It can also serve as a replacement of the non-interpolating proof system, allowing the calculation of an interpolant on the fly. We prove our interpolating calculus to be *sound and complete* for QFPA. Our completeness result states that, for any valid implication, there exists a proof of its validity in our calculus, and the proof can be annotated with partial interpolants satisfying the proof rules.

In the case of QFPA, the primary difficulty when extracting interpolants from a proof is the treatment of *mixed cuts*: applications of a cut-rule (such as Gomory cuts [17] or the Omega rule [13]) to inequalities that have been derived as linear combinations of inequalities from both A and B. Our work extends earlier interpolation procedures for linear arithmetic, in particular [8,10], by defining an interpolating cut-rule called STRENGTHEN that can handle even mixed cuts. The rule subsumes a variety of cut-rules for integer linear programming, including Gomory cuts and the Omega rule, so that interpolants can be extracted from proofs using either of those rules by reduction to STRENGTHEN.

To implement our interpolation method, we have extended the PRINCESS theorem prover [15] to generate proofs, using the proof rules presented in this paper. We have applied the interpolating prover to a large number of publicly available linear integer arithmetic benchmarks, such as from the QF-LIA category of the SMT library. We compare the efficiency of the prover to the only currently known interpolation method for Presburger arithmetic, which is based on local-variable quantification and subsequent brute-force quantifier elimination (QE). Our experiments not only demonstrate the weaknesses of interpolation using QE, but also indicate the robustness and efficiency of our proof-based interpolation procedure, in terms of both time and interpolant size.

2 Preliminaries

Presburger arithmetic. We assume familiarity with classical first-order logic (e.g., [4]). Let x range over an infinite set X of variables, c over an infinite set C of constant symbols, and α over the integers $\mathbb{Z}$. The syntax of *Presburger arithmetic* is defined by the following grammar:

$$\begin{aligned} \phi \ &::= \ t \doteq 0 \;\Big|\; t \leq 0 \;\Big|\; \alpha \mid t \;\Big|\; \phi \wedge \phi \;\Big|\; \phi \vee \phi \;\Big|\; \neg\phi \;\Big|\; \forall x.\phi \;\Big|\; \exists x.\phi \\ t \ &::= \ \alpha \;\Big|\; c \;\Big|\; x \;\Big|\; \alpha t + \cdots + \alpha t \end{aligned}$$

The symbol t denotes terms of linear arithmetic. For simplicity, we only allow 0 as the right-hand side of equalities and inequalities. The explicit divisibility operator $\alpha \mid t$, which is short for $\exists s.\ \alpha s - t \doteq 0$, is included to permit quantifier-free interpolants for formulae such as $y - 2x \doteq 0 \wedge y - 2z - 1 \doteq 0$, with interpolant $2 \mid y$. We use the abbreviations *true* and *false* for the equalities $0 \doteq 0$ and $1 \doteq 0$, and $\phi \rightarrow \psi$ as abbreviation for $\neg\phi \vee \psi$. Simultaneous substitution of terms $t_1, \ldots, t_n$ for variables $x_1, \ldots, x_n$ in ϕ is denoted by $[x_1/t_1, \ldots, x_n/t_n]\phi$; we assume that variable capture is avoided by renaming bound variables as necessary. As short-hand notation, we sometimes also quantify over constants (as in $\forall c.\phi$) and assume that the constants are implicitly replaced by fresh variables. For reasons of presentation, we further assume that terms t are implicitly simplified to 0 or to the form $\alpha_1 t_1 + \cdots + \alpha_n t_n$, in which $0 \notin \{\alpha_1, \ldots, \alpha_n\}$, and $t_1, \ldots, t_n$ are pairwise distinct variables, constants, or 1.

The semantics of Presburger arithmetic is defined over the universe $\mathbb{Z}$ of integers in the standard way [4]. Furthermore, we only allow quantifiers that can be handled by Skolemization (only universal/existential quantifiers under an even/odd number of negations).

Gentzen-style sequent calculi. If Γ, Δ are finite sets of formulae and C is a formula, all without free variables, then $\Gamma \vdash \Delta$ is a *sequent*. The sequent is *valid* if the formula $\bigwedge \Gamma \rightarrow \bigvee \Delta$ is valid. A calculus *rule* is a binary relation between a finite set of sequents called the premises, and a sequent called the conclusion. A sequent calculus rule is *sound* if, for all instances

$$\frac{\Gamma_1 \vdash \Delta_1 \quad \cdots \quad \Gamma_n \vdash \Delta_n}{\Gamma \vdash \Delta}$$

whose premises $\Gamma_1 \vdash \Delta_1, \ldots, \Gamma_n \vdash \Delta_n$ are valid, the conclusion $\Gamma \vdash \Delta$ is valid, too. Proof trees are defined to grow upwards. Each node is labeled with a sequent, and each non-leaf node is related to the node(s) directly above it through an instance of a calculus rule. A proof is *closed* if it is finite and all leaves are justified by an instance of a rule without premises.

The *interpolating* sequent calculus for QFPA presented in this paper extends the ground fragment of the sequent calculus in [15].

$$\begin{array}{c l}
* & \\ \hline
\ldots, \mathit{false} \vdash & \textsc{close-left}' \\ \hline
\ldots, 1 \leq 0 \vdash & \textsc{simp}' \\ \hline
\ldots, -x \leq 0, x + 1 \leq 0 \vdash & \textsc{fm-elim}' \\ \hline
\ldots, -x \leq 0, -b + x \leq 0, 3b - 2x + 1 \leq 0 \vdash & \textsc{fm-elim}' \\ \hline
\ldots, -2x \leq 0, -2b + 2x - 1 \leq 0, 3b - 2x + 1 \leq 0 \vdash & \textsc{simp}'^{+} \\ \hline
\ldots, -2x \leq 0, -2b + 2x - 1 \leq 0, c - 3b - 1 \doteq 0, c - 2x \leq 0 \vdash & \textsc{red}' \\ \hline
a - 2x \doteq 0, -a \leq 0, 2b - a \leq 0, -2b + a - 1 \leq 0, c - 3b - 1 \doteq 0, c - a \leq 0 \vdash & \textsc{red}'^{+} \\ \hline
a - 2x \doteq 0 \wedge -a \leq 0 \wedge 2b - a \leq 0 \wedge -2b + a - 1 \leq 0 \wedge c - 3b - 1 \doteq 0 \wedge c - a \leq 0 \vdash & \textsc{and-left}'^{+}
\end{array}$$

Fig. 1. Unsatisfiability proof for the examples of Sect. 3

3 A Motivating Example

Consider the following program with variables ranging over unbounded integers:

```
if (a == 2*x && a >= 0) {
  b = a / 2; c = 3*b + 1; assert (c > a); }
```

We would like to verify the assertion in the program. To this end, the program is translated into the QFPA formula below. Note that `b = a / 2` is converted into a conjunction of two inequalities, and that the assertion is negated:

$$a - 2x \doteq 0 \wedge -a \leq 0 \wedge 2b - a \leq 0 \wedge -2b + a - 1 \leq 0 \wedge c - 3b - 1 \doteq 0 \wedge c - a \leq 0 \quad (1)$$

The unsatisfiability of (1) implies that no run of the program violates the assertion. Fig. 1 shows a refutation of (1) in the Gentzen-style sequent calculus used in this paper (the right-hand side Δ happens to be empty in all sequents, which is not true in general). We add the prime symbol $'$ to the rule names to distinguish them from the interpolating rules introduced later. The proof starts with the conjunction (1) in the bottom sequent of the tree. Repeatedly applying the rule AND-LEFT$'$ (denoted AND-LEFT$'^{+}$) splits the conjunction into a list of arithmetic literals. The equality $a - 2x \doteq 0$ is used to reduce the inequalities $-a \leq 0$, $-2b + a - 1 \leq 0$, and $c - a \leq 0$ by means of substitution (rule RED$'$). Similarly, $c - 3b - 1 \doteq 0$ is used to reduce $c - 2x \leq 0$. The inequalities $-2x \leq 0$ and $-2b + 2x - 1 \leq 0$ are simplified (rule SIMP$'$) by eliminating the coefficient 2; in the latter inequality, this requires rounding. Unsatisfiability of the remaining inequalities follows from two applications of the Fourier-Motzkin rule FM-ELIM$'$, and the proof can be closed.

Interpolants for unsatisfiable formulae like (1) can reveal additional information about the program being investigated, for instance intermediate assertions. Suppose we want to compute an invariant for the program point immediately after `b = a / 2`. Let A denote the part of equation (1) encoding the program up to this point. Currently, the only known interpolation method for QFPA is to quantify out the local variables, i.e., variable x from A:

$$\exists x.\ (a - 2x \doteq 0 \wedge -a \leq 0 \wedge 2b - a \leq 0 \wedge -2b + a - 1 \leq 0)\,,$$

which simplifies via *quantifier elimination* (QE) to $-a \leq 0 \wedge 2b - a \doteq 0$. Existentially quantifying out the local variables from A (or, universally, the local variables from the remaining part of (1)) always returns the strongest (respectively, weakest) interpolant for an unsatisfiable formula. These "extremal" interpolants may be very large, however. Suppose we modify the conditional in the program by adding further conjuncts that are unnecessary for the safety of the program:

$$\texttt{if (a == 2*x \&\& a >= 0 \&\& a >= } n\texttt{*y - } \tfrac{n}{2} \texttt{ \&\& a <= } n\texttt{*y)} \tag{2}$$

where $n \in 2\mathbb{Z}$ is a parameter. The strongest (quantifier-free) interpolant, denoted I_s^n, grows linearly in n and thus exponentially in the size of the program:

$$I_s^n \;\equiv\; -a \leq 0 \wedge 2b - a \doteq 0 \wedge (n \mid a \vee n \mid (a+1) \vee \cdots \vee n \mid (a + \tfrac{n}{2}))\,.$$

A weaker but much more succinct interpolant is the inequality $-3b + a \leq 0$. We demonstrate in this paper that *proof-based* interpolation provides a way of obtaining such succinct interpolants. Proofs can compactly encode the unsatisfiability of a formula and abstract away from irrelevant facts, enabling the extraction of succinct interpolants; this is of particular importance for program verification, where interpolants carrying unnecessary details can delay or prevent the discovery of inductive invariants (e.g., [11]). We therefore propose to *lift* proofs of unsatisfiability to *interpolating proofs*. This way, we avoid many disadvantages of QE-based interpolation, namely (i) its high complexity, (ii) its inflexibility in always returning a strongest or weakest interpolant, and (iii) the need to restart from scratch in order to consider a new partitioning of the unsatisfiable formula into A and B (in contrast, a proof-based method can extract many interpolants from a single proof).

4 An Interpolating Sequent Calculus for QFPA

In order to extract interpolants from proofs of unsatisfiable conjunctions $A \wedge B$, we introduce *interpolating sequents* as an extension of the Gentzen-style sequents defined in Sect. 2. Formulae in interpolating sequents are labeled either with the letter L to indicate that they are derived purely from A, the letter R for formulae derived purely from B, or with *partial interpolants* (PIs) that record the A-contribution to a formula obtained jointly from A and B. Similarly as in [4], the labels L/R will be used to handle analytic rules that operate only on subformulae of the input formulae, while rewriting rules for arithmetic may mix parts of A and B and therefore require partial interpolants (as in [10]).

More formally, if ϕ is a formula and t, t^A are terms, all without free variables, then $\lfloor \phi \rfloor_L$ and $\lfloor \phi \rfloor_R$ are L/R-labeled formulae and $t \doteq 0\,[t^A \doteq 0]$, $t \doteq 0\,[t^A \not\doteq 0]$, and $t \leq 0\,[t^A \leq 0]$ are formulae labeled with the partial interpolants $t^A \doteq 0$, $t^A \not\doteq 0$, and $t^A \leq 0$, respectively. Furthermore, if Γ, Δ are sets of labeled formulae and I is an unlabeled formula such that (i) none of the formulae contains free variables, (ii) Γ only contains formulae $\lfloor \phi \rfloor_L$, $\lfloor \phi \rfloor_R$, $t \doteq 0\,[t^A \doteq 0]$, or $t \leq 0\,[t^A \leq 0]$, and (iii) Δ only contains formulae $\lfloor \phi \rfloor_L$, $\lfloor \phi \rfloor_R$, $t \doteq 0\,[t^A \doteq 0]$, or $t \doteq 0\,[t^A \not\doteq 0]$, then $\Gamma \vdash \Delta \blacktriangleright I$ is an *interpolating sequent*.

$$
\begin{array}{cl}
* & \\
\hline
\ldots, 2 \leq 0\,[-6b + 2a \leq 0] \;\vdash\; \blacktriangleright I_1 & \text{CLOSE-INEQ} \\
\hline
\ldots, -2x \leq 0\,[-2x \leq 0], 2x + 2 \leq 0\,[-6b + 2a + 2x \leq 0] \;\vdash\; \blacktriangleright I_1 & \text{FM-ELIM} \\
\hline
\mathcal{A} \quad \mathcal{B} \quad \ldots, -2b + 2x \leq 0\,[-2b + 2x - 1 \leq 0], 3b - 2x + 1 \leq 0\,[a - 2x \leq 0] \;\vdash\; \blacktriangleright I_1 & \text{FM-ELIM} \\
\hline
\ldots, -2b + 2x - 1 \leq 0\,[-2b + 2x - 1 \leq 0], 3b - 2x + 1 \leq 0\,[a - 2x \leq 0] \;\vdash\; \blacktriangleright I_2 & \text{STRENGTHEN} \\
\hline
\begin{array}{c} \ldots, -2x \leq 0\,[-2x \leq 0], -2b + 2x - 1 \leq 0\,[-2b + 2x - 1 \leq 0], \\ c - 3b - 1 \doteq 0\,[0 \doteq 0], c - 2x \leq 0\,[a - 2x \leq 0] \end{array} \;\vdash\; \blacktriangleright I_2 & \text{RED-LEFT} \\
\hline
\begin{array}{c} \ldots, a - 2x \doteq 0\,[a - 2x \doteq 0], -2b + a - 1 \leq 0\,[-2b + a - 1 \leq 0], \\ -a \leq 0\,[-a \leq 0], c - 3b - 1 \doteq 0\,[0 \doteq 0], c - a \leq 0\,[0 \leq 0] \end{array} \;\vdash\; \blacktriangleright I_2 & \text{RED-LEFT}^+ \\
\hline
\begin{array}{c} \lfloor a - 2x \doteq 0 \rfloor_L, \lfloor -a \leq 0 \rfloor_L, \ldots, \lfloor -2b + a - 1 \leq 0 \rfloor_L, \\ \lfloor c - 3b - 1 \doteq 0 \rfloor_R, \lfloor c - a \leq 0 \rfloor_R \end{array} \;\vdash\; \blacktriangleright I_2 & \text{IPI}^+ \\
\hline
\begin{array}{c} \lfloor a - 2x \doteq 0 \land -a \leq 0 \land 2b - a \leq 0 \land -2b + a - 1 \leq 0 \rfloor_L, \\ \lfloor c - 3b - 1 \doteq 0 \land c - a \leq 0 \rfloor_R \end{array} \;\vdash\; \blacktriangleright I_2 & \text{AND-LEFT}^+
\end{array}
$$

Fig. 2. The interpolating version of Fig. 1. The initial interpolant generated by CLOSE-INEQ is $I_1 = (-6b + 2a \leq 0) \equiv (-3b + a \leq 0)$, which is by STRENGTHEN combined with the interpolants *false* and ϕ from the subproofs $\mathcal{A}$ and $\mathcal{B}$ to form the final interpolant $I_2 = (I_1 \lor (\mathit{false} \land \phi)) \equiv I_1$.

The semantics of interpolating sequents is defined with the help of projections $\Gamma_L =_{\text{def}} \{\phi \mid \lfloor\phi\rfloor_L \in \Gamma\}$ and $\Gamma_R =_{\text{def}} \{\phi \mid \lfloor\phi\rfloor_R \in \Gamma\}$ that extract the L/R-parts of a set Γ of labeled formulae. A sequent $\Gamma \vdash \Delta \blacktriangleright I$ is *valid* if (i) the (Gentzen-style) sequent $\Gamma_L \vdash I, \Delta_L$ is valid, (ii) the sequent $\Gamma_R, I \vdash \Delta_R$ is valid, and (iii) the constants in I occur in both $\Gamma_L \cup \Delta_L$ and $\Gamma_R \cup \Delta_R$. Note that formulae annotated with PIs are irrelevant for deciding whether an interpolating sequent is valid; this only depends on L/R-formulae. The semantics of PIs is made precise in Sect. 4.3; intuitively, a labeled formula $\phi\,[\phi^A]$ in an interpolation problem $A \land B$ expresses the implications $A \Rightarrow \phi^A$ and $B \land \phi^A \Rightarrow \phi$.

As special cases, $\lfloor A\rfloor_L \vdash \lfloor C\rfloor_R \blacktriangleright I$ reduces to I being an interpolant of the implication $A \Rightarrow C$, while $\lfloor A\rfloor_L, \lfloor B\rfloor_R \vdash \;\blacktriangleright I$ captures the concept of interpolants I for conjunctions $A \land B$ common in formal verification.

Example. We illustrate the concept of interpolating sequents with the proof in Fig. 2, which is the interpolating version of the proof in Fig. 1 and will serve as a running example in the whole section. For sake of brevity, we omit the subproofs $\mathcal{A}$ and $\mathcal{B}$. Due to the soundness of the applied calculus (stated in Sect. 4.3), the root sequent of the proof is valid, which implies that $I_2 \equiv (-3b + a \leq 0)$ is an interpolant for the unsatisfiable conjunction (1). Note that I_2 is the inequality discussed in Sect. 3 as a succinct interpolant and intermediate program assertion.

In the remainder of Sect. 4, we explain the rules of our interpolating calculus given in Fig. 3, 4. As usual in sequent calculi, the rules are applied in the upward direction, starting from a sequent $\Gamma \vdash \Delta \blacktriangleright ?$ with unknown interpolant that is to be proven (the proof root), and applying rules to successively decompose and simplify the sequent until a closure rule becomes applicable. The unknown interpolants of sequents have to be left open while building a proof and can only be filled in once all proof branches are closed.

$$\frac{\Gamma, t \circ 0\,[t \circ 0], \lfloor t \circ 0 \rfloor_L \vdash \Delta \blacktriangleright I}{\Gamma, \lfloor t \circ 0 \rfloor_L \vdash \Delta \blacktriangleright I}\ \text{IPI-LEFT} \qquad \frac{\Gamma \vdash t \doteq 0\,[t \doteq 0], \lfloor t \doteq 0 \rfloor_L, \Delta \blacktriangleright I}{\Gamma \vdash \lfloor t \doteq 0 \rfloor_L, \Delta \blacktriangleright I}\ \text{IPI-RIGHT}$$

$$\frac{\Gamma, t \circ 0\,[0 \circ 0], \lfloor t \circ 0 \rfloor_R \vdash \Delta \blacktriangleright I}{\Gamma, \lfloor t \circ 0 \rfloor_R \vdash \Delta \blacktriangleright I}\ \text{IPI-LEFT} \qquad \frac{\Gamma \vdash t \doteq 0\,[0 \neq 0], \lfloor t \doteq 0 \rfloor_R, \Delta \blacktriangleright I}{\Gamma \vdash \lfloor t \doteq 0 \rfloor_R, \Delta \blacktriangleright I}\ \text{IPI-RIGHT}$$

$$\frac{*}{\Gamma, t \doteq 0\,[t^A \doteq 0] \vdash \Delta \blacktriangleright \exists_{LA}\, t^A \doteq 0}\ \text{CLOSE-EQ-LEFT} \qquad (t \doteq 0 \text{ is unsatisfiable})$$

$$\frac{*}{\Gamma, \alpha \leq 0\,[t^A \leq 0] \vdash \Delta \blacktriangleright \exists_{LA}\, t^A \leq 0}\ \text{CLOSE-INEQ} \qquad (\alpha > 0)$$

$$\frac{*}{\Gamma \vdash 0 \doteq 0\,[t^A \doteq 0], \Delta \blacktriangleright \exists_{LA}\, t^A \neq 0}\ \text{CLOSE-EQ-RIGHT}$$

$$\frac{*}{\Gamma \vdash 0 \doteq 0\,[t^A \neq 0], \Delta \blacktriangleright \exists_{LA}\, t^A \doteq 0}\ \text{CLOSE-NEQ-RIGHT}$$

Fig. 3. Initialization and closure rules. In the rules IPI-LEFT-L/R, $\circ \in \{\doteq, \leq\}$ denotes a relation symbol. In the rules CLOSE-*, $\exists_{LA}$ denotes existential quantification $\exists c_1, \ldots, c_n.$, where $c_1, \ldots, c_n$ are the constants that occur in Γ_L, Δ_L but not in Γ_R, Δ_R. An equality $t^A \doteq 0$ is unsatisfiable if and only if it is of the form $\alpha_1 d_1 + \cdots + \alpha_n d_n + \alpha_0 \doteq 0$ and $\gcd(\alpha_1, \ldots, \alpha_n) \nmid \alpha_0$ (with the convention $\gcd() = 0$).

4.1 Propositional, Initialization, and Closure Rules

To construct a proof for an interpolation problem $A \wedge B$, we start with a sequent $\lfloor A \rfloor_L, \lfloor B \rfloor_R \vdash \ \blacktriangleright ?$ that only contains L/R-labeled formulae and apply propositional and Skolemization rules to decompose A and B (the applications of rule AND-LEFT in Fig. 2). Because our propositional rules closely follow standard interpolating calculi (see [9,4]), we only show two of these rules, namely the top-most two in Fig. 4. When splitting over L-disjunctions in the antecedent (OR-LEFT-L), it is necessary to form the disjunction of the interpolants derived in the subproofs. Analogously, R-disjunctions yield conjunctive interpolants. All propositional rules propagate the L/R-label of formulae to their subformulae, unchanged. For brevity, we have omitted rules to move inequalities from the succedent to the antecedent.

Once the decomposition of formulae results in arithmetic literals, the *initialization* rules in the upper part of Fig. 3 are used to turn L/R-formulae into formulae with PIs, to prepare them for later rewriting (the applications IPI in Fig. 2). Generally, PIs for L-literals are chosen to be the literals themselves, while empty PIs are introduced for R-literals: the intuition is that L-formulae are fully contributed by A, while R-formulae do not contain any A-contribution at all.

We observe that the IPI rules do not remove the L/R-formula to which they are applied (the formula occurs both in the conclusion and in the premise). The reason is that L/R-formulae in sequents, besides their logical meaning, track the vocabulary of symbols occurring in the input formulae A, B; the vocabulary is

$$\dfrac{\begin{array}{c}\Gamma, \lfloor\phi\rfloor_L \vdash \Delta \blacktriangleright I \\ \Gamma, \lfloor\psi\rfloor_L \vdash \Delta \blacktriangleright J\end{array}}{\Gamma, \lfloor\phi \vee \psi\rfloor_L \vdash \Delta \blacktriangleright I \vee J}\ \textsc{or-left-l} \qquad \dfrac{\Gamma, \lfloor\phi\rfloor_D, \lfloor\psi\rfloor_D \vdash \Delta \blacktriangleright I}{\Gamma, \lfloor\phi \wedge \psi\rfloor_D \vdash \Delta \blacktriangleright I}\ \textsc{and-left}$$

$$\dfrac{\Gamma, t \doteq 0\,[t^A \doteq 0], s + \alpha \cdot t \circ 0\,[s^A + \alpha \cdot t^A \circ 0] \vdash \Delta \blacktriangleright I}{\Gamma, t \doteq 0\,[t^A \doteq 0], s \circ 0\,[s^A \circ 0] \vdash \Delta \blacktriangleright I}\ \textsc{red-left}$$

$$\dfrac{\Gamma, t \doteq 0\,[t^A \doteq 0] \vdash s + \alpha \cdot t \doteq 0\,[s^A + \alpha \cdot t^A \circ 0], \Delta \blacktriangleright I}{\Gamma, t \doteq 0\,[t^A \doteq 0] \vdash s \doteq 0\,[s^A \circ 0], \Delta \blacktriangleright I}\ \textsc{red-right}$$

$$\dfrac{\Gamma, \lfloor u - c \doteq 0\rfloor_L \vdash \Delta \blacktriangleright I}{\Gamma \vdash \Delta \blacktriangleright I}\ \textsc{col-red-l} \qquad \dfrac{\Gamma, \alpha \cdot t \circ 0\,[\alpha \cdot t^A \circ 0] \vdash \Delta \blacktriangleright I}{\Gamma, t \circ 0\,[t^A \circ 0] \vdash \Delta \blacktriangleright I}\ \textsc{mul-left}$$

$$\dfrac{\Gamma, \lfloor u - c \doteq 0\rfloor_R \vdash \Delta \blacktriangleright I}{\Gamma \vdash \Delta \blacktriangleright I}\ \textsc{col-red-r} \qquad \dfrac{\Gamma \vdash \alpha \cdot t \doteq 0\,[\alpha \cdot t^A \circ 0], \Delta \blacktriangleright I}{\Gamma \vdash t \doteq 0\,[t^A \circ 0], \Delta \blacktriangleright I}\ \textsc{mul-right}$$

$$\dfrac{\Gamma, \lfloor \exists x.\, \alpha x + t \doteq 0\rfloor_D \vdash \Delta \blacktriangleright I}{\Gamma, \lfloor \alpha \mid t\rfloor_D \vdash \Delta \blacktriangleright I}\ \textsc{div-left}$$

$$\dfrac{\Gamma, \lfloor (\alpha \mid t + 1) \vee \cdots \vee (\alpha \mid t + \alpha - 1)\rfloor_D \vdash \Delta \blacktriangleright I}{\Gamma \vdash \lfloor \alpha \mid t\rfloor_D, \Delta \blacktriangleright I}\ \textsc{div-right}$$

$$\dfrac{\Gamma, s \leq 0\,[s^A \leq 0], t \leq 0\,[t^A \leq 0], \alpha s + \beta t \leq 0\,[\alpha s^A + \beta t^A \leq 0] \vdash \Delta \blacktriangleright I}{\Gamma, s \leq 0\,[s^A \leq 0], t \leq 0\,[t^A \leq 0] \vdash \Delta \blacktriangleright I}\ \textsc{fm-elim}$$

$$\dfrac{\begin{array}{c}\Gamma, t \doteq 0\,[t^A \doteq 0] \vdash \Delta \blacktriangleright E \\ \Gamma, t + 1 \leq 0\,[t^A \leq 0] \vdash \Delta \blacktriangleright I^0 \\ \Gamma, t + 1 \leq 0\,[t^A + 1 \leq 0] \vdash \Delta \blacktriangleright I^1\end{array}}{\Gamma, t \leq 0\,[t^A \leq 0] \vdash \Delta \blacktriangleright I^1 \vee (E \wedge I^0)}\ \textsc{strengthen}$$

$$\dfrac{\begin{array}{c}\Gamma, t + 1 \leq 0\,[t^A + 1 \leq 0] \vdash \Delta \blacktriangleright I \\ \Gamma, -t + 1 \leq 0\,[-t^A + 1 \leq 0] \vdash \Delta \blacktriangleright J\end{array}}{\Gamma \vdash t \doteq 0\,[t^A \doteq 0], \Delta \blacktriangleright I \vee J}\ \textsc{split-eq}$$

$$\dfrac{\begin{array}{c}\Gamma, t + 1 \leq 0\,[t^A \leq 0] \vdash \Delta \blacktriangleright I \\ \Gamma, -t + 1 \leq 0\,[-t^A \leq 0] \vdash \Delta \blacktriangleright J\end{array}}{\Gamma \vdash t \doteq 0\,[t^A \not\doteq 0], \Delta \blacktriangleright I \wedge J}\ \textsc{split-neq}$$

Fig. 4. Rules for propositional connectives, equalities, divisibility, and inequalities. In AND-LEFT, we assume $D \in \{L, R\}$. In RED-LEFT and MUL-LEFT, $\circ \in \{\doteq, \leq\}$, while $\circ \in \{\doteq, \not\doteq\}$ in RED-RIGHT and MUL-RIGHT. In COL-RED-L and COL-RED-R, c is a constant that does not occur in the conclusion or in u. The term u in COL-RED-L must only contain constants from $\Gamma_L \cup \Delta_L$, while u in COL-RED-R must only contain constants from $\Gamma_R \cup \Delta_R$. In MUL-LEFT and MUL-RIGHT, $\alpha > 0$ is a positive literal. In DIV-LEFT and DIV-RIGHT, $D \in \{L, R\}$, x is an arbitrary variable, and $\alpha > 0$. In FM-ELIM, $\alpha > 0$ and $\beta > 0$ are positive integers.

used in condition (iii) of the definition of valid interpolating sequents, but also in the closure rules discussed next. For completeness, it is never necessary to apply IPI rules twice on a proof branch to the same L/R-formula.

Finally, once rewriting (discussed in Sect. 4.2) has produced an unsatisfiable literal in an antecedent (or a valid literal in a succedent), a closure rule can be used to close the proof branch and to derive an interpolant from the PI of the unsatisfiable literal (the application CLOSE-INEQ in Fig. 2). Closure rules are given in the lower part of Fig. 3. Because PIs can still contain local symbols that occur only in $\Gamma_L \cup \Delta_L$ (and are not allowed in interpolants), it may be necessary to introduce existential quantifiers at this point. We note, however, that quantifiers in quantified literals can be eliminated in polynomial time; e.g., $\exists c_1, \ldots, c_n.\ \alpha_1 c_1 + \cdots + \alpha_n c_n + t \doteq 0$ is equivalent to the divisibility judgement $\gcd(\alpha_1, \ldots, \alpha_n) \mid t$.

4.2 Rewriting Rules for Equality, Inequality and Divisibility

The arithmetic rewriting rules form a calculus to solve systems of equalities by means of Gaussian elimination and Euclid's algorithm (the middle part of Fig. 4), as well as a calculus for systems of inequalities based on Fourier-Motzkin elimination and cutting planes (the lower part of Fig. 4). Decision procedures for QFPA in terms of the corresponding non-interpolating rules have been introduced in [14,15] and directly carry over to the interpolating case. We therefore focus on the differences between the normal and the interpolating rules.

The rules RED-LEFT/RIGHT rewrite (in)equalities with equalities in the antecedent; in both cases, PIs are simply propagated along with the literals (RED-LEFT is applied repeatedly in Fig. 2). The RED rules alone do not form a complete calculus for integer equalities and have to be complemented with COL-RED-L/R to introduce fresh constants defined in terms of existing constants (the rules resemble *col*umn *red*uctions when encoding systems of equalities as matrices). In combination, RED and COL-RED are able to simulate the equality elimination procedure in [13], as well as standard procedures to transform sets of equalities (or matrices) to Hermite and Smith normalform [6,5]. Because COL-RED-L/R only introduce local L/R-constants, it is guaranteed that the new constants do not occur in interpolants.

In contrast to [14,15], we do not introduce a simplification rule SIMP$'$ for literals, as full simplification is not always possible in the presence of PIs. For instance, the equality $2x \doteq 0\,[a \doteq 0]$ cannot be simplified to the form $x \doteq 0\,[t^A \doteq 0]$ (as it would happen in [14,15]) because the factor 2 does not occur in the PI. This raises a potential problem, as terms αx cannot be rewritten to 0 with the help of $2x \doteq 0$ if α is odd. As a solution, we introduce the rules MUL-LEFT/RIGHT to multiply terms with positive integers prior to rewriting.

Similar to rewriting with equalities, inequalities can be added to each other with the help of the rule FM-ELIM realizing Fourier-Motzkin variable elimination. The STRENGTHEN rule is introduced to achieve completeness over the integers (Fig. 2 shows applications of FM-ELIM and STRENGTHEN). Compared to the calculi in [14,15], the use of STRENGTHEN in our interpolating calculus is

Table 1. Sequents with partial interpolants and correctness conditions (i) and (ii)

Partial interpolant annotation	Sequent (i)	Sequent (ii)
$\Gamma, t \doteq 0[t^A \doteq 0] \vdash \Delta$	$\Gamma_L \vdash t^A \doteq 0, \Delta_L$	$\Gamma_R \vdash t - t^A \doteq 0, \Delta_R$
$\Gamma, t \leq 0[t^A \leq 0] \vdash \Delta$	$\Gamma_L \vdash t^A \leq 0, \Delta_L$	$\Gamma_R \vdash t - t^A \leq 0, \Delta_R$
$\Gamma \vdash t \doteq 0[t^A \doteq 0], \Delta$	$\Gamma_L, t^A \doteq 0 \vdash \Delta_L$	$\Gamma_R \vdash t - t^A \doteq 0, \Delta_R$
$\Gamma \vdash t \doteq 0[t^A \not\doteq 0], \Delta$	$\Gamma_L \vdash t^A \doteq 0, \Delta_L$	$\Gamma_R, t - t^A \doteq 0 \vdash \Delta_R$

threefold: (i) STRENGTHEN can simulate the OMEGA-ELIM rule in [15], (ii) as shown in Fig. 2, repeated application of STRENGTHEN can be used to round inequalities $\alpha t + \beta \leq 0$ to $\alpha t + \alpha\lceil\frac{\beta}{\alpha}\rceil \leq 0$ (which is done by SIMP′ in [14,15]), and (iii) STRENGTHEN can simulate the law of anti-symmetry that is implemented by the rule ANTI-SYMM′ in [14,15]. As STRENGTHEN is the most central rule in our calculus, we provide a detailed discussion in Sect. 5.

4.3 Properties of the Calculus

Soundness. Our interpolating calculus generates correct interpolants: whenever a sequent $\lfloor A \rfloor_L \vdash \lfloor C \rfloor_R \blacktriangleright I$ is derived, the implications $A \Rightarrow I \Rightarrow C$ are valid, and all constants in I occur in both A and C. More generally:

Lemma 1 (Soundness). *If an interpolating sequent $\Gamma \vdash \Delta \blacktriangleright I$ without any PIs is provable in the calculus, then it is valid. This implies, in particular, that the sequent $\Gamma_L, \Gamma_R \vdash \Delta_L, \Delta_R$ is valid.*

To prove this lemma, we first need to define the semantics of PIs (although the sequent $\Gamma \vdash \Delta \blacktriangleright I$ in the lemma does not contain any PIs, they are likely to be introduced in the course of a proof). We say that a PI is *correct* if the sequents (i) and (ii) given in Table 1 are valid, t^A only contains constants that occur in $\Gamma_L \cup \Delta_L$, and $t - t^A$ only contains constants that occur in $\Gamma_R \cup \Delta_R$. Soundness is then proven in two steps: (i) We show that all PIs in a closed proof are correct by induction on the distance of a sequent from the root of the proof: assuming that all PIs in the conclusion of a rule application are correct, we prove that the PIs in the rule premises are correct. (ii) We show the validity of all sequents in a closed proof by induction on the size of sub-proofs: assuming that all premises of a rule are valid, we prove that the conclusion is valid, too.

As a technical difficulty, we need to annotate some rules by introducing further auxiliary formulae in the premises to ensure (i) holds. These annotations are only required for the soundness proof; soundness of the rules with auxiliary formulae directly implies soundness of the original rules.

Completeness. Vice versa, whenever an implication $A \Rightarrow C$ holds, our calculus is able to derive an interpolant. We have to ban quantifiers that cannot be handled by Skolemization.

Lemma 2 (Completeness). *Suppose Γ, Δ are sets of labeled formulae $\lfloor \phi \rfloor_L$ and $\lfloor \phi \rfloor_R$ such that all occurrences of existential quantifiers in Γ / Δ are under* *an even/odd number of negations, and all occurrences of universal quantifiers in Γ/Δ are under an odd/even number of negations. If $\Gamma_L, \Gamma_R \vdash \Delta_L, \Delta_R$ is valid, then there is a formula I such that $\Gamma \vdash \Delta \blacktriangleright I$ is provable.*

The lemma follows from the completeness of the calculi in [14,15] by means of proof lifting: given that $\Gamma_L, \Gamma_R \vdash \Delta_L, \Delta_R$ is valid, there is a proof of this fact in the non-interpolating calculus. This proof can be lifted by replacing each rule application with an application of the corresponding interpolating rule.

5 Strengthening and Mixed Cuts

Reasoning in linear integer arithmetic generally requires some kind of *cut-rule* to deal with the phenomenon of formulae that are satisfiable over the rationals, but unsatisfiable over integers. The non-interpolating calculus in [14] provides two rules for this: the SIMP′ rule to round inequalities $\alpha t + \beta \leq 0$ to $\alpha t + \alpha\lceil\frac{\beta}{\alpha}\rceil \leq 0$ (which resembles Gomory cuts [17]), and the general STRENGTHEN′ rule:

$$\frac{\Gamma, t \doteq 0 \vdash \Delta \quad \Gamma, t+1 \leq 0 \vdash \Delta}{\Gamma, t \leq 0 \vdash \Delta}\ \textsc{strengthen}'$$

Because STRENGTHEN′ subsumes rounding via the rule SIMP′, we can ignore the latter rule for the time being and concentrate on STRENGTHEN′.

In order to lift STRENGTHEN′ to the interpolating calculus, we can first observe that two special cases are easy to handle:

$$\frac{\Gamma, t \doteq 0\,[t \doteq 0] \vdash \Delta \blacktriangleright I \quad \Gamma, t+1 \leq 0\,[t+1 \leq 0] \vdash \Delta \blacktriangleright J}{\Gamma, t \leq 0\,[t \leq 0] \vdash \Delta \blacktriangleright I \vee J}\ \textsc{strengthen-l}$$

$$\frac{\Gamma, t \doteq 0\,[0 \doteq 0] \vdash \Delta \blacktriangleright I \quad \Gamma, t+1 \leq 0\,[0 \leq 0] \vdash \Delta \blacktriangleright J}{\Gamma, t \leq 0\,[0 \leq 0] \vdash \Delta \blacktriangleright I \wedge J}\ \textsc{strengthen-r}$$

These cases are called *pure cuts* in [8], because the PIs tell that the inequality $t \leq 0$ has been derived only from L- or only from R-formulae, respectively. Strengthening inequalities of this kind corresponds to splitting a disjunction labeled with L or R.

The general case is known as *mixed cut* [8] and encompasses an application of STRENGTHEN to a formula $t \leq 0\,[t^A \leq 0]$ with $t^A \notin \{0, t\}$; the rule for this general case is given in Fig. 4 and features *three* premises, one more than the non-interpolating rule STRENGTHEN′. To understand the shape of STRENGTHEN, note that we can represent $t \leq 0$ as the sum of the inequalities $t^A \leq 0$ and $t - t^A \leq 0$, the first of which is derived from L-formulae, and the second from R-formulae. The effect of STRENGTHEN can then be simulated by applying STRENGTHEN-L to $t^A \leq 0\,[t^A \leq 0]$, and afterward STRENGTHEN-R to $t - t^A \leq 0\,[0 \leq 0]$; the combined application of the two rules explains the interpolant $I^1 \vee (E \wedge I^0)$ resulting from STRENGTHEN.

Complexity. Non-interpolating refutations of unsatisfiable conjunctions of literals have exponential size in the worst case [17]. Similarly, it can be shown that any valid sequent (without quantifiers or propositional connectives) has interpolants of worst-case exponential size that can be derived using a proof of worst-case exponential size (using the rules STRENGTHEN-L/R from above).

In general, however, lifting a non-interpolating to an interpolating proof can increase the size of the proof exponentially, due to two reasons: (i) the fact that STRENGTHEN in Fig. 4 has three premises, while the non-interpolating rule STRENGTHEN′ has only two, which can make it necessary to repeatedly duplicate subproofs during lifting (this is partly addressed in Sect. 5.1), and (ii) because the rule SIMP′ (which has to be simulated by STRENGTHEN in the interpolating calculus) often allows very succinct proofs. As a result, there are unsatisfiable conjunctions $A \wedge B$ with non-interpolating proofs of linear size, although all interpolants have exponential size.

5.1 Successive Strengthening

It is quite common that STRENGTHEN is applied repeatedly to a sequence $t \leq 0$, $t+1 \leq 0$, $t+2 \leq 0$, $\ldots$ of inequalities, for instance to simulate rounding of an inequality or the Omega rule [13]. Because each application of STRENGTHEN generates two new inequalities, $2^k - 1$ applications are necessary in order to strengthen an inequality $t \leq 0$ to $t + k \leq 0$, and the resulting interpolant will be of exponential size as well. To tackle this growth, we present an optimized rule that captures k-fold strengthening and requires only a quadratic number of premises. The optimized rule k-STRENGTHEN exploits the fact that many of the goals created by repeated application of STRENGTHEN are redundant:

$$\frac{\begin{array}{c}\left\{\Gamma, t+i \doteq 0\,[t_A + j \doteq 0] \;\vdash\; \Delta \;\blacktriangleright\; E_i^j\right\}_{0\leq j\leq i<k} \\ \left\{\Gamma, t+k \leq 0\,[t_A + j \leq 0] \;\vdash\; \Delta \;\blacktriangleright\; I^j\right\}_{0\leq j\leq k}\end{array}}{\Gamma, t \leq 0\,[t_A \leq 0] \;\vdash\; \Delta \;\blacktriangleright\; K}\;k\text{-STRENGTHEN}$$

where the resulting interpolant K is defined by:

$$K = \bigvee_{0\leq j\leq k} \Big(I^j \wedge \bigwedge_{j\leq i<k} E_i^j \Big) \tag{3}$$

The size of K grows quadratically, rather than exponentially, in k. Thus, whenever the STRENGTHEN rule is to be applied k times in succession, it is possible and more efficient to use the k-STRENGTHEN rule instead.

The number of premises of k-STRENGTHEN (but not the size of the resulting interpolant) can be reduced further to a linear number: any two premises generating E_i^j and E_i^l differ only in the partial interpolant of $t + i \leq 0$, not in any other formula. We can exploit this by treating the family $(E_i^j)_{0\leq j\leq i}$ as a *single* premise that is parameterized in the free variable j. This way, a single subproof can generate a parameterized interpolant $E_i(j)$. The parameter j can be instantiated to the values $0 \leq j \leq i$ when constructing K. Parametrized interpolants $I(j)$ can be derived similarly.

Interpolation of rounding operations. An additional optimization is possible when the rule k-STRENGTHEN is used to round an inequality $\alpha t + \beta \leq 0$ to $\alpha t + \alpha\lceil\frac{\beta}{\alpha}\rceil \leq 0$. Rounding corresponds to k-STRENGTHEN with $k = \alpha\lceil\frac{\beta}{\alpha}\rceil - \beta$:

$$\dfrac{\begin{array}{l}\left\{\Gamma, \alpha t + \beta + i \doteq 0\,[t_A + j \doteq 0] \;\vdash\; \Delta \;\blacktriangleright E_i^j\right\}_{0\leq j\leq i<k} \\ \left\{\Gamma, \alpha t + \alpha\lceil\frac{\beta}{\alpha}\rceil \leq 0\,[t_A + j \leq 0] \;\vdash\; \Delta \;\blacktriangleright I^j\right\}_{0\leq j\leq k}\end{array}}{\Gamma, \alpha t + \beta \leq 0\,[t_A \leq 0] \;\vdash\; \Delta \;\blacktriangleright K}\;k\text{-}\textsc{strengthen}$$

We can observe that $\alpha t + \beta + i \doteq 0$ is unsatisfiable for $0 \leq i < \alpha\lceil\frac{\beta}{\alpha}\rceil - \beta$, so that the equality-premises can be closed immediately via CLOSE-EQ-LEFT. Consequently, the interpolants $E_i^j = E^j = (\exists_{LA}\, t^A + j \doteq 0)$ do not depend on i, and the overall interpolant can be simplified to $K = I^k \vee \bigvee_{0\leq j<k}(I^j \wedge E^j)$.

Example. We use k-STRENGTHEN to compute an interpolant for the conjunction $A \wedge B$ with $A = -y + 5x - 1 \leq 0 \wedge y - 5x \leq 0$ and $B = 5z - y + 1 \leq 0 \wedge -5z + y - 2 \leq 0$. Note that $A \wedge B$ is satisfiable over rationals, but unsatisfiable over the integers. An interpolating proof of unsatisfiability is as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{c}*\\ \vdots \\ \{\cdots \blacktriangleright E_i^j\}\end{array} \qquad \dfrac{\dfrac{*}{\ldots, 1 \leq 0\,[j-1 \leq 0] \;\vdash\; \blacktriangleright j-1 \leq 0}\;\textsc{close-ineq}}{\begin{array}{c}\ldots, -5z+5x \leq 0\,[-y+5x-1+j \leq 0], \\ 5z-5x+1 \leq 0\,[y-5x \leq 0]\end{array} \;\vdash\; \blacktriangleright j-1 \leq 0}\;\textsc{fm-elim}}{\ldots, -5z+5x-3 \leq 0\,[-y+5x-1 \leq 0], 5z-5x+1 \leq 0\,[y-5x \leq 0] \;\vdash\; \blacktriangleright K}\;3\text{-}\textsc{strengthen}}{\begin{array}{c}\ldots, y-5x \leq 0\,[y-5x \leq 0], 5z-y+1 \leq 0\,[0 \leq 0], \\ -5z+5x-3 \leq 0\,[-y+5x-1 \leq 0]\end{array} \;\vdash\; \blacktriangleright K}\;\textsc{fm-elim}}{\begin{array}{c}-y+5x-1 \leq 0\,[-y+5x-1 \leq 0], y-5x \leq 0\,[y-5x \leq 0], \\ 5z-y+1 \leq 0\,[0 \leq 0], -5z+y-2 \leq 0\,[0 \leq 0]\end{array} \;\vdash\; \blacktriangleright K}\;\textsc{fm-elim}}{\dfrac{\lfloor -y+5x-1 \leq 0\rfloor_L, \lfloor y-5x \leq 0\rfloor_L, \lfloor 5z-y+1 \leq 0\rfloor_R, \lfloor -5z+y-2 \leq 0\rfloor_R \;\vdash\; \blacktriangleright K}{\lfloor -y+5x-1 \leq 0 \wedge y-5x \leq 0\rfloor_L, \lfloor 5z-y+1 \leq 0 \wedge -5z+y-2 \leq 0\rfloor_R \;\vdash\; \blacktriangleright K}\;\textsc{and-left}^+}\;\textsc{ipi}^+$$

Most importantly, the rule 3-STRENGTHEN is used to round $-5z + 5x - 3 \leq 0$ to $-5z + 5x \leq 0$, from which a contradiction can be derived via FM-ELIM. In the premises of 3-STRENGTHEN, the inequality interpolants $I^j = (j - 1 \leq 0)$ and the equality interpolants $E^j = (\exists x.\ -y + 5x - 1 + j \doteq 0) \equiv (5 \mid (y + 1 - j))$ are derived as discussed above. The overall interpolant is:

$$K = \underbrace{3 - 1 \leq 0}_{I^k} \vee \bigvee_{0\leq j<3} (\underbrace{j - 1 \leq 0}_{I^j} \wedge \underbrace{5 \mid (y + 1 - j)}_{E^j}) \quad\equiv\quad 5 \mid (y+1) \vee 5 \mid y$$

6 Experimental Evaluation

We implemented[1] the proposed interpolating calculus on top of the PRINCESS theorem prover [15], including all optimizations described in Sect. 5. To this end, we extended PRINCESS to generate proofs. The interpolation procedure then processes the proof and generates an interpolant using the rules presented in this paper. The benchmarks for our experiments are derived from the SMT-LIB category QF-LIA. We evaluate them on an Intel Pentium Xeon with 3 GHz and

[1] Implementation and benchmarks: www.philipp.ruemmer.org/iprincess.shtml

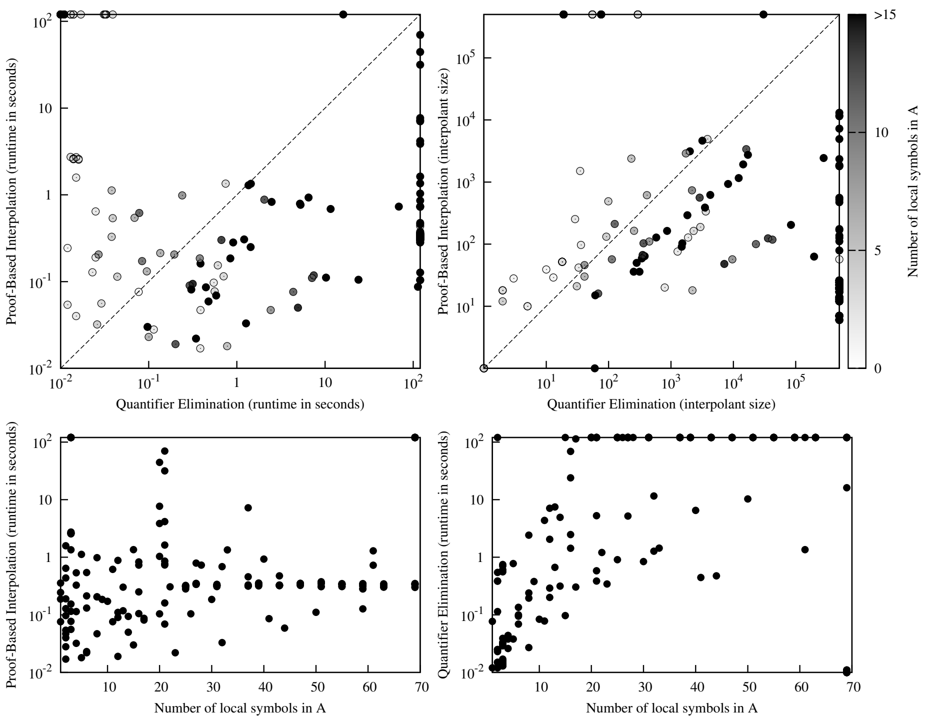


Fig. 5. Benchmarks comparing interpolant extraction with quantifier elimination

4 MB cache, running Linux. Because SMT-LIB benchmarks are usually conjunctions at the outermost level, we partitioned them into $A \wedge B$ by choosing the first $\frac{k}{10} \cdot n$ of the benchmark conjuncts as A, the rest as B (where n is the total number of conjuncts, and $k \in \{1, \ldots, 9\}$). Partitionings where A did not contain any local symbols (constants or propositional variables) were ignored.

Since (to the best of our knowledge) no other interpolation procedure for QFPA was available, we compared the performance of the Princess interpolation procedure with interpolation by quantifier elimination (QE), eliminating all local symbols in A. For the latter, we use the implementation of the Omega [13] test available in Princess. The results are shown in Fig. 5.

The upper left diagram compares runtimes of proof-based interpolation (PBI) with QE, with a timeout of $120s$. We do not include the time to generate proofs, because in typical applications (like software model checking) many interpolants will be generated from each proof, and because QE does not decide the input formula. Considering only the cases without timeout, proving took on average about 4 times as long as the extraction of all interpolants from one proof. The diagram shows that PBI outperforms QE in 147 out of 205 cases, while QE is faster in 58 cases. QE times out for 103 of the benchmarks, PBI for 29. When analyzing the cases where QE is faster than PBI, we observed that QE typically performs well when A only contains few local symbols, i.e., when few quantifiers need to be eliminated. We highlight cases where the number of local symbols is less than 15 by gray points in the diagrams; with an increasing number of local symbols, the performance of QE quickly degrades. To quantify this phenomenon, we measured interpolation runtimes classified by the number of local symbols in A: the two lower diagrams in Fig. 5 show that PBI is a lot less dependent on the number of such symbols than QE.

The upper right diagram compares the sizes of the interpolants (the number of operators) generated by the two techniques. In 149 cases, the interpolants obtained using PBI are smaller than those derived by QE, in 122 cases they are at least one order of magnitude smaller.

7 Related Work and Conclusions

Related work. Interpolation for propositional logic, linear rational arithmetic, and uninterpreted functions is a well-explored field. In particular, McMillan presents an interpolating theorem prover for rational arithmetic and uninterpreted functions [10]; an interpolating SMT solver for the same logic has been developed by Beyer et al. [1]. Rybalchenko et al. [16] introduce an interpolation procedure for this logic that works without constructing proofs.

Interpolation has also been investigated in several fragments of integer arithmetic. McMillan considers the logic of difference-bound constraints [11], which is decidable by reduction to rational arithmetic. As an extension, Cimatti et al. [2] present an interpolation procedure for the $\mathcal{UTVPI}$ fragment of linear integer arithmetic. Both fragments allow efficient reasoning and interpolation, but are not sufficient to express many typical program constructs, such as integer division. In [5], separate interpolation procedures for two theories are presented, namely (i) QFPA restricted to conjunctions of integer linear (dis)equalities and (ii) QFPA restricted to conjunctions of stride constraints. The combination of both fragments with integer linear inequalities is not supported, however.

Kapur et al. [7] prove that full QFPA is closed under interpolation (as an instance of a more general result about recursively enumerable theories), but their proof does not directly give rise to an efficient interpolation procedure. Lynch et al. [8] define an interpolation procedure for linear rational arithmetic, and extend it to integer arithmetic by means of Gomory cuts. No interpolating rule is provided for mixed cuts, however, which means that sometimes formulae are generated that are not true interpolants because they violate the vocabulary condition (i.e., contain symbols that are not common to A and B).

Conclusions. We have presented the first interpolating sequent calculus for quantifier-free Presburger arithmetic, permitting arbitrary combinations of linear integer equalities, inequalities, and stride predicates. Our calculus is intended to be used with a reasoning engine for sequent calculi, resulting in an interpolating decision procedure for Presburger arithmetic. We have implemented our calculus rules in Princess and demonstrated experimentally that our method is able to generate much more succinct interpolants than quantifier elimination, which is the only other method for Presburger interpolation we are aware of.

Currently, we are working on the integration of our interpolation procedure into a software model checker based on lazy abstraction [11]. The model checker uses interpolation to refine the abstraction and avoids the expensive image computation required by predicate abstraction. When using our QFPA interpolation procedure, we expect to be able to verify software with more complex numerical features than other model checkers.

Acknowledgments. We want to thank Jerome Leroux, Vijay D'Silva, Georg Weissenbacher, and the anonymous referees for discussions and/or comments.

References

1. Beyer, D., Zufferey, D., Majumdar, R.: CSIsat: Interpolation for LA+EUF. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 304–308. Springer, Heidelberg (2008)
2. Cimatti, A., Griggio, A., Sebastiani, R.: Interpolant generation for UTVPI. In: Schmidt, R.A. (ed.) CADE 2009. LNCS, vol. 5663, pp. 167–182. Springer, Heidelberg (2009)
3. Craig, W.: Linear reasoning. a new form of the Herbrand-Gentzen theorem. The Journal of Symbolic Logic 22(3), 250–268 (1957)
4. Fitting, M.C.: First-Order Logic and Automated Theorem Proving, 2nd edn. Springer, Heidelberg (1996)
5. Jain, H., Clarke, E., Grumberg, O.: Efficient interpolation for linear diophantine (dis)equations and linear modular equations. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 254–267. Springer, Heidelberg (2008)
6. Kannan, R., Bachem, A.: Polynomial algorithms for computing the Smith and Hermite normal forms of an integer matrix. SIAM J. Comput. 8(4), 499–507 (1979)
7. Kapur, D., Majumdar, R., Zarba, C.G.: Interpolation for data structures. In: SIGSOFT '06/FSE-14, pp. 105–116. ACM, New York (2006)
8. Lynch, C., Tang, Y.: Interpolants for linear arithmetic in SMT. In: Cha, S(S.), Choi, J.-Y., Kim, M., Lee, I., Viswanathan, M. (eds.) ATVA 2008. LNCS, vol. 5311, pp. 156–170. Springer, Heidelberg (2008)
9. Maehara, S.: On the interpolation theorem of Craig. Sugaku 12, 235–237 (1960)
10. McMillan, K.L.: An interpolating theorem prover. Theor. Comput. Sci. 345(1) (2005)
11. McMillan, K.L.: Lazy abstraction with interpolants. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 123–136. Springer, Heidelberg (2006)
12. Presburger, M.: Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In: Comptes Rendus du Ier congrès de Mathématiciens des Pays Slaves (1929)
13. Pugh, W.: The Omega test: a fast and practical integer programming algorithm for dependence analysis. Communications of the ACM 8, 102–114 (1992)
14. Rümmer, P.: A sequent calculus for integer arithmetic with counterexample generation. In: VERIFY. CEUR, vol. 259 (2007), http://ceur-ws.org/
15. 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)
16. Rybalchenko, A., Sofronie-Stokkermans, V.: Constraint solving for interpolation. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 346–362. Springer, Heidelberg (2007)
17. Schrijver, A.: Theory of Linear and Integer Programming. Wiley, Chichester (1986)

Bugs, Moles and Skeletons: Symbolic Reasoning for Software Development

Leonardo de Moura and Nikolaj Bjørner

Microsoft Research, One Microsoft Way, Redmond, WA, 98052, USA
{leonardo,nbjorner}@microsoft.com

Abstract. Symbolic reasoning is in the core of many software development tools such as: bug-finders, test-case generators, and verifiers. Of renewed interest is the use of symbolic reasoning for synthesing code, loop invariants and ranking functions. Satisfiability Modulo Theories (SMT) solvers have been the focus of increased recent attention thanks to technological advances and an increasing number of applications. In this paper we review some of these applications that use software verifiers as bug-finders "on steroids" and suggest that new model finding techniques are needed to increase the set of applications supported by these solvers.

1 Introduction

Symbolic reasoning is present in many diverse areas including software and hardware verification, type inference, static program analysis, test-case generation, scheduling and planning. In the software industry, symbolic reasoning has been successfully used in many test-case generation and bug-finding tools.

Symbolic reasoning is attractive for *verifiation*, but we have found it even more compelling for finding **bugs**. Test-case generation tools produce **moles** that are test inputs which exercise particular program paths and their main goal is to increase code coverage. Of recent interest is the use of **skeletons**, also known as templates or schemas, when using symbolic reasoning in domain specific ways.

We claim these tools are successful in industry because their results, moles and bugs, can be easily digested, and domain specific skeletons are simple to formulate. For example, generated moles can be directly executed on the system under test. So it is straightforward to check and re-use the result from test-case generation tools. We here make a case for the importance of symbolic reasoners supporting the hunt for bugs and moles and the creation of skeletons.

A long-running and natural use of symbolic reasoning tools has been in the context of *program verification*, and indeed, a lot of our experience with symbolic reasoning has been rooted in program verification systems. The ideal of verified software has been a long-running quest since Floyd and Hoare introduced program verification by assigning logical assertions to programs. Yet, the starting point of this paper is making a case that using symbolic reasoning tools are compelling in the context of even partial program exploration and design, and this domain offers compelling challenges for symbolic reasoning systems. The ideal of verified software amounts to a formidable task. It includes grasping with problems that are often quite tangentical to the software being verified. Common pittfalls are that an axiomatization of the environment/runtime may be incorrect and the properties being verified are not the right ones. Such pittfalls are hard to avoid as verification is intimately tied to abstraction. Unfortunately unsound abstractions are so much easier to come around than sound ones. Examples where unsound abstractions creep in include using arithmetic over the integers ($\mathcal{Z}$) instead of machine arithmetic, using memory model simplifications (e.g., pointer arithmetic), and ignoring concurrency. In other cases the challenge is not about simplifying the verification task, but it is about correctly encoding the underlying environment and runtime. Finally, with a Floyd-Hoare proof in house, and a trusted system model, the question is *how can I trust the verifier?* One approach to answering this question is by using *certificates* generated by the symbolic reasoning system. However, certificate generation can produce a significant overhead on automatic theorem provers in terms of memory and time. Another solution is the use of certified theorem provers. Verifying the verifier has become the ultimate distraction.

Independently of all these hurdles, in our point of view, software verification systems can be seen as bug-finding tools with *much better coverage*. Of course, in this case, the tool must be capable of reporting why a proof attempt did not succeeded. This view is used in almost every software verification project at Microsoft. It is not uncommon for these projects to demonstrate value by reporting the discovery of non-trivial bugs in software that was heavily tested by standard techniques. Following this view, a certificate/proof should be seen as a "the verifier cannot find more bugs for you" result.

Between these two extremes, bug-finding and verification, there is another application that is undergoing a rennaisance: synthesis. The idea of synthesizing code is not new, it dates back to the late 60's [12,16]. Due to the recent advances in first-order theorem proving, SMT and QBF solving, it is becoming more feasible to synthesize non trivial glue code [15], small algorithms [17], ranking functions [4] and procedures [14]. The outcome of a synthesis tool is not as simple to check as the one produced by a bug-finding tool, but it is more tangible than a proof of correctness. In principle, developers can inspect and test the synthesized code independently of the symbolic reasoner.

2 Symbolic Reasoning at Microsoft

Z3 [5] is an SMT solver and the main symbolic reasoning engine used at Microsoft. SMT solvers combine the problem of Boolean Satisfiability with domains, such as, those studied in convex optimization and term-manipulating symbolic systems. They involve the decision problem, completeness and incompleteness of logical theories, and finally complexity theory.

2.1 Dynamic Symbolic Execution

SMT solvers play a central role in the context of *dynamic* symbolic execution, also called *smart white-box fuzzing*. There are today several industry applied tools based on dynamic symbolic execution, including CUTE, Exe, DART, SAGE, Pex, and Yogi [11]. These tools collect explored program paths as formulas and use solvers for identifying new test input (moles) that can steer execution into new branches.

SMT solvers are a good fit for symbolic execution because they rely on a solver that can find feasible solutions to logical constraints. They also use combinations of theories that are already supported by the solvers. To illustrate the basic idea of dynamic symbolic execution consider the greatest common divisor program 2.1. It takes the inputs x and y and produces the greatest common divisor of x and y.

```
int GCD(int x,int y) {
  while (true) {
    int m = x % y;
    if (m == 0) return y;
    x = y;
    y = m;
  }
}
```

Program 2.1: GCD Program

Program 2.2 represents the static single assignment unfolding corresponding to the case where the loop is exited in the second iteration. We use assertions to enforce that the condition of the if-statement is not satisfied in the first iteration, and it is in the second. The sequence of instructions is equivalently represented as a formula where the assignment statements have been turned into equations.

```
int GCD(int x0,int y0) {
  int m0 = x0 % y0;
  assert (m0 != 0);
  int x1 = y0;
  int y1 = m0;
  int m1 = x1 % y1;
  assert (m1 == 0);
}
```

$$\begin{array}{ll} (m_0 = x_0 \ \% \ y_0) & \wedge \\ \neg(m_0 = 0) & \wedge \\ (x_1 = y_0) & \wedge \\ (y_1 = m_0) & \wedge \\ (m_1 = x_1 \ \% \ y_1) & \wedge \\ (m_1 = 0) & \end{array}$$

Program 2.2: GCD Path Formula

The resulting path formula is satisfiable. One satisfying assignment that can be found using an SMT solver is of the form:

$$x_0 = 2, \ y_0 = 4, \ m_0 = 2, \ x_1 = 4, \ y_1 = 2, \ m_1 = 0$$

It can be used as input to the original program. In the case of this example, the call `GCD(2,4)` causes the loop to be entered twice, as expected. Smart white-box fuzzing is actively used at Microsoft. It complements traditional black-box fuzzing, where the program being fuzzed is opaque, and fuzzing is performed by pertubing input vectors using random walks. It has been instrumental in uncovering several subtle security critical bugs that black-box methods have been unable to find.

2.2 Static Program Analysis

Static program analysis tools work in a similar way as dynamic symbolic execution tools. They also check feasibility of program paths. On the other hand they can analyze software libraries and utilities independently of how they are used. One advantage of using modern SMT solvers in static program analysis is that SMT solvers nowadays accurately capture the semantics of most basic operations used by commonly used programming languages. We use the program in Figure 1 to illustrate the need for static program analysis to use bit-precise reasoning. The program searches for an index in a sorted array `arr` that contains a key.

```
int binary_search(
  int[] arr, int low, int high, int key) {
    assert (low > high || 0 <= low < high);
    while (low <= high) {
        // Find middle value
        int mid = (low + high) / 2;
        assert (0 <= mid < high);
        int val = arr[mid];
        // Refine range
    if (key == val) return mid;
    if (val > key) low = mid+1;
    else high = mid-1;
    }
    return -1;
}
```

Fig. 1. Binary search

The `assert` statement is a *pre-condition*, for the procedure. It restricts the input to fall within the bounds of the array `arr`. The program performs several operations involving arithmetic, so a theory and corresponding solver that understands arithmetic appears to be a good match. It is however important to take into account that languages, such as Java, C# and C/C++ all use 32-bit integers as the representation for values of type `int`. This means that the accurate theory for `int` is two-complements modular arithmetic. The maximal positive 32-bit integer is $2^{31} - 1$ and the smallest negative 32-bit integer is -2^{31}. If both `low` and `high` are 2^{30}, `low + high` evaluates to 2^{31}, which is treated as the negative number -2^{31}. The presumed assertion $0 \leq mid < high$ therefore does not hold. Fortunately, several modern SMT solvers support the theory of *bit-vectors*, which accurately captures the semantics of modular arithmetic. The bug does not escape an analysis based on the theory of bit-vectors. Such an analysis would check that the array read `arr[mid]` is within bounds during the first iteration by checking the formula:

$$\begin{aligned} & \mathit{low} > \mathit{high} \vee 0 \leq \mathit{low} < \mathit{high} < \mathit{arr.length} \\ \wedge\ & \mathit{low} \leq \mathit{high} \\ \rightarrow\ & 0 \leq (\mathit{low} + \mathit{high})/2 < \mathit{arr.length} \end{aligned}$$

As we saw, the formula is not valid. The values $\texttt{low} = \texttt{high} = 2^{30}$, $\texttt{arr.length} = 2^{30} + 1$ provide a counter-example. An integration with the solver Z3 and the static analysis tool PREfix led to the automatic discovery of several overflow-related bugs in Microsoft's rather large code-base.

2.3 Software Verification

Extended static checking uses the methods developed for program verification, but in the more limited context of checking absence of run-time errors. The SMT solver Simplify [7] was developed in the context of the extended static checking systems ESC/Modula 3 and ESC/Java [10]. This work has been the inspiration for several subsequent extended static program checkers, including Why [9] and Boogie [1]. These systems are actively used as bridges from several different front-ends to SMT solver backends. Boogie, for instance, is used as a backend for systems that verify code from languages, such as an extended version of C# (called Spec#), as well as low level systems code written in C. Current practice indicates that one person can drive these tools to verify selected extended static properties of large code bases with several hundreds of thousands of lines. This effort relies heavily on some of the automated methods used in software model-checking. A more ambitious project is the Verifying C-Compiler system [8], which targets functional correctness properties of Microsoft's Viridian Hyper-Visor. The Hyper-Visor is a relatively small (100K lines) operating system layer, yet correctness properties are challenging to formulate and establish. The entire verification effort is estimated to be around 60 man-years.

2.4 Synthesis

Finally, there is recent and active interest in using modern SMT solvers in the context of synthesis of inductive loop invariants [18] and synthesis of program fragments [14], such as sorting, matrix multiplication, de-compression, graph, and bit-manipulating algorithms. Take for instance the Strassen's matrix multiplication algorithm in the special case of 2×2 matrices. Synthesizing it amounts to arranging a set of (7) multipliers and adders to obtain equivalent results as the standard matrix multiplication algorithm that uses 8 multipliers. The search process can be carried out on a multipliers that manipulate words of length 2-3 bits. The synthesized code can then be checked on full bit-widths (32 or 64 bits). These applications share a common trait in the way they use their underlying symbolic solver. They search a template *vocabulary* of instructions, that are composed as a model in a satisfying assignment. Section 3.3 goes into more detail.

3 Symbolic Reasoning Support for Models

3.1 Streams of Candidate Models

Most SMT solvers are capable of producing models for satisfiable quantifier-free formulas. A model is an interpretation that makes the formula true. For example, the interpretation $\{a \mapsto 2, b \mapsto 5\}$ is a model for the formula $a \geq 0 \wedge b \geq a + 3$. This capability is essential in many industrial applications, because moles and bugs are extracted from models.

Quantifiers are usually used to axiomatize the environment/runtime, state properties, specify frame axioms, etc. For example, the formula $\forall i, j.\ i \leq j \rightarrow f(i) \leq f(j)$ is used to say that f is a non-decreasing function. Quantifier reasoning in SMT is a long-standing challenge. The practical method employed in modern SMT solvers is to instantiate quantified formulas based on heuristics, which is not refutationally complete even for pure first-order logic. Moreover, refutationally complete procedures are not sufficient, since they will only guarantee that a proof of unsatisfiability will be found eventually for unsatisfiable formulas. However, in industry, we are mainly interested in the satisfiable instances, where a refutationally complete procedure may not even terminate. Some SMT solvers support decidable fragments [2,6,20], unfortunately they are not expressive enough to encode all symbolic reasoning problems found in practice.

A pragmatic approach for dealing with the problem above is to produce *candidate models*. Given a formula of the form $F \wedge G$, where G is quantifier-free, a candidate model is an interpretation that satisfies G and many instances of the universally quantified formulas in F. For example, consider the following simple satisfiable formula

$$\begin{array}{r} \overbrace{\forall i, j.\ i \leq j \rightarrow f(i) \leq f(j)}^{F} \wedge \\ \underbrace{w \geq v + 2\ \wedge\ f(v) \leq 1\ \wedge\ f(w) \leq 3}_{G} \end{array}$$

Standard SMT solvers will produce a candidate model such as:

$$v \mapsto 0,\ w \mapsto 2,\ f \mapsto [0 \mapsto 1,\ 2 \mapsto 3,\ \mathit{else} \mapsto 4]$$

The interpretation for f is a *function graph*, it states that the value of the function f at 0 is 1, at 2 is 3, and for all other values is 4. This interpretation satisfies G, and satisfies the instance $v \leq w \rightarrow f(v) \leq f(w)$ of F, but it clearly does not satisfy F.

Candidate models are relevant because they may contain enough information to help the developer to understand why some property does not hold, or some program location is reachable. Moles and bugs may still be extracted from them, and the actual program (i.e., the definitive oracle) can be executed to confirm they are indeed correct. This observation suggests a particular tool flow not very often explored. The basic idea is to use the actual program as an oracle, to check whether the candidate model really induces a valid mole/bug or not. If it does, then the tool terminates. Otherwise, it informs the solver that the candidate model is a not valid, and the search continues. In this approach, the solver is forced to generate a *stream* of more and more refined candidate models until a valid mole/bug can be successfully extracted.

3.2 Model Checking Quantifiers

Given a candidate model I, it is useful to have a procedure P that checks whether the interpretation I satisfies a universally quantified formula F or not. We say P is a *model checking procedure.* To describe how P can be constructed, let us describe how interpretations are particularly encoded in Z3. In Z3, we assume there is an intended interpretation $\mathcal{T}$ for the supported set of theories T. In the case of Z3, T is the union of the following theories: linear arithmetic, bit-vectors, arrays, inductive data-types, and uninterpreted functions. Given a satisfiable formula F, a model I is a function that maps the structure $\mathcal{T}$ that satisfies T, into an expanded structure M that satisfies $F \cup T$. Our models also come equipped with a set of formulas R that restricts the class of structures that satisfy T. For example, if T is the empty theory, then R is just a cardinality constraint on the size of the universe. When needed, we use fresh constant symbols k_1, $\ldots$, k_n (ur-elements) to name the elements in $|M|$ (i.e., the universe of M). In Z3, the interpretation of an uninterpreted symbol s is an expression $I_s[\bar{x}]$, which contains only interpreted symbols and the fresh constants $k_1, \ldots, k_n$. For uninterpreted constants c, $I_c[\bar{x}]$ is just a ground term I_c. For uninterpreted function and predicate symbols, the term $I_s[\bar{x}]$ should be viewed as a *lambda expression.* For example, the candidate model described in the previous section is encoded as:

$$v \mapsto 0, \;\; w \mapsto 2, \;\; f(x) \mapsto ite(x = 0, 1, ite(x = 2, 3, 4))$$

Where $ite(c, t, e)$ is the if-the-else term.

When models are encoded this way, it is straightforward to check whether a universally quantified formula $\forall \bar{x}.\ F[\bar{x}]$ is satisfied by a candidate model or not [20]. Let $F^I[\bar{x}]$ be the formula obtained from $F[\bar{x}]$ by replacing any term $f(\bar{t})$ with $I_f[\bar{t}]$, when f is uninterpreted. We claim a candidate model satisfies $\forall \bar{x}.\ F[\bar{x}]$ if and only if $R \wedge \neg F^I[\bar{s}]$ is unsatisfiable, where $\bar{s}$ is a tuple of fresh constant symbols. In the previous example, the formula $\forall i, j.\ i \leq j \rightarrow f(i) \leq f(j)$ is not satisfied by the candidate model, because the following formula is satisfiable.

$$s_1 \leq s_2 \wedge \neg(ite(s_1 = 0, 1, ite(s_1 = 2, 3, 4)) \leq ite(s_2 = 0, 1, ite(s_2 = 2, 3, 4)))$$

For instance, this formula is satisfied by $\{s_1 \mapsto 1, s_2 \mapsto 2\}$.

Similarly to the oracle-approach based on the actual program, new instances of universally quantified formulas can be extracted from failed model checking attempts. The new instance has the property that it will "block" the current candidate model from being produced again by the solver.

This particular way of encoding models allows Z3 to represent interpretations for function symbols that are not expressible by finite function graphs. For example, the following candidate model

$$v \mapsto 0,\ w \mapsto 2,\ f(x) \mapsto \mathit{ite}(x \leq 0, 1, \mathit{ite}(x \leq 2, 3, 4))$$

is a model for our working example, because the following ground formula is unsatisfiable.

$$s_1 \leq s_2 \land \neg(\mathit{ite}(s_1 \leq 0, 1, \mathit{ite}(s_1 \leq 2, 3, 4)) \leq \mathit{ite}(s_2 \leq 0, 1, \mathit{ite}(s_2 \leq 2, 3, 4)))$$

Candidate models with this particular shape can be automatically computed because our example is in the *array property* decidable fragment [2].

3.3 Skeleton Based Model Finding and Synthesis

Satisfiability solvers have been used to synthesize loop invariants [3,13], code [19], and ranking functions [4]. To illustrate these ideas, consider the following abstract program:

```
pre
while (c) {
  T
}
post
```

In the loop invariant synthesis problem, we want to synthesize a predicate I that can be used to show that *post* holds in the end of the *while-loop*. Let, $\mathit{pre}[s]$ be a formula encoding the set of states reachable before the beginning of the loop, $c[s]$ be the encoding of the entering condition, $T[s, s']$ be the transition relation, and $\mathit{post}[s]$ be the encoding of the property we want to prove. Then, the loop invariant exists if the following formula is satisfiable, and any model can be used to extract the loop invariant.

$$\begin{array}{l} \forall s.\ \mathit{pre}[s] \rightarrow I(s)\ \land \\ \forall s, s'.\ I(s) \land c[s] \land T[s, s'] \rightarrow I(s')\ \land \\ \forall s.\ I(s) \land \neg c[s] \rightarrow \mathit{post}[s] \end{array}$$

Similarly, in the ranking function synthesis problem, we want to synthesize a function *rank* that decreases after each loop iteration. The idea is to use this function to show a particular loop always terminate in the program. This problem can be encoded as the following satisfiability problem.

$$\begin{array}{l} \forall s.\ \mathit{rank}(s) \geq 0\ \land \\ \forall s, s'.\ c[s] \land T[s, s'] \rightarrow \mathit{rank}(s') < \mathit{rank}(s) \end{array}$$

Let us now illustrate these general schemas using the following simple example program. The program increments x and y in lock-step in a loop and we wish to check that the loop terminates and that $y = n$ at the end of the loop.

```
assert(n >= 0);
x = 0; y = 0;
while (x < n) {
   x = x + 1;
   y = y + 1;
}
assert(y == n);
```

For this simple program, the formulas associated with invariant and ranking synthesis problems are:

$$\begin{aligned}
&\forall x, y, n.\ n \geq 0 \wedge x = 0 \wedge y = 0 \rightarrow I(x, y, n)\ \wedge \\
&\forall x, y, n, x', y', n'.\ I(x, y, n) \wedge x < n \wedge x' = x + 1 \wedge y' = y + 1 \wedge n' = n \rightarrow \\
&\qquad\qquad\qquad\qquad I(x', y', n')\ \wedge \\
&\forall x, y, n.\ I(x, y, n) \wedge \neg(x < n) \rightarrow y = n
\end{aligned}$$

and

$$\begin{aligned}
&\forall x, y, n.\ \mathit{rank}(x, y, n) \geq 0\ \wedge \\
&\forall x, y, n, x', y', n'.\ x < n \wedge x' = x + 1 \wedge y' = y + 1 \wedge n' = n \rightarrow \\
&\qquad\qquad\qquad\qquad \mathit{rank}(x', y', n') < \mathit{rank}(x, y, n)
\end{aligned}$$

Both formulas are satisfiable, the following interpretations are models for them:

$$I(x, y) \mapsto x = y \wedge x \leq n$$

and

$$\mathit{rank}(x, y, n) \mapsto \mathit{ite}(x \leq n, n - x, 0)$$

Thus, in principle, these problems can be attacked by any SMT solver with support for universally quantified formulas, and capable of producing models. Unfortunately, to the best of our knowledge, no SMT solver can handle this kind of problem, even when n, x and y range over finite domains, such as machine integers. They will not terminate or give-up in both problems. For these reasons, many synthesis tools only use SMT solvers to decide quantifier-free formulas. In these applications, the SMT solver is usually used to check whether a candidate interpretation for I and *rank* is valid or not. The synthesis tool search for candidate interpretations using *templates*. Abstractly, a template is a **skeleton** that can be instantiated. For example, when searching for a ranking function, the synthesis tool may limit the search to functions that are linear combinations of the input.

This approach can be easily incorporated to SMT solvers that support the techniques described in the previous section. Given a collection of skeletons, the basic idea is to search for models where the intepretation of function and predicate symbols are instances of the given skeletons. We say an SMT solver based on this approach is a *skeleton based model finder*. In this context, an SMT solver may even report a formula to be *unsatisfiable modulo a collection of skeletons*.

Similarly to the approach used to represent models in Z3 (Section 3.2), skeletons are expressions containing free variables, and should be also viewed as lambda expressions. However, skeletons may also contain fresh constants that must be instantiated. For example, the skeleton $ax + b$, where a and b are fresh constants, may be used as a template for the interpretation of unary function symbols. The expressions $x + 1$ ($\{a \mapsto 1, b \mapsto 1\}$) and $2x$ ($\{a \mapsto 2, b \mapsto 0\}$) are instances of this skeleton.

As usual, we assume the input formula is of the form $F \wedge G$, where G is quantifier free. We also assume a collection of skeletons S is provided by the user. First, we use an SMT solver to check whether $F \wedge G$ is satisfiable or not. If it returns *unsat* or *sat*, then we terminate. In practice, for satisfiable instances, the SMT solver will most likely return just a candidate model. Then, for each function symbol f in G, we select a skeleton $s_f[\bar{x}]$ from S. Next, we check whether the following formula is satisfiable or not.

$$F \wedge G \wedge \bigwedge_{f(\bar{t}) \in G} f(\bar{t}) = s_f[\bar{t}]$$

If this formula is unsatisfiable, we conclude that the selected skeletons cannot be used to satisfy the formula. Let C be the set of fresh constants used in the skeletons. So, if the SMT solver returns a candidate model, it must assign values for each constant in C, and these values are used to instantiate the skeletons. After the skeletons are instantiated, the new interpretation for the uninterpreted function symbols can be checked using the model checking technique described in Section 3.2. If the model checking step fails, then new quantifier instances are generated and added to G, and the process continues.

For example, consider the following very simple formula

$$\overbrace{\forall x.\ g(x) \geq 2x \ \wedge\ \forall x.\ f(x) \leq g(x) + 1}^{F} \wedge \\ \underbrace{g(0) \leq 0 \ \wedge\ f(0) \geq 0}_{G}$$

Assume our collection of skeletons S is $\{ax + b\}$. That is, we are looking for models where the interpretation of every function symbol is a linear function. Assume the SMT solver terminates producing a candidate model, then we select $a_f x + b_f$ and $a_g x + b_g$ as the skeletons for f and g respectively. Note that we use a different set of fresh constants for f and g. Then, we check whether the following formula is satisfiable or not.

$$F \wedge G \wedge \underbrace{g(0) = a_g 0 + b_g \wedge f(0) = a_f 0 + b_f}_{E_1}$$

Assume the SMT solver returns a candidate model for the formula above assigning $\{a_g \mapsto 0, b_g \mapsto 0, a_f \mapsto 0, b_f \mapsto 0\}$. So, using this assignment, our interpretation for g and f is the constant function 0. This interpretation satisfies the quantifier $\forall x.\ f(x) \leq g(x) + 1$, but fails to satisfy $\forall x.\ g(x) \geq 2x$, because the induced model checking formula $\neg(0 \geq 2s_1)$ is satisfiable. A possible model is $s_1 \mapsto 1$. Then, instantiating the quantifier with $x = 1$, we obtain a new set of ground formulas $G_1 = G \wedge g(1) \geq 2$. Assume the SMT solver terminates producing a candidate model for $F \wedge G_1$. Then, we check whether the following formula is satisfiable or not.

$$F \wedge G_1 \wedge \underbrace{E_1 \wedge g(1) = a_g + b_g}_{E_2}$$

In this case, we obtain the new assignment $\{a_g \mapsto 2, b_g \mapsto 0, a_f \mapsto 0, b_f \mapsto 0\}$, which corresponds to the interpretations $g(x) \mapsto 2x$ and $f(x) \mapsto 0$. Now, the first quantifier is satisfied, but $\forall x.\ f(x) \leq g(x) + 1$ fails because the model checking formula $\neg(0 \leq 2s_1 + 1)$ is satisfiable. A possible model is $s_1 \mapsto -1$. Then, we obtain $G_2 = G_1 \wedge f(-1) \leq g(-1) + 1$. Similarly to the previous steps, we check the satisfiability of

$$F \wedge G_2 \wedge E_2 \wedge g(-1) = -a_g + b_g \wedge f(-1) = -a_f + b_f$$

We obtain the assignment $\{a_g \mapsto 2, b_g \mapsto 0, a_f \mapsto 2, b_f \mapsto 0\}$, which corresponds to the interpretations $g(x) \mapsto 2x$ and $f(x) \mapsto 2x$. Now, both quantifiers are satisfied by this interpretation and the SMT solver can report $F \wedge G$ as satisfiable.

4 Conclusion

A long-running and natural use of symbolic reasoning tools has been in the context of *program verification*. However, given the many software verification and analysis tools used at Microsoft, we have found that the most attractive are the ones for finding bugs and producing moles. Skeletons enable a new set of promising applications based on synthesis. They are currently applied as layers on top of SMT solvers. We believe that supporting these techniques natively as part of quantifier instantiation engines is a useful and promising technique for searching models of quantified satisfiable formulas.

References

1. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# Programming System: An Overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005)
2. Bradley, A.R., Manna, Z., Sipma, H.B.: What's decidable about arrays? In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 427–442. Springer, Heidelberg (2005)
3. Colón, M.: Schema-guided synthesis of imperative programs by constraint solving. In: Etalle, S. (ed.) LOPSTR 2004. LNCS, vol. 3573, pp. 166–181. Springer, Heidelberg (2005)
4. Cook, B., Kroening, D., Rümmer, P., Wintersteiger, C.M.: Ranking function synthesis for bit-vector relations. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 236–250. Springer, Heidelberg (2010)

5. de Moura, L., Bjørner, N.S.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
6. de Moura, L., Bjørner, N.: Deciding Effectively Propositional Logic using DPLL and substitution sets. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 410–425. Springer, Heidelberg (2008)
7. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM 52(3), 365–473 (2005)
8. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A Practical System for Verifying Concurrent C. In: TPHOL (2009)
9. Filliâtre, J.-C.: Why: a multi-language multi-prover verification tool. Technical Report 1366, LRI, Université Paris Sud (2003)
10. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended Static Checking for Java. In: PLDI, pp. 234–245 (2002)
11. Godefroid, P., de Halleux, J., Nori, A.V., Rajamani, S.K., Schulte, W., Tillmann, N., Levin, M.Y.: Automating Software Testing Using Program Analysis. IEEE Software 25(5), 30–37 (2008)
12. Green, C.C.: Application of theorem proving to problem solving. In: IJCAI, pp. 219–240 (1969)
13. Gulwani, S., Srivastava, S., Venkatesan, R.: Constraint-based invariant inference over predicate abstraction. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 120–135. Springer, Heidelberg (2009)
14. Jha, S., Gulwani, S., Seshia, S., Tiwari, A.: Oracle-guided component-based program synthesis. In: ICSE (to appear, 2010)
15. Lowry, M.R., Philpot, A., Pressburger, T., Underwood, I.: Amphion: Automatic programming for scientific subroutine libraries. In: Raś, Z.W., Zemankova, M. (eds.) ISMIS 1994. LNCS, vol. 869, pp. 326–335. Springer, Heidelberg (1994)
16. Manna, Z., Waldinger, R.J.: Toward automatic program synthesis. ACM Commun. 14(3), 151–165 (1971)
17. Solar-Lezama, A., Tancau, L., Bodik, R., Saraswat, V., Seshia, S.A.: Combinatorial sketching for finite programs. In: ASPLOS (2006)
18. Srivastava, S., Gulwani, S.: Program Verification using Templates over Predicate Abstraction. In: PDLI (2009)
19. Srivastava, S., Gulwani, S., Foster, J.: From program verification to program synthesis. In: POPL (2010)
20. Ge, Y., de Moura, L.: Complete instantiation for quantified SMT formulas. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 306–320. Springer, Heidelberg (2009)

Automating Security Analysis: Symbolic Equivalence of Constraint Systems[⋆]

Vincent Cheval, Hubert Comon-Lundh, and Stéphanie Delaune

LSV, ENS Cachan & CNRS & INRIA Saclay Île-de-France

Abstract. We consider security properties of cryptographic protocols, that are either trace properties (such as confidentiality or authenticity) or equivalence properties (such as anonymity or strong secrecy).

Infinite sets of possible traces are symbolically represented using *deducibility constraints*. We give a new algorithm that decides the trace equivalence for the traces that are represented using such constraints, in the case of signatures, symmetric and asymmetric encryptions. Our algorithm is implemented and performs well on typical benchmarks. This is the first implemented algorithm, deciding symbolic trace equivalence.

1 Introduction

Security protocols are small distributed programs aiming at some security goal, though relying on untrusted communication media. Formally proving that such a protocol satisfies a security property (or finding an attack) is an important issue, in view of the economical and social impact of a failure.

Starting in the 90s, several models and automated verification tools have been designed. For instance both protocols, intruder capabilities and security properties can be formalized within first-order logic and dedicated resolution strategies yield relevant verification methods [18,21,6]. Another approach, initiated in [19], consists in symbolically representing the traces using deducibility constraints. Both approaches were quite successful in finding attacks/proving security protocols. There are however open issues, that concern the extensions of the methods to larger classes of protocols/properties [11]. For instance, most efforts and successes only concerned, until recently, *trace properties*, i.e., security properties that can be checked on each individual sequence of messages corresponding to an execution of the protocol. A typical example of a trace property is the *confidentiality*, also called *weak secrecy*: a given message m should not be deducible from any sequence of messages, that corresponds to an execution of the protocol. Agreement properties, also called *authenticity properties*, are other examples of trace properties.

There are however security properties that cannot be stated as properties of a single trace. Consider for instance a voter casting her vote, encrypted with a public key of a server. Since there are only a fixed, known, number of possible plaintexts, the confidentiality is not an issue. A more relevant property is the

⋆ This work has been partially supported by the ANR project SeSur AVOTÉ.

ability to relate the voter's identity with the plaintext of the message. This is a property in the family of *privacy* (or *anonymity*) properties [15]. Another example is the *strong secrecy*: m is strongly secret if replacing m with any m' in the protocol, would yield another protocol that is indistinguishable from the first one: not only m itself cannot be deduced, but the protocol also does not leak any piece of m. These two examples are not trace properties, but *equivalence properties*: they can be stated as the indistinguishability of two processes. In the present paper, we are interested in automating the proofs of equivalence properties. As far as we know, there are only three series of works that consider the automation of equivalence properties for security protocols[1].

The first one [7] is an extension of the first-order logic encoding of the protocols and security properties. The idea is to overlap the two processes that are supposedly equivalent, forming a *bi-process*, then formalize in first-order logic the simultaneous moves (the single move of the bi-process) upon reception of a message. This method checks a stronger equivalence than observational equivalence, hence it fails on some simple (cook up) examples of processes that are equivalent, but their overlapping cannot be simulated by the moves of a single bi-process. The procedure might also not terminate or produce false attacks, but considers an unbounded number of protocol instances.

The second one [3] (and [14]) assumes a fixed (bounded) number of sessions. Because of the infinite number of possible messages forged by an attacker, the number of possible traces is still infinite. The possible traces of the two processes are symbolically represented by two deducibility constraints. Then [3] provides with a decision procedure, roughly checking that the solutions, *and the recipes that yield the solutions* are identical for both constraints. This forces to compute the solutions and the associated recipes and yields an unpractical algorithm.

The third one [17,9] is based on an extension of the small attack property of [20]. They show that, if two processes are not equivalent, then there must exist a small witness of non-equivalence. A decision of equivalence can be derived by checking every possible small witness. As in the previous method, the main problem is the practicality. The number of small witnesses is very large as all terms of size smaller than a given bound have to be considered. Consequently, neither this method nor the previous one have been implemented.

We propose in this paper another algorithm for deciding equivalence properties. As in [3,9], we consider *trace equivalence*, which coincides with observational equivalence for determinate processes [14]. In that case, the equivalence problem can be reduced to the symbolic equivalence of finitely many pairs of deducibility constraints, each of which represents a set of traces (see [14]). We consider signatures, pairing, symmetric and asymmetric encryptions, which is slightly less general than [3,9], who consider arbitrary subterm-convergent theories. The main idea of our method is to simultaneously solve pairs of constraints, instead of solving each constraint separately and comparing the solutions, as in [3]. These pairs are successively split into several pairs of systems, while preserving the symbolic

[1] [16] gives a logical characterization of the equivalence properties. It is not clear if this can be of any help in deriving automated decision procedures.

equivalence: roughly, the father pair is in the relation if, and only if, all the sons pairs are in the relation. This is not fully correct, since, for termination purposes, we need to keep track of some earlier splitting, using additional predicates. Such predicates, together with the constraint systems, yield another notion of equivalence, which is preserved upwards, while the former is preserved downwards. When a pair of constraints cannot be split any more, then the equivalence can be trivially checked.

A preliminary version of the algorithm has been implemented and works well (within a few seconds) on all benchmarks. The same implementation can also be used for checking the static equivalence and for checking the constraints satisfiability. We also believe that it is easier (w.r.t. [3,9]) to extend the algorithm to a more general class of processes (including disequality tests for instance) and to avoid the detour through trace equivalence. This is needed to go beyond the class of determinate processes.

We first state precisely the problem in Section 2, then we give the algorithm, actually the transformation rules, in Section 3. We sketch the correctness and termination proofs in Section 4 and provide with a short summary of the experiments in Section 5. Detailed proofs of the results can be found in [8].

2 Equivalence Properties and Deducibility Constraints

We use the following straightfoward example for illustrating some definitions:

Example 1. Consider the following very simple handshake protocol:

$$\begin{array}{l} A \to B : \mathsf{enc}(N_A, K_{AB}) \\ B \to A : \mathsf{enc}(f(N_A), K_{AB}) \end{array}$$

The agent A sends a random message N_A to B, encrypted with a key K_{AB}, that is shared by A and B only. The agent B replies by sending $f(N_A)$ encrypted with the same key. The function f is any function, for instance a hash function.

Consider only one session of this protocol: a sends $\mathsf{enc}(n_a, k_{ab})$ and waits for $\mathsf{enc}(f(n_a), k_{ab})$. The agent b is expecting a message of the form $\mathsf{enc}(x, k_{ab})$. The variable x represents the fact that b does not know in advance what is this randomly generated message. Then he replies by sending out $\mathsf{enc}(f(x\sigma), k_{ab})$. All possible executions are obtained by replacing x with any message $x\sigma$ such that the attacker can supply with $\mathsf{enc}(x\sigma, k_{ab})$ and then with $\mathsf{enc}(f(n_a), k_{ab})$. This is represented by the following constraint:

$$C := \left\{ \begin{array}{r} a,\ b,\ \mathsf{enc}(n_a, k_{ab}) \overset{?}{\vdash} \mathsf{enc}(x, k_{ab}) \\ a,\ b,\ \mathsf{enc}(n_a, k_{ab}),\ \mathsf{enc}(f(x), k_{ab}) \overset{?}{\vdash} \mathsf{enc}(f(n_a), k_{ab}) \end{array} \right.$$

Actually, C has only one solution: x has to be replaced by n_a. There is no other way for the attacker to forge a message of the form $\mathsf{enc}(x, k_{ab})$.

2.1 Function Symbols and Terms

We will use the set of function symbols $\mathcal{F} = \mathcal{N} \cup \mathcal{C} \cup \mathcal{D}$ where:

- $\mathcal{C} = \{\mathsf{enc}, \mathsf{aenc}, \mathsf{pub}, \mathsf{sign}, \mathsf{vk}, \langle\ \rangle\}$ is the set of *constructors*;
- $\mathcal{D} = \{\mathsf{dec}, \mathsf{adec}, \mathsf{check}, \mathsf{proj}_1, \mathsf{proj}_2\}$ is the set of *destructors*;
- $\mathcal{N}$ is a set of constants, called *names*.

In addition, $\mathcal{X}$ is a set of variables x, y, z,... The *constructor terms* (resp. *ground constructor terms*) are built on $\mathcal{C}$, $\mathcal{N}$ and $\mathcal{X}$ (resp. $\mathcal{C}, \mathcal{N}$). The term rewriting system below is convergent: we let $t\!\downarrow$ be the normal form of t.

$$\begin{array}{lll}\mathsf{adec}(\mathsf{aenc}(x, \mathsf{pub}(y)), y) \to x & \mathsf{proj}_1(\langle x, y\rangle) \to x & \mathsf{dec}(\mathsf{enc}(x, y), y) \to x \\ \mathsf{check}(\mathsf{sign}(x, y), \mathsf{vk}(y)) \to x & \mathsf{proj}_2(\langle x, y\rangle) \to y & \end{array}$$

A (ground) *recipe* records the attacker's computation. It is used as a witness of how some deduction has been performed. Formally, it is a term built on $\mathcal{C}, \mathcal{D}$ and a set of special variables $\mathcal{AX} = \{ax_1, \ldots, ax_n, \ldots\}$, that can be seen as pointers to the hypotheses, or known messages. Names are excluded from recipes: names that are known to the attacker must be given explicitly as hypotheses.

Example 2. Given $\mathsf{enc}(a, b)$ and b, the recipe $\zeta = \mathsf{dec}(ax_1, ax_2)$ is a witness of how to deduce a: $\zeta\{ax_1 \mapsto \mathsf{enc}(a, b); ax_2 \mapsto b\}\!\downarrow = a$.

The recipes are generalized, including possibly variables that range over recipes: (general) recipes are terms built on $\mathcal{C}, \mathcal{D}, \mathcal{AX}$ and $\mathcal{X}_r$, a set of recipe variables, that are written using capital letters $X, X_1, X_2, \ldots$.

We denote by $var(u)$ is the set of variables of any kind that occur in u.

2.2 Frames

The *frame* records the messages that have been sent by the participants of the protocol; it is a symbolic representation of a set of sequences of messages. The frame is also extended to record some additional informations on attacker's deductions. Typically $\mathsf{dec}(X, \zeta), i \triangleright u$ records that, using a decryption with the recipe ζ, on top of a recipe X, allows to get u (at stage i). After recording this information in the frame, we may forbid the attacker to use a decryption on top of X, forcing him to use this "direct access" from the frame.

Definition 1. *A* frame *ϕ is a sequence $\zeta_1, i_1 \triangleright u_1, \ldots, \zeta_n, i_n \triangleright u_n$ where $u_1, \ldots, u_n$ are constructor terms, $i_1, \ldots, i_n \in \mathbb{N}$, and $\zeta_1, \ldots, \zeta_n$ are general recipes. The* domain *of the frame ϕ, denoted $dom(\phi)$, is the set $\{\zeta_1, \ldots, \zeta_n\} \cap \mathcal{AX}$. It must be equal to $\{ax_1, \ldots, ax_m\}$ for some m that is called the* size *of ϕ. A frame is* closed *when $u_1, \ldots, u_n$ are ground terms and $\zeta_1, \ldots, \zeta_n$ are ground recipes.*

Example 3. The messages of Example 1 are recorded in a frame of size 4.

$$\{ax_1, 1 \triangleright a,\ ax_2, 2 \triangleright b,\ ax_3, 3 \triangleright \mathsf{enc}(n_a, k_{ab}),\ ax_4, 4 \triangleright \mathsf{enc}(f(x), k_{ab})\}.$$

A frame ϕ defines a substitution $\{ax \mapsto u \mid ax \in dom(\phi), ax \triangleright u \in \phi\}$. A closed frame is *consistent* if, for every $\zeta \triangleright u \in \phi$, we have that $\zeta\phi\!\downarrow = u$.

2.3 Deducibility Constraints

The following definitions are consistent with [12]. We generalize however the usual definition, including equations between recipes, for example, in order to keep track of some choices in our algorithm.

Definition 2. *A* deducibility constraint *(sometimes called simply* constraint *in what follows) is either $\perp$ or consists of:*

1. *a subset S of $\mathcal{X}$ (the free variables of the constraint);*
2. *a frame ϕ, whose size is some m;*
3. *a sequence $X_1, i_1 \overset{?}{\vdash} u_1;\ \ldots;\ X_n, i_n \overset{?}{\vdash} u_n$ where*
   - *$X_1, \ldots, X_n$ are distinct variables in $\mathcal{X}_r$, $u_1, \ldots, u_n$ are constructor terms, and $0 \leq i_1 \leq \ldots \leq i_n \leq m$.*
   - *for every $0 \leq k \leq m$, $var(ax_k\phi) \subseteq \bigcup_{i_j<k} var(u_j)$;*
4. *a conjunction E of equations and disequations between terms;*
5. *a conjunction E' of equations and disequations between recipes.*

The variables X_i represent the recipes that might be used to deduce the right hand side of the deducibility constraint. The indices indicate which initial segment of the frame can be used. We use this indirect representation, instead of the seemingly simpler notation of Example 1, because the transformation rules that will change the frame don't need then to be reproduced on all relevant left sides of deducibility constraints.

Example 4. Back to Example 1, the deducibility constraint is formally given by $S = \{x, y\}$, $E = E' = \emptyset$, the frame ϕ as in Example 3 and the sequence:

$$D \ = \ X_1, 3 \overset{?}{\vdash} \mathsf{enc}(x, k_{ab});\ \ X_2, 4 \overset{?}{\vdash} \mathsf{enc}(f(n_a), k_{ab}).$$

For sake of simplicity, in what follows, we will forget about the first component (the free variables). This is justified by an invariant of our transformation rules: initially all variables are free and each time new variables are introduced, their assignment is determined by an assignment of the free variables.

Definition 3. *A* solution *of a deducibility constraint $C = (\phi, D, E, E')$ consists of a mapping σ from variables to ground constructor terms and a substitution θ mapping $\mathcal{X}_r$ to ground recipes, such that:*

- *for every $\zeta, i \rhd u \in \phi$, $var(\zeta\theta) \subseteq \{ax_1, \ldots, ax_i\}$ and $\zeta\theta(\phi\sigma)\downarrow = u\sigma\downarrow$ (i.e. the frame is consistent after instanciating the variables);*
- *for every $X_i, j \overset{?}{\vdash} u_i$ in D, $var(X_i\theta) \subseteq \{ax_1, \ldots, ax_j\}$ and $X_i\theta(\phi\sigma)\downarrow = u_i\sigma\downarrow$;*
- *for every equation $u \overset{?}{=} v$ (resp. $u \overset{?}{\neq} v$) in E, $u\sigma\downarrow = v\sigma\downarrow$ (resp. $u\sigma\downarrow \neq v\sigma\downarrow$);*
- *for every equation $\zeta \overset{?}{=} \zeta'$ (resp. $\zeta \overset{?}{\neq} \zeta'$) in E', $\zeta\theta = \zeta'\theta$ (resp. $\zeta\theta \neq \zeta'\theta$).*

$\mathsf{Sol}(C)$ *is the set of solutions of C. By convention,* $\mathsf{Sol}(\perp) = \emptyset$*.*

Example 5. Coming back to Example 4, a solution is (σ, θ) with:

- $\sigma = \{x \mapsto n_a,\ y \mapsto \langle a, \mathsf{enc}(n_a, k_{ab})\rangle\}$, and
- $\theta = \{X_1 \mapsto ax_3,\ X_2 \mapsto ax_4,\ X_3 \mapsto \langle ax_1, ax_3\rangle\}$.

Each solution of a constraint corresponds to a possible execution of the protocol, together with the attacker's actions that yield this execution. For instance an attack on the confidentiality of a term s can be modeled by adding $X, m \overset{?}{\vdash} s$ to the constraint system (X is a fresh variable and m is the size of the frame). This represents the derivability of s from the messages sent so far. Note that there might be several attacker's recipes yielding the same trace.

Example 6. Consider another very simple example: the Encrypted Password Transmission protocol [13], which is informally described by the rules:

$$\begin{array}{l} A \rightarrow B : \langle N_A, \mathsf{pub}(K_A)\rangle \\ B \rightarrow A : \mathsf{aenc}(\langle N_A, P\rangle, \mathsf{pub}(K_A)) \end{array}$$

Assume that a first sends a message whereas b is waiting for a message of the form $\langle x, \mathsf{pub}(k_a)\rangle$. Then b responds by sending $\mathsf{aenc}(\langle x, p\rangle, \mathsf{pub}(k_a))$. The corresponding deducibility constraint is (S, ϕ, D, E, E') where $S = \{x, y\}$, $E = E' = \emptyset$, and the sequences ϕ and D are as follows:

$$\phi = \begin{cases} ax_1, 1 \rhd \mathsf{pub}(k_a);\ \ ax_2, 2 \rhd \mathsf{pub}(k_b); \\ ax_3, 3 \rhd \langle n_a, \mathsf{pub}(k_a)\rangle; \\ ax_4, 4 \rhd \mathsf{aenc}(\langle x, p\rangle, \mathsf{pub}(k_a)) \end{cases} \qquad D = \begin{cases} X_1, 3 \overset{?}{\vdash} \langle x, \mathsf{pub}(k_a)\rangle \\ X_2, 4 \overset{?}{\vdash} \mathsf{aenc}(\langle n_a, y\rangle, \mathsf{pub}(k_a)) \end{cases}$$

There are several solutions. For instance, the "honest solution" (σ_h, θ_h) is given by $\sigma_h = \{x \mapsto n_a, y \mapsto p\}$ and $\theta_h = \{X_1 \mapsto ax_3, X_2 \mapsto ax_4\}$. Another solution is (σ, θ) where $\sigma = \{x \mapsto \mathsf{pub}(k_a), y \mapsto n_a\}$ and $\theta = \{X_1 \mapsto \langle ax_1, ax_1\rangle, X_2 \mapsto \mathsf{aenc}(\langle \mathsf{proj}_1(ax_3), \mathsf{proj}_1(ax_3)\rangle, ax_1)\}$.

2.4 Static Equivalence

Two sequences of terms are *statically equivalent* if, whatever an attacker observes on the first sequence, the same observation holds on the second sequence [2]:

Definition 4. *Two closed frames ϕ and ϕ' having the same size m are* statically equivalent*, which we write $\phi \sim_s \phi'$, if*

1. *for any ground recipe ζ such that $var(\zeta) \subseteq \{ax_1, \ldots, ax_m\}$, we have that*
$$\zeta\phi\!\downarrow \textit{ is a constructor term if, and only if, } \zeta\phi'\!\downarrow \textit{ is a constructor term}$$
2. *for any ground recipes ζ, ζ' such that $var(\{\zeta, \zeta'\}) \subseteq \{ax_1, \ldots, ax_m\}$, and the terms $\zeta\phi\!\downarrow$, $\zeta'\phi\!\downarrow$ are constructor terms, we have that*
$$\zeta\phi\!\downarrow = \zeta'\phi\!\downarrow \textit{ if, and only, if } \zeta\phi'\!\downarrow = \zeta'\phi'\!\downarrow.$$

Example 7. Consider the frames $\phi_1 = \{ax_1 \rhd a, ax_2 \rhd \mathsf{enc}(a, b), ax_3 \rhd b\}$ and $\phi_2 = \{ax_1 \rhd a, ax_2 \rhd \mathsf{enc}(c, b), ax_3 \rhd b\}$. $\phi_1 \not\sim_s \phi_2$ since choosing $\zeta = \mathsf{dec}(ax_2, ax_3)$ and $\zeta' = ax_1$ yields $\zeta\phi_1\!\downarrow = \zeta'\phi_1\!\downarrow = a$ while $\zeta\phi_2\!\downarrow \neq \zeta'\phi_2\!\downarrow$.

On the other hand, $\{ax_1 \rhd a, ax_2 \rhd \mathsf{enc}(a,b)\} \sim_s \{ax_1 \rhd a, ax_2 \rhd \mathsf{enc}(c,b)\}$ since, intuitively, there is no way to open the ciphertexts or to construct them, hence no information on the content may leak.

2.5 Symbolic Equivalence

Now we wish to check static equivalence on any possible trace. This is captured by the following definition:

Definition 5. *Let C and C' be two constraints whose corresponding frames are ϕ and ϕ'. C is* symbolically equivalent *to C', $C \approx_s C'$, if:*
- for all $(\theta, \sigma) \in \mathsf{Sol}(C)$, there exists σ' such that $(\theta, \sigma') \in \mathsf{Sol}(C')$, and $\phi\sigma \sim_s \phi'\sigma'$,
- for all $(\theta, \sigma') \in \mathsf{Sol}(C')$, there exists σ such that $(\theta, \sigma) \in \mathsf{Sol}(C)$, and $\phi\sigma \sim_s \phi'\sigma'$.

Example 8. As explained for instance in [3], the security of the handshake protocol against offline guessing attacks can be modeled as an equivalence property between two samples of the protocol instance, one in which, at the end of the protocol, the key is revealed and the other in which a random number is revealed instead. This amounts to check the symbolic equivalence of the two constraints:

- $C_1 = (\phi \cup \{ax_5, 5 \rhd k_{ab}\}, D \cup \{X_3, 5 \overset{?}{\vdash} y\}, \emptyset, \emptyset)$, and
- $C_2 = (\phi \cup \{ax_5, 5 \rhd k\}, D \cup \{X_3, 5 \overset{?}{\vdash} y\}, \emptyset, \emptyset)$

where D is as in Example 4 and ϕ is as in Example 3.

The constraints C_1 and C_2 are *not* symbolically equivalent: considering the assignment $\sigma = \{x \mapsto n_a, y \mapsto n_a\}$, there is a recipe $X_3\theta = \mathsf{dec}(ax_3, ax_5)$ yielding this solution, while any solution σ' of C_2 maps x to n_a and, if $X_3\theta = \mathsf{dec}(ax_3, ax_5)$, we must have $y\sigma'\downarrow = \mathsf{dec}(\mathsf{enc}(n_a, k_{ab}), k)$, which is not possible since this is not a constructor term.

Any trace equivalence problem can be expressed as an instance of the equivalence of an *initial pair of constraints*, that is a pair of the form (ϕ_1, D_1, E_1, E_1'), (ϕ_2, D_2, E_2, E_2') in which:

- $E_1' = E_2' = \emptyset$, and E_1, E_2 only contain equations;
- $\phi_1 = \{ax_1, 1 \rhd u_1, \ldots, ax_m, m \rhd u_m\}$, and $D_1 = X_1, i_1 \overset{?}{\vdash} s_1;\ \ldots;\ X_n, i_n \overset{?}{\vdash} s_n$;
- $\phi_2 = \{ax_1, 1 \rhd v_1, \ldots, ax_m.m \rhd v_m\}$, and $D_2 = X_1, i_1 \overset{?}{\vdash} t_1;\ \ldots;\ X_n, i_n \overset{?}{\vdash} t_n$.

Or else it is a pair as above, in which one of the components is replaced with $\perp$.

In particular, the number of components in the frame and in the deducibility part are respectively identical in the two constraints, when none of them is $\perp$. *This will be an invariant in all our transformation rules.* Hence we will always assume this without further mention. This is unchanged by the transformations, unless the constraint becomes $\perp$. We keep the notation m for the size of the frames. Finally, the consistency of the frame after instanciation (the first condition of Definition 3) is satisfied for all solutions of initial constraints and is again an invariant, hence we will not care of this condition.

As explained in [14], such initial constraints are sufficient for our applications. The case where one of the component is $\perp$ solves the satisfiability problem for the constraint: the constraint solving procedure of [12] solves this specific instance.

3 Transformation Rules

The main result of this paper is a decision procedure for symbolic equivalence of an initial pair of constraints:

Theorem 1. *Given an initial pair (C, C'), it is decidable whether $C \approx_s C'$.*

This result in itself is already known (e.g. [3,9]), but, as claimed in the introduction, the known algorithms cannot yield any reasonable implementation. We propose here a new algorithm/proof, which is implemented. As pointed in [14], this yields a decision algorithm for the observational equivalence of simple processes without replication nor else branch. The class of simple processes captures most existing protocols.

The decision algorithm works by rewriting pairs of constraints, until a trivial failure or a trivial success is found. These rules are branching: they rewrite a pair of constraints into two pairs of constraints. Transforming the pairs of constraints therefore builds a binary tree. Termination requires to keep track of some information, that is recorded using flags, which we describe first. In Section 4, we show that the tree is then finite: the rules are terminating. The transformation rules are also correct: if all leaves are success leaves, then the original pair of constraints is equivalent. They are finally complete: if the two original constraints are equivalent then any of two pairs of constraints resulting from a rewriting steps are also equivalent.

3.1 Flags

The flags are additional constraints that restrict the recipes. We list them here, together with (a sketch of) their semantics.

Constraints $X, i \overset{?}{\vdash}_F u$ may be indexed with a set F consisting of propositions $\texttt{NoCons}_f$ where f is a constructor. Any solution (θ, σ) such that $X\theta$ is headed with f is then excluded. Expressions $\zeta, j \rhd_F u$ in a frame are indexed with a set F consisting of:

- $\texttt{NoCons}_f$ (as above) discards the solutions (θ, σ) such that a subterm of a recipe allows to deduce $u\sigma$ using f as a last step.
- $\texttt{NoDest}_f(i)$ where f is a destructor and $i \leq m$ discards the solutions (θ, σ) such that there exists $X, j \overset{?}{\vdash} v$ with $j \leq i$ and $\zeta'_2, \ldots, \zeta'_n$ where $f(\zeta\theta, \zeta'_2, \ldots, \zeta'_n)$ occurs as a subterm in $X\theta$, unless we use a shortcut explicitly given in the frame.
- $\texttt{NoUse}$. The corresponding elements of the frame cannot be used in any recipe, and avoids shifting the indices.

3.2 The Rules

The rules are displayed in Figure 1 for single constraints. We explain in Section 3.3 how they are applied to pairs of constraints (an essential feature of our algorithm). A simple idea would be to guess the top function symbol of a recipe and replace the recipe variable with the corresponding instance. When the head symbol of a recipe is a constructor and the corresponding term is not a variable, this is nice, since the constraint becomes simpler. This is the purpose of the rule Cons. When the top symbol of a recipe is a destructor, the constraint becomes more complex, introducing new terms, which yields non-termination.

Our strategy is different for destructors: we switch roughly from the top position of the recipe to the *redex position*. Typically, in case of symmetric encryption, if a ciphertext is in the frame, we will guess whether the decryption key is deducible, and at which stage.

The Cons rule simply guesses whether the top symbol of the recipe is a constructor f. Either it is, and then we can split the constraint, or it is not and we add a flag forbidding this. The rule Axiom also guesses whether a trivial recipe can be applied. If so, the constraint can simply be removed. Otherwise, it means that the right-hand-side of the deducibility constraint is different from the members of the frame. The Dest rule is more tricky. If v is a non-variable member of the frame, that can be unified with a non variable subterm of a left side of a rewrite rule (for instance v is a ciphertext), we guess whether the rule can be applied to v. This corresponds to the equation $u_1 \overset{?}{=} v$, that yields an instance of w, the right member of the rewrite rule, provided that the rest of the left member is also deducible: we get constraints $X_2, i \overset{?}{\vdash} u_2; \ldots ; X_n, i \overset{?}{\vdash} u_n$. The flag `NoDest` is added in any case to the frame, since we either already applied the destructor, and this application result is now recorded in the frame by $f(\zeta, X_2, \ldots, X_n), i \rhd w$, or else it is assumed that f applied to v will not yield a redex.

The remaining rules cover the comparisons that an attacker could perform at various stages. The equality rules guess equalities between right sides of deducibility constraints and/or members of the frame. If a member of the frame is deducible at an early stage, then this message does not bring any new information to the attacker: it becomes useless, hence the `NoUse` flag.

Finally, the last rule is the only rule that is needed to get in addition a static equivalence decision algorithm, as in [1]. Thanks to this rule, if a subterm of the frame is deducible, then there will be a branch in which it is deduced.

3.3 How to Use the Transformation Rules

In the previous section we gave rules that apply on a single constraint. We explain here how they are extended to pairs of constraints. If one of the constraint is $\perp$, then we proceed as if there was a single constraint. Otherwise, the indices i (resp. i_1, i_2) and the recipes X, ζ (resp. $X_1, X_2, \zeta_1, \zeta_2$) matching the left side of the rules *must be identical in both constraints*: we apply the rules at the same positions in both constraints.

$$\begin{tikzcd}[column sep=small] & X_1, i \overset{?}{\vdash}_F t_1; \cdots ; X_n, i \overset{?}{\vdash}_F t_n; X \overset{?}{=} f(X_1, \ldots, X_n) \\ \underline{\textsc{Cons}} : X, i \overset{?}{\vdash}_F f(t_1, \ldots, t_n) \arrow[ur] \arrow[dr] & \\ & X, i \overset{?}{\vdash}_{F+\texttt{NoCons}_f} f(t_1, \ldots, t_n) \end{tikzcd}$$

If $\texttt{NoCons}_f \notin F$ and $X_1, \ldots X_n$ are fresh variables.

$$\begin{tikzcd}[column sep=small] & u \overset{?}{=} v;\ X \overset{?}{=} \zeta \\ \underline{\textsc{Axiom}} : X, i \overset{?}{\vdash}_F v \arrow[ur] \arrow[dr] & \\ & X, i \overset{?}{\vdash}_F v;\ X \overset{?}{\neq} \zeta \end{tikzcd}$$

If $v \notin \mathcal{X}$, ϕ contains $\zeta, j \rhd_G u$ with $\texttt{NoUse} \notin G$, and $i \geq j$.

$$\begin{tikzcd}[column sep=small] & X_2, i \overset{?}{\vdash} u_2;\ \cdots ;\ X_n, i \overset{?}{\vdash} u_n;\ u_1 \overset{?}{=} v;\ \zeta, j \rhd_{G+\texttt{NoDest}_f(m)} v; \\ \underline{\textsc{Dest}} : \zeta, y \rhd_G v \arrow[ur] \arrow[dr] & f(\zeta, X_2, \ldots, X_n), i \rhd w \\ & \zeta, j \rhd_{G+\texttt{NoDest}_f(i)} v \end{tikzcd}$$

If $v \notin \mathcal{X}$, $\texttt{NoUse} \notin G$, there is a rewrite rule $f(u_1, \ldots, u_n) \to w$, $k < i$ whenever $\texttt{NoDest}_f(k) \in G$ and i is minimal such that $j \leq i$ and there is some constraint $X, i \overset{?}{\vdash} w$ ($i = m$ if there is no such constraint).

$$\begin{tikzcd}[column sep=small] & \zeta_1, i_1 \rhd_{F_1} u_1;\ \zeta_2, i_2 \rhd_{F_2} u_1;\ u_1 \overset{?}{=} u_2 \\ \underline{\textsc{Eq-left-left}} : \zeta_1, i_1 \rhd_{F_1} u_1;\ \zeta_2, i_2 \rhd_{F_2} u_2 \arrow[ur] \arrow[dr] & \\ & \zeta_1, i_1 \rhd_{F_1} u_1;\ \zeta_2, i_2 \rhd_{F_2} u_2;\ u_1 \overset{?}{\neq} u_2 \end{tikzcd}$$

If $\texttt{NoUse} \notin F_1 \cup F_2$ and $i_1 \leq i_2$.

$$\begin{tikzcd}[column sep=small] & X_1 = X_2;\ u_1 \overset{?}{=} u_2 \\ \underline{\textsc{Eq-right-right}} : X_2, i_2 \overset{?}{\vdash} u_2 \arrow[ur] \arrow[dr] & \\ & X_2, i_2 \overset{?}{\vdash} u_2;\ u_1 \overset{?}{\neq} u_2 \end{tikzcd}$$

If $X_1, i_1 \overset{?}{\vdash} u_1;$ and $i_1 \leq i_2$.

$$\begin{tikzcd}[column sep=small] & \zeta, j \rhd_{G+\texttt{NoUse}} u;\ u \overset{?}{=} v \\ \underline{\textsc{Eq-left-right}} : \zeta, j \rhd_G v \arrow[ur] \arrow[dr] & \\ & \zeta, j \rhd_G v;\ u \overset{?}{\neq} v \end{tikzcd}$$

If $X, i \overset{?}{\vdash}_F u;$, $\texttt{NoUse} \notin G$ and $j > i$.

$$\begin{tikzcd}[column sep=small] & X_1, m \overset{?}{\vdash} u_1;\ \cdots ;\ X_n, m \overset{?}{\vdash} u_n; \\ \underline{\textsc{Ded-subterms}} : \zeta, i \rhd_F f(u_1, \ldots, u_n) \arrow[ur] \arrow[dr] & \zeta, i \rhd_{F+\texttt{NoCons}_f} u \\ & \zeta, i \rhd_{F+\texttt{NoCons}_f} f(u_1, \ldots, u_n) \end{tikzcd}$$

If $\texttt{NoCons}_f, \texttt{NoUse} \notin F$ and $X_1, \ldots, X_n$ are fresh variables.

All rules assume that the equations have a mgu and that this mgu is eagerly applied to the resulting constraint without yielding any trivial disequation.

Fig. 1. Transformation rules

We have to explain now what happens when, on a given pair (C, C') a rule can be applied on C and not on C' (or the converse).

Example 9. Let $C = (\phi, D, E, E')$ and $C' = (\phi, D', E, E')$ where $E = E' = \emptyset$, $\phi = ax_1, 1 \rhd a$, $D = X, 1 \overset{?}{\vdash} \mathsf{enc}(x_1, x_2)$, and $D' = X, 1 \overset{?}{\vdash} x$. The rule Cons can be applied on C and not on C'. However, we have to consider solutions where $\mathsf{enc}(x_1, x_2)\sigma$ and $x\sigma'$ are both obtained by a construction. Hence, it is important to enable this rule on both sides. For this, we first apply the substitution $x \mapsto \mathsf{enc}(y_1, y_2)$ where y_1, y_2 are fresh variables. This yields the two pairs of constraints (C_1, C_1') and (C_2, C_2') (forgetting about equations):

– $C_1 = (\phi, X_1, 1 \overset{?}{\vdash} x_1; X_2, 1 \overset{?}{\vdash} x_2)$ and $C_1' = (\phi, X_1, 1 \overset{?}{\vdash} y_1;\ X_2, 1 \overset{?}{\vdash} y_2)$;
– $C_2 = (\phi, X, 1 \overset{?}{\vdash}_{\mathtt{NoCons}_{\mathsf{enc}}} \mathsf{enc}(x_1, x_2))$ and $C_2' = (\phi, X, 1 \overset{?}{\vdash}_{\mathtt{NoCons}_{\mathsf{enc}}} x)$.

Therefore, the rule Cons, (this is similar for Ded-subterms), when applied to pairs of constraints comes in three versions: either the rule is applied on both sides or, if $X, i \overset{?}{\vdash} f(t_1, \ldots, t_n)$ (resp. $\zeta \rhd f(t_1, \ldots, t_n)$) is in C, and $X, i \overset{?}{\vdash} x$ (resp. $\zeta \rhd x$) is in C', we may apply the rule on the pair of constraints, adding to C' the equation $x \overset{?}{=} f(x_1, \ldots, x_n)$ where $x_1, \ldots, x_n$ are fresh variables. The third version is obtained by switching C and C'. This may introduce new variables, that yield a termination issue, which we discuss in Section 4.1. Similarly, the rules Axiom and Dest assume that $v \notin \mathcal{X}$. This has to be satisfied by C or C'. In case of the rule Dest, this means that the variables of the rewrite rule might not be immediately eliminated: this may also introduce new variables. For the rules Eq-left-left, Eq-right-right and Eq-left-right, we require that at least one new non-trivial equality (or disequality) is added to one of the two constraints (otherwise there is a trivial loop).

For all rules, if a rule is applicable on one constraint and not the other, we do perform the transformation, however replacing a constraint with $\perp$ when a condition becomes false or meaningless. Furthermore, we also replace a constraint C with $\perp$ when:

– the rule Dest cannot be applied on C; and
– C contains a constraint $X, i \overset{?}{\vdash} v$ such that v is not a variable and the rules Cons and Axiom cannot be applied to it.

Altogether this yields a transformation relation $(C, C') \to (C_1, C_1'), (C_2, C_2')$ on pairs of constraints: a node labeled (C, C') has two sons, respectively labeled (C_1, C_1') and (C_2, C_2').

Our algorithm can be stated as follows:

– Construct, from an initial pair of constraints (C_0, C_0') a tree, by applying as long as possible a transformation rule to a leaf of the tree.
– If, at some point, there is a leaf to which no rule is applicable and that is labeled $(C, \perp)$ or $(\perp, C)$ where $C \neq \perp$, then we stop with $C_0 \not\approx_s C_0'$.
– Otherwise, if the construction of the tree stops without reaching such a failure, return $C_0 \approx_s C_0'$.

Our algorithm can also be used to decide static equivalence of frames, as well as the (un)satisfiability of a constraint. Furthermore, in case of failure, a witness of the failure can be returned, using the equations of the non-$\perp$ constraint.

4 Correctness, Completeness and Termination

4.1 Termination

In general, the rules might not terminate, as shown by the following example:

Example 10. Consider the initial pair of contraints (C, C') given below:

$$C = \begin{cases} a \overset{?}{\vdash} \mathsf{enc}(x_1, x_2) \\ a, b \overset{?}{\vdash} x_1 \end{cases} \qquad C' = \begin{cases} a \overset{?}{\vdash} y_1 \\ a, b \overset{?}{\vdash} \mathsf{enc}(y_1, y_2) \end{cases}$$

We may indeed apply Cons yielding (on one branch):

$$C_1 = \begin{cases} a \overset{?}{\vdash} x_1 \\ a \overset{?}{\vdash} x_2 \\ a, b \overset{?}{\vdash} x_1 \end{cases} \qquad C'_1 = \begin{cases} a \overset{?}{\vdash} z_1 \\ a \overset{?}{\vdash} z_2 \qquad \text{and } y_1 \overset{?}{=} \mathsf{enc}(z_1, z_2) \\ a, b \overset{?}{\vdash} \mathsf{enc}(\mathsf{enc}(z_1, z_2), y_2) \end{cases}$$

Then, again using Cons, we get back as a subproblem the original constraints.

Fortunately, there is a simple complete strategy that avoids this behavior, by breaking the symmetry between the two constraints components. We assume in the following that, applying

- Cons to (C, C') where $X, i \overset{?}{\vdash} x \in C$ and $X, i \overset{?}{\vdash} f(t_1, \ldots, t_n) \in C'$,
- Ded-subterms to (C, C') where $\zeta, j \rhd x \in C$ and $\zeta, j \rhd f(t_1, \ldots, t_n) \in C'$,
- Dest to (C, C') where $X, i \overset{?}{\vdash} u; \zeta, j \rhd x \ \in C$ and $X, i \overset{?}{\vdash} u'; \zeta, j \rhd v' \ \in C'$

are only allowed when no other rule can be applied.

There is however no such restriction, when we switch the elements of the pair. If we come back to Example 10, we still apply the same transformation rule to the pair (C, C'), but we cannot apply Cons to (C_1, C'_1) since Eq-right-right can be applied to the constraint C_1, yielding a failure: $C \not\approx_s C'$.

Lemma 1. *With the above strategy, the transformation rules are terminating on any initial pair of constraint systems.*

Idea of the proof: as long as no new first-order variable is introduced, the set of first-order terms appearing in the constraint is roughly bounded by the subterms of the constraint. (This relies on the properties of the rewrite system). Loops are then prevented by the flags. Now, because of the eager application of substitutions, the only cases in which new first-order variables are introduced are the above cases of applications of Cons, Ded-subterms and Dest. Until new variables are introduced in the right constraints, the above argument applies:

the sequence of transformations is finite. Then, according to the strategy, when new variables are introduced on the right constraint, no other rule may apply. This implies that the left constraint (considered in isolation) is irreducible: it is of the form $X_1, i_1 \overset{?}{\vdash} x_1, \ldots, X_n, i_n \overset{?}{\vdash} x_n, \ldots$ where $x_1, \ldots x_n$ are distinct variables (which we call a *solved constraint*). From this point onwards, the rules Dest, Ded-subterms will never be applicable and therefore, no element will be added to the frames. Then, either usable elements of the frames are strictly decreasing (using a Eq-left-right) or else we preserve the property of being solved on the left. In the latter case, the first termination argument can be applied to the right constraint.

4.2 Correctness

The transformation rules yield a finite tree labeled with pairs of constraints.

Lemma 2. *If all leaves of a tree, whose root is labeled with (C_0, C_0') (a pair of initial constraints), are labeled either with $(\bot, \bot)$ or with some (C, C') with $C \neq \bot, C' \neq \bot$, then $C_0 \approx_s C_0'$.*

The idea of the proof is to first analyse the structure of the leaves. We introduce a restricted symbolic equivalence $\approx_s^r$ such that $C \approx_s^r C'$ for any leaf whose two label components are distinct from $\bot$. Roughly, this restricted equivalence will only consider the recipes that satisfied the additional constraints induced by the flags. Then we show that $\approx_s^r$ is preserved upwards in the tree: for any transformation rule, if the two pairs of constraints labeling the sons of a node are respectively in $\approx_s^r$, then the same property holds for the father. Finally, $\approx_s^r$ coincides with $\approx_s$ on the initial constraints (that contain no flag).

4.3 Completeness

We prove that the symbolic equivalence is preserved by the transformation rules, which yields:

Lemma 3. *If (C_0, C_0') is a pair of initial constraints such that $C_0 \approx_s C_0'$, then all leaves of a tree, whose root is labeled with (C_0, C_0'), are labeled either with $(\bot, \bot)$ or with some (C, C') with $C \neq \bot$ and $C' \neq \bot$.*

5 Implementation and Experiments

An Ocaml implementation of an early version of the procedure described in this paper, as well as several examples, are available at `http://www.lsv.ens-cachan.fr/~cheval/programs/index.php` (around 5000 lines of Ocaml). Our implementation closely follows the transformation rules that we described. For efficiency reasons, a strategy on checking the rules applicability has been designed in addition.

We checked the implementation on examples of static equivalence problems, on examples of satisfiability problems, and on symbolic equivalence problems that come from actual protocols. On all examples the tool terminates in less than a second (on a standard laptop). Note that the input of the algorithm is a pair of constraints: checking the equivalence of protocols would require in addition an interleaving step, that could be expensive.

We have run our tool on the following family of examples presented in [5]:

$$\phi_n = \{ax_1 \rhd t_n^0, ax_2 \rhd c_0, ax_3 \rhd c_1\} \text{ and } \phi'_n = \{ax_1 \rhd t_n^1, ax_2 \rhd c_0, ax_3 \rhd c_1\}$$

where $t_0^i = c_i$ and $t_{n+1}^i = \langle \mathsf{enc}(t_n^i, k_n^i), k_n^i \rangle$, $i \in \{0, 1\}$. In these examples, the size of the distinguishing tests increase exponentially while the sizes of the frames grow linearly. As KiSs [10], our tool outperforms YAPA [4] on such examples.

For symbolic equivalences, we cannot compare with other tools (there is no such tools); we simply tested the program on some home made benchmarks as well as on the handshake protocol, several versions of the encrypted password transmission protocol, the encrypted key exchange protocol [13], each for the offline guessing attack property. We checked also the strong secrecy for the corrected Dennin-Sacco key distribution protocol. Unfortunately we cannot (yet) check anonymity properties for e-voting protocols, as we would need to consider more cryptographic primitives.

6 Conclusion

We presented a new algorithm for deciding symbolic equivalence, which performs well in practice. There is still some work to do for extending the results and the tool. First, we use trace equivalence, which requires to consider all interleavings of actions; for each such interleaving, a pair of constraints is generated, which is given to our algorithm. This requires an expensive overhead (which is not implemented), that might be unnecessary. Instead, we wish to extend our algorithm, considering pairs of sets of constraints and use a symbolic bisimulation. This looks feasible and would avoid the detour through trace equivalence. This would also allow drop the determinacy assumption on the protocols and to compare our method with ProVerif [7].

We considered only positive protocols; we wish to extend the algorithm to non-positive protocols, allowing disequality constraints from the start. Finally, we need to extend the method to other cryptographic primitives, typically blind signatures and zero-knowledge proofs.

Acknowledgments. We wish to thank Sergiu Bursuc for fruitful discussions.

References

1. Abadi, M., Cortier, V.: Deciding knowledge in security protocols under equational theories. Theoretical Computer Science 367(1-2), 2–32 (2006)
2. Abadi, M., Fournet, C.: Mobile values, new names, and secure communication. In: Proc. of 28th ACM Symposium on Principles of Programming Languages, POPL'01 (2001)

3. Baudet, M.: Deciding security of protocols against off-line guessing attacks. In: Proc. of 12th ACM Conference on Computer and Communications Security (2005)
4. Baudet, M.: YAPA, Yet Another Protocol Analyzer (2008), `http://www.lsv.ens-cachan.fr/~baudet/yapa/index.html`
5. Baudet, M., Cortier, V., Delaune, S.: YAPA: A generic tool for computing intruder knowledge. In: Treinen, R. (ed.) RTA 2009. LNCS, vol. 5595, pp. 148–163. Springer, Heidelberg (2009)
6. Blanchet, B.: An automatic security protocol verifier based on resolution theorem proving (invited tutorial). In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, Springer, Heidelberg (2005)
7. Blanchet, B., Abadi, M., Fournet, C.: Automated verification of selected equivalences for security protocols. Journal of Logic and Algebraic Programming 75(1), 3–51 (2008)
8. Cheval, V., Comon-Lundh, H., Delaune, S.: Automating security analysis: symbolic equivalence of constraint systems. Technical report (2010), `http://www.lsv.ens-cachan.fr/~cheval/programs/technical-report.pdf`
9. Chevalier, Y., Rusinowitch, M.: Decidability of symbolic equivalence of derivations (unpublished draft) (2009)
10. Ciobâcă, Ş.: Kiss (2009), `http://www.lsv.ens-cachan.fr/~ciobaca/kiss`
11. Comon-Lundh, H.: Challenges in the automated verification of security protocols. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 396–409. Springer, Heidelberg (2008)
12. Comon-Lundh, H., Cortier, V., Zalinescu, E.: Deciding security properties of cryptographic protocols. application to key cycles. Transaction on Computational Logic 11(2) (2010)
13. Corin, R., Doumen, J., Etalle, S.: Analysing password protocol security against off-line dictionary attacks. Electr. Notes Theor. Comput. Sci. 121, 47–63 (2005)
14. Cortier, V., Delaune, S.: A method for proving observational equivalence. In: Proc. of 22nd Computer Security Foundations Symposium (CSF'09), pp. 266–276. IEEE Comp. Soc. Press, Los Alamitos (2009)
15. Delaune, S., Kremer, S., Ryan, M.D.: Verifying privacy-type properties of electronic voting protocols. Journal of Computer Security 17(4), 435–487 (2009)
16. Fendrup, U., Hüttel, H., Jensen, J.N.: Modal logics for cryptographic processes. Theoretical Computer Science 68 (2002)
17. Huttel, H.: Deciding framed bisimulation. In: 4th International Workshop on Verification of Infinite State Systems, INFINITY'02, pp. 1–20 (2002)
18. Meadows, C.: The NRL protocol analyzer: An overview. Journal of Logic Programming 26(2), 113–131 (1996)
19. Millen, J., Shmatikov, V.: Constraint solving for bounded-process cryptographic protocol analysis. In: Proc. of 8th ACM Conference on Computer and Communications Security (2001)
20. Rusinowitch, M., Turuani, M.: Protocol insecurity with finite number of sessions is np-complete. In: Proc. of 14th Computer Security Foundations Workshop (2001)
21. Weidenbach, C.: Towards an automatic analysis of security protocols in first-order logic. In: Ganzinger, H. (ed.) CADE 1999. LNCS (LNAI), vol. 1632, pp. 314–328. Springer, Heidelberg (1999)

System Description: The Proof Transformation System CERES[⋆]

Tsvetan Dunchev, Alexander Leitsch, Tomer Libal, Daniel Weller, and Bruno Woltzenlogel Paleo

Institute of Computer Languages (E185), Vienna University of Technology, Favoritenstraße 9, 1040 Vienna, Austria
{cdunchev,leitsch,shaolin,weller,bruno}@logic.at

Abstract. Cut-elimination is the most prominent form of proof transformation in logic. The elimination of cuts in formal proofs corresponds to the removal of intermediate statements (lemmas) in mathematical proofs. The cut-elimination method CERES (cut-elimination by resolution) works by extracting a set of clauses from a proof with cuts. Any resolution refutation of this set then serves as a skeleton of an ACNF, an LK-proof with only atomic cuts.

The system CERES, an implementation of the CERES-method has been used successfully in analyzing nontrivial mathematical proofs (see [4]).In this paper we describe the main features of the CERES system with special emphasis on the extraction of Herbrand sequents and simplification methods on these sequents. We demonstrate the Herbrand sequent extraction and simplification by a mathematical example.

1 Introduction

Proof analysis is a central mathematical activity which proved crucial to the development of mathematics. Indeed many mathematical concepts such as the notion of group or the notion of probability were introduced by analyzing existing arguments. In some sense the analysis and synthesis of proofs form the very core of mathematical progress.

Cut-elimination introduced by Gentzen [8] is the most prominent form of proof transformation in logic and plays a key role in automatizing the analysis of mathematical proofs. The removal of cuts corresponds to the elimination of intermediate statements (lemmas) from proofs resulting in a proof which is analytic in the sense that all statements in the proof are subformulas of the result. Therefore, the proof of a combinatorial statement is converted into a purely combinatorial proof.

The development of the method CERES (cut-elimination by resolution) was inspired by the idea to fully automate cut-elimination on real mathematical proofs, with the aim of obtaining new interesting elementary proofs. While a

⋆ Supported by the Austrian Science Fund (project no. P22028-N13).

fully automated treatment proved successful for mathematical proofs of moderate complexity (e.g. the "tape proof" [2] and the "lattice proof" [9]), more complex mathematical proofs required an interactive use of CERES; this way we successfully analyzed Fürstenberg's proof of the infinitude of primes (see [4]) and obtained Euclid's argument of prime construction. Even in its interactive use CERES proved to be superior to the reductive cut-elimination due to additional structural information provided by the characteristic clause set (see below).

CERES [5,6] is a cut-elimination method that is based on resolution. The method roughly works as follows: From the input proof φ of a sequent S a clause term is extracted and evaluated to an unsatisfiable set of clauses $\mathrm{CL}(\varphi)$, the *characteristic clause set*. A resolution refutation γ of $\mathrm{CL}(\varphi)$, which is obtained using a first-order theorem prover, serves as a skeleton for an (atomic cut normal form) ACNF ψ, a proof of S which contains at most atomic cuts. This method of cut-elimination has been implemented in the system `CERES`[1]. The system is capable of dealing with formal proofs in an extended version **LKDe** of **LK**, among them also very large ones.

However, the large size of ACNFs, automatically generated by `CERES`, turned out problematic in practice. Indeed, the aim is not only to produce an ACNF ψ from φ, but also to interpret ψ as a *mathematical* proof. In fact, the huge sizes of output proofs result from an inherent redundancy of formal calculi. Less redundant representations of the underlying mathematical arguments can be obtained by extracting a *Herbrand sequent* $H(\psi)$ from an ACNF ψ (see [9] and [11]). Thereby, $H(\psi)$ is a sequent consisting of instances of the quantifier-free parts of the formulas in S (we assume that S is skolemized). Though Herbrand sequents proved clearly superior to ACNFs in the analysis by humans, further simplifications of these sequents turned out important in practice. In this system description we lay specific emphasis on the extraction and simplification of Herbrand sequents, and illustrate transformations by an example.

By its high efficiency (the core of the method is first-order theorem proving by resolution and paramodulation), and by automatically extracting crucial structural information from proofs (e.g. the characteristic clause set) `CERES` proved useful in *automated proof mining*, thus contributing to an experimental culture of *computer-aided proof analysis* in mathematics.

2 The System `CERES`

The cut-elimination system `CERES` is written in ANSI-C++. The core functionality of `CERES`[3] allows the user to input a first order proof φ and obtain an ACNF(φ). The core system also includes two additional tools: the compiler `hlk`[2] for the intermediary proof language HandyLK and the proof viewer ProofTool[3]. The system follows a uniform data representation model for proofs and sequents in the form of XML using the proofdatabase DTD[4].

[1] Available at `http://www.logic.at/ceres/`

[2] `http://www.logic.at/hlk`

[3] `http://www.logic.at/prooftool`

[4] `http://www.logic.at/ceres/xml/5.0/proofdatabase.dtd`

This functionality is extended in the current system by various optimizations on the resulted proof. CERES allows the computation of a Herbrand-sequent of the theorem and applies to it several simplification algorithms. Due to the important role of resolution provers in CERES, the system interfaces now with two additional provers: Prover9[5] and ATP[6].

The execution cycle starts with the mathematician using HandyLK to produce a formal **LKDe**-proof φ. **LKDe** is an extension of **LK**[2] to include definition and equality rules. HandyLK produces a formal proof by focusing on essential information as input. In particular, propositional inferences and context formulas are not required. HandyLK also simplifies the writing of proofs in a tree form by supplying meta-variables denoting proofs. Finally, HandyLK enables the definition of proof schemata by supporting parameterization over meta-terms and meta-formulas.

Since the restriction to skolemized proofs is crucial to the CERES-method, the system includes a proof skolemization transformation sk following Andrew's method[1]. sk(φ) is then parsed by CERES to produce CL(φ).

CL(φ) is given as input to one of the three possible theorem provers (Otter, Prover9 and ATP) and a resolution refutation is extracted. Otter and its successor Prover9 are both very efficient resolution provers that produce the resolution tree as output. Because a fully automatic refutation is not always possible, the interactive prover ATP was developed. ATP is a basic resolution prover that supports interaction with the user as well as customizable refinements. Interaction was used in order to validate and complete (manually obtained) refutations while customization can be used in order to implement specific and more efficient refinements[12].

In the last phase, CERES maps the refutation into an **LKDe**-proof such that resolution steps are mapped into atomic cuts, paramodulation into equality rules, etc. Moreover, it inserts proof projections (which are cut-free parts of the input proof[5,6]) in order to obtain an ACNF(φ).

For a convenient analysis of the results, the system is equiped with the proof viewer/editor ProofTool. ProofTool is capable of presenting all data objects used in the process: the original and skolemized proof, the profile, the refutation, the ACNF-proof and the simplified Herbrand-sequent.

Herbrand-sequent extraction and simplification. As a post-process, the system now supports the extraction of a Herbrand-sequent $H(\varphi)$ from ACNF(φ). This process transforms the ACNF(φ) into a proof in an intermediary calculus $\mathbf{LKe}_A$, in which weakly quantified formulas are being replaced by an array of their instances. By "inverting" the transformation, we obtain $H(\varphi)$, which is still valid and contains all the desired information in a more compact form. $H(\varphi)$ is simplified further by the application of three algorithms[7].

The first simplification algorithm S_{use} removes formulas containing irrelevant information: formulas which were introduced either by weakening or as the side-formulas of the main formulas of some other inferences.

[5] http://www.cs.unm.edu/~mccune/prover9/

[6] http://www.logic.at/atp

An algebraic simplification S_{alg} is performed on the resulted sequent by normalizing each term with regard to a user-defined set of rewriting rules.

The last simplification algorithm S_{log} strips logically irrelevant formulas. As a Herbrand-sequent is always valid with regard to a given theory, we negate $H(\varphi)$ and apply a resolution theorem prover in order to obtain a refutation γ of the background theory and $\neg H(\varphi)$. The logically irrelevant formulas are all those formulas not appearing as leaves in γ.

3 An Example

In this section, we will treat a simple example from lattice theory. There are several different, but equivalent, definitions of the notion of *lattice*. Usually, the equivalence of several statements is shown by proving a cycle of implications. While this approach is elegant, it has the drawback that it does not provide *direct* proofs between the statements. Using cut-elimination, direct proofs of the implications between any two of the statements can be obtained. Hence we will demonstrate how the `CERES` system can be used to automatically generate such a proof via cut-elimination, how the Herbrand sequent extracted from the resulting proof can be simplified, and how the simplified Herbrand sequent provides a minimal explicit construction which was implicit in the original proof.

Lattice definitions. We will consider three definitions of the notion of lattice: two are algebraic, using 3-tuples $\langle L, \cap, \cup \rangle$, while the third one depends on the notion of partially ordered set $\langle S, \leq \rangle$.

Definition 1 (Algebraic Lattices). *A* semi-lattice *is a set L together with an operation $\circ$ fulfilling for all $x, y, z \in L$*

$$x \circ y = y \circ x \quad \text{and} \quad x \circ x = x \quad \text{and} \quad (x \circ y) \circ z = x \circ (y \circ z).$$

A $L1$-lattice is a set L together with operations $\cap$ (meet) *and $\cup$* (join) *s.t. both $\langle L, \cap \rangle$ and $\langle L, \cup \rangle$ are semi-lattices and for all $x, y \in L$*

$$x \cap y = x \leftrightarrow x \cup y = y.$$

A $L2$-lattice is a set L together with operations $\cap$ and $\cup$ s.t. both $\langle L, \cap \rangle$ and $\langle L, \cup \rangle$ are semi-lattices which for all $x, y \in L$ obey the absorption laws

$$(x \cap y) \cup x = x \quad \text{and} \quad (x \cup y) \cap x = x$$

Definition 2 (Partial Order). *A binary relation $\leq$ on a set S is called* partial order *if for all $x, y, z \in S$*

$$x \leq x \quad \text{and} \quad (x \leq y \wedge y \leq x) \rightarrow x = y \quad \text{and} \quad (x \leq y \wedge y \leq z) \rightarrow x \leq z.$$

Definition 3 (Lattices Based on Partial Orders). *A $L3$-lattice is a partially ordered set $\langle S, \leq \rangle$ s.t. for all $x, y \in S$ there exist a greatest lower bound $\cap$ and a least upper bound $\cup$, i.e. for all $z \in S$*

$$\begin{gathered} x \cap y \leq x \wedge x \cap y \leq y \wedge (z \leq x \wedge z \leq y \rightarrow z \leq x \cap y) \text{ and} \\ x \leq x \cup y \wedge y \leq x \cup y \wedge (x \leq z \wedge y \leq z \rightarrow x \cup y \leq z). \end{gathered}$$

It is well known that the above three definitions of lattice are equivalent. We will formalize the proofs of $L1 \rightarrow L3$ and $L3 \rightarrow L2$ in order to extract a direct proof of $L1 \rightarrow L2$, i.e. one which does not use the notion of partial order.

Formalization of the Lattice Proof. The full **LKDe**-proof of $L1 \rightarrow L2$, formalized in the HandyLK language and compiled to **LKDe** by `hlk`, has 260 rules (214 rules, if structural rules, except cut, are not counted). It is too large to be displayed here. Below we show only a part of it, which is close to the end-sequent and depicts the main structure of the proof, based on the cut-rule with $L3$ as the cut-formula. The full proofs, conveniently viewable with `ProofTool`, are available on the website of `CERES`.

We note here that the proof is formalized in the theory of semi-lattices: it uses (instances of the open versions of) the semi-lattice axioms, and hence the theorem is valid in the theory $\mathcal{T}$ of semi-lattices, but not in general.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{c}[p_R]\\ \vdash R\end{array} \quad \dfrac{\begin{array}{c}[p_{AS}]\\ \vdash AS\end{array} \quad \begin{array}{c}[p_T]\\ \vdash T\end{array}}{\vdash AS \wedge T}\,\wedge : r}{\vdash R \wedge (AS \wedge T)}\,\wedge : r}{\vdash POSET}\,d : r \qquad \dfrac{[p_{GLB}] \quad [p_{LUB}]}{L1 \vdash GLB \wedge LUB}\,\wedge : r}{L1 \vdash POSET \wedge (GLB \wedge LUB)}\,\wedge : r}{L1 \vdash L3}\,d : r \qquad \begin{array}{c}[p_3^2]\\ L3 \vdash L2\end{array}}{L1 \vdash L2}\,cut$$

- $L1 \equiv \forall x \forall y((x \cap y) = x \rightarrow (x \cup y) = y) \wedge ((x \cup y) = y \rightarrow (x \cap y) = x)$
- $L2 \equiv \forall x \forall y(x \cap y) \cup x = x \wedge \forall x \forall y(x \cup y) \cap x = x$
- $L3 \equiv POSET \wedge (GLB \wedge LUB)$
- p_{AS}, p_T, p_R are proofs of, respectively, anti-symmetry, transitivity and reflexivity of $\leq$, which is defined as $x \leq y \equiv x \cap y = x$.
- p_{GLB} and p_{LUB} are proofs that $\cap$ and $\cup$ are greatest lower bound and greatest upper bound, respectively.
- p_3^2 is a proof that $L3$-lattices are $L2$-lattices.

Cut-Elimination of the Lattice Proof. Prior to cut-elimination, the formalized proof is skolemized by `CERES`, resulting in a proof of the skolemized end-sequent $L1 \vdash (s_1 \cap s_2) \cup s_1 = s_1 \wedge (s_3 \cup s_4) \cap s_3 = s_3$, where s_1, s_2, s_3 and s_4 are skolem constants for the strongly quantified variables of $L2$. Then `CERES` eliminates cuts (using Prover9 for computing the refutation), producing a proof in ACNF (available for visualization with `ProofTool` in the website of `CERES`).

Herbrand Sequent Extraction of the ACNF of the Lattice Proof. As our proof under investigation uses axioms of the theory $\mathcal{T}$, also our ACNF φ is a proof in $\mathcal{T}$, and the Herbrand sequent $H(\varphi)$, extracted from φ according to the algorithm from [9], is valid in $\mathcal{T}$. After the application of S_{use}, $H(\varphi)$ becomes the sequent H':

$$\begin{aligned}
&s_1 \cup (s_1 \cup (s_1 \cap s_2)) = s_1 \cup (s_1 \cap s_2) \rightarrow s_1 \cap (s_1 \cup (s_1 \cap s_2)) = s_1,\\
&s_1 \cap s_1 = s_1 \rightarrow s_1 \cup s_1 = s_1,\\
&(s_1 \cap s_2) \cap s_1 = s_1 \cap s_2 \rightarrow (s_1 \cap s_2) \cup s_1 = s_1,\\
&(s_1 \cup (s_1 \cap s_2)) \cup s_1 = s_1 \rightarrow (s_1 \cup (s_1 \cap s_2)) \cap s_1 = s_1 \cup (s_1 \cap s_2),\\
&(s_3 \cup (s_3 \cup s_4) = s_3 \cup s_4 \rightarrow s_3 \cap (s_3 \cup s_4) = s_3\\
&\vdash L2, (s_1 \cap s_2) \cup s_1 = s_1 \wedge (s_3 \cup s_4) \cap s_3 = s_3
\end{aligned}$$

Observe that S_{use} has pruned some subformulas from $H(\varphi)$: as $H(\varphi)$ is a Herbrand sequent of our theorem, its antecedent only contains instances of $L1$. Observe that the formulas in the antecedent of H' are not instances of $L1$ (some conjuncts where deleted), but still H' is valid in $\mathcal{T}$ and contains the relevant information from the ACNF.

By S_{log}, H' is further pruned and four formulas are deleted, finally resulting in the Herbrand sequent H''

$$\begin{aligned}
&(s_1 \cap s_2) \cap s_1 = s_1 \cap s_2 \rightarrow (s_1 \cap s_2) \cup s_1 = s_1,\\
&s_3 \cup (s_3 \cup s_4) = s_3 \cup s_4 \rightarrow s_3 \cap (s_3 \cup s_4) = s_3\\
&\vdash (s_1 \cap s_2) \cup s_1 = s_1 \wedge (s_3 \cup s_4) \cap s_3 = s_3
\end{aligned}$$

H'' is minimal in the sense that if we remove a formula from H'', the resulting sequent is not valid in $\mathcal{T}$ anymore. This is not the case in general; minimality is determined by the resolution refutation computed in S_{log}.

H'' now gives rise to an elementary proof of the theorem $L1 \vdash L2$: Our goal is to prove (1) $(s_1 \cap s_2) \cup s_1 = s_1$ and (2) $(s_3 \cup s_4) \cap s_3 = s_3$. For (1), we prove $(s_1 \cap s_2) \cap s_1 = s_1 \cap s_2$ using idempotency, associativity and commutativity of $\cap$ and conclude with $L1$. For (2), we prove $s_3 \cup (s_3 \cup s_4) = s_3 \cup s_4$ using idempotency and associativity of $\cup$. We conclude with $L1$ and commutativity of $\cap$.

Summarizing, the `CERES` system has taken as input a proof in lattice theory which used the auxiliary notion of partial order. By cut-elimination, a new proof not using any auxiliary notions is computed. From this proof, a Herbrand sequent summarizing the mathematical information (i.e. the instantiations) is extracted. This sequent is further pruned, resulting in a compact presentation of the relevant mathematical ideas of the proof (in this case, an algebraic construction not visible in the input proof). In the present example, the algorithm S_{alg} was not used: we refer to [7] for further examples.

4 Summary of Recent Improvements and Future Work

The simplification of Herbrand sequents is one of the most important features recently added to the `CERES`. Redundancy in the resulting sequent is significantly reduced and the terms are rewritten to a more readable normal-form. However, even if the simplified Herbrand sequent is completely redundancy-free, it can still be large and, consequently, the user can still face difficulties to formulate the informal mathematical proof that it summarizes. Our experience [9] indicates that enriching the Herbrand sequent with certain kinds of links between its atomic sub-formulas might provide helpful information to the user. These links would resemble the axiom links of proof nets or the connections of the connection method, and they could be obtained by analyzing either the axioms of the ACNF or the resolved literals in the refutation used in the simplifcation of the Herbrand sequent. The theoretical investigation and the implementation of such links remains for future work.

As mentioned in Section 2, CERES relies on resolution theorem provers to refute characteristic clause sets or profiles. While only Otter was originally supported, now it is also possible to use CERES together with Prover9 and ATP. For the future, we intend to support the TPTP/TSTP format. It is also desirable to interface CERES with proof assistants like Isabelle, Coq, PVS and Mizar. On the one hand, CERES would benefit from the large libraries of proofs written in the languages of these systems, and on the other hand, these systems would benefit from the proof transformations of CERES. The main obstacle has always been the differences in the logical frameworks used by each of these systems and by CERES.

Our most ambitious current goal is an extension of CERES to higher-order logic [10]. This task encountered a few hard theoretical obstacles, such as the difficulty of skolemizing higher-order proofs [10], as well as practical obstacles, such as the need to change the core data-structures of CERES in order to support higher-order formulas. This led to the decision of implementing a new version of CERES, currently under development in the more flexible language Scala.

References

1. Andrews, P.B.: Resolution in Type Theory. J. of Symbolic Logic 36, 414–432 (1971)
2. Baaz, M., Hetzl, S., Leitsch, A., Richter, C., Spohr, H.: Proof Transformation by CERES. In: Borwein, J.M., Farmer, W.M. (eds.) MKM 2006. LNCS (LNAI), vol. 4108, pp. 82–93. Springer, Heidelberg (2006)
3. Baaz, M., Hetzl, S., Leitsch, A., Richter, C., Spohr, H.: System Description: The Cut-Elimination System CERES. In: Proc. ESCoR 2006, pp. 159–167 (2006)
4. Baaz, M., Hetzl, S., Leitsch, A., Richter, C., Spohr, H.: CERES: An analysis of Fürstenberg's proof of the infinity of primes. Th. Co. Sci. 403, 160–175 (2008)
5. Baaz, M., Leitsch, A.: Cut-Elimination and Redundancy-Elimination by Resolution. Journal of Symbolic Computation 29, 149–176 (2000)
6. Baaz, M., Leitsch, A.: Towards a Clausal Analysis of Cut-Elimination. Journal of Symbolic Computation 41, 381–410 (2006)
7. Dunchev, T.: Simplification of Herbrand Sequents. Master Thesis (2009)
8. Gentzen, G.: Untersuchungen über das logische Schliessen. Mathematische Zeitschrift 39, 176–210, 405–431 (1934-1935)
9. Hetzl, S., Leitsch, A., Weller, D., Woltzenlogel Paleo, B.: Herbrand sequent extraction. In: Autexier, S., Campbell, J., Rubio, J., Sorge, V., Suzuki, M., Wiedijk, F. (eds.) AISC 2008, Calculemus 2008, and MKM 2008. LNCS (LNAI), vol. 5144, pp. 462–477. Springer, Heidelberg (2008)
10. Hetzl, S., Leitsch, A., Weller, D., Woltzenlogel Paleo, B.: A Clausal Approach to Proof Analysis in Second-Order Logic. In: Logical Foundations of Computer Sci. (2009)
11. Woltzenlogel Paleo, B.: Herbrand Sequent Extraction. VDM-Verlag (2008)
12. Woltzenlogel Paleo, B.: A General Analysis of Cut-Elimination by CERes. PhD Thesis (2009)

Premise Selection in the Naproche System

Marcos Cramer, Peter Koepke, Daniel Kühlwein, and Bernhard Schröder

Mathematical Institute, University of Bonn
German Linguistics, University of Duisburg-Essen
{cramer,koepke,kuehlwei}@math.uni-bonn.de,
bernhard.schroeder@uni-due.de
http://www.naproche.net

Abstract. Automated theorem provers (ATPs) struggle to solve problems with large sets of possibly superfluous axiom. Several algorithms have been developed to reduce the number of axioms, optimally only selecting the necessary axioms. However, most of these algorithms consider only single problems. In this paper, we describe an axiom selection method for series of related problems that is based on logical and textual proximity and tries to mimic a human way of understanding mathematical texts. We present first results that indicate that this approach is indeed useful.

Keywords: formal mathematics, automated theorem proving, axiom selection.

1 Introduction

Reducing the search space of ATP problems is a long standing problem in the ATP community. In 1987 Larry Wos called a solution to the problem of definition expansion and contraction "one of the more significant advances in the field of automated reasoning" [22]. In recent years, several algorithms have been developed to tackle this problem. (e.g. SRASS [17], SInE [8], Gazing [1], MaLARea [18], and the work by Meng and Paulson [13]). In this paper, we describe an axiom selection method for series of related problems that is based on logical and textual proximity.

The Naproche project (NAtural language PROof CHEcking) studies the semi-formal language of mathematics as used in mathematical journals and textbooks from the perspectives of linguistics, logic and mathematics. As part of the Naproche project, we develop the Naproche system [5], a program that can automatically check texts written in the Naproche controlled natural language (CNL) for logical correctness. We test our system by reformulating parts of mathematical textbooks and the basics of mathematical theories in the Naproche CNL and checking the resulting texts.

The checking process is similar to how a human reader would verify the correctness of a text. Each statement in the text that is not an axiom[1], a definition

[1] Here, axiom is used in the mathematical sense, e.g. the axiom of choice.

or an assumption must follow from the information given so far. Using logical terms, we can say that the statement has to follow from its premises. In the Naproche system, such statements create proof obligations. A proof obligations is an ATP problem with the statement as conjecture and the premises as axioms.

The longer a text is, the more premises are available, which makes proof obligations harder to discharge. Thus, we need to find a way to reduce the number of premises, i.e. reduce the search space of the ATP. For single ATP problems, successful algorithms (e.g. SInE, SRASS) exist. There are also programs that were developed for larger theories, e.g. Gazing and MaLARea. However, to our knowledge there is no system that is based on a 'human' understanding of a proof. The texts we are dealing with read like normal mathematical proofs in natural language. We developed a premise selection algorithm that tries to use some of the information implicit in the human structuring of the proof text.

We first give a quick overview of the Naproche system. Section 2 explains our premise selection algorithm. First results are presented in section 3.

2 The Naproche System

The Naproche system [5] checks texts that are written in a controlled natural language for mathematics for correctness. We call this controlled natural language the Naproche CNL. Texts written in the Naproche CNL read like normal mathematical texts. The Naproche CNL is described in a separate paper (see [3]). A quick overview can be found online[2].

The input text is first translated into a linguistic representation called Proof Representation Structure (PRS, see [3], [4]). From such a PRS the program determines which statements have to be checked and creates the corresponding proof obligations [10]. For the actual proving we use the TPTP infrastructure [16]. The proof obligations are translated into ATP problems in the TPTP format and then sent to an ATP.

There are two main long term goals of the Naproche Project: Firstly, to provide a more natural system for formalising mathematics, and secondly to function as a tool that can help undergraduate students to learn how to write formally correct proofs and thus get used to the semi-formal language of mathematics.

2.1 An Example Text

We present a short example texts taken from the Naproche translation of Euclid's Elements [7]. Note that Naproche uses LaTeX-sourcecode as input. The example shows the compiled version.

Example: Let a, b and c be distinct points. By Theorem 1 there is a point d, such that $\overline{da} = \overline{db} = \overline{ab}$. Let M be the line such that b and d are on M. Let α be the circle such that b is the center of α, and c is on α.

[2] `http://www.naproche.net/wiki/doku.php?id=dokumentation:language`

2.2 Related Work

There are several projects that are similar to Naproche. We will just name a few:

A. Trybulec's Mizar [12] is arguably the most prominent. It was started in 1973, and by today many non-trivial mathematical theorems have been proved. An active community continues to formulate and prove theorems in Mizar. The results are published regularly in the journal *Formalized Mathematics.*

The Isabelle [14] team is working on Isar [21], a "human-readable structured proof language". The System for Automated Deduction (SAD, [19]) checks texts that are written in its input language, ForThel [20], for correctness.

Claus Zinn did his PhD on *Understanding Informal Mathematical Discourse* [23], but focused on only two examples. The DIALOG group [2] did experiments with mathematical language in a tutoring context. Mohan Ganesalingam [6] studied the language of mathematics in detail, but did not implement his ideas (yet).

What distinguishes Naproche is our focus on deep linguistic analysis of non-annotated natural language. We try to keep our input language as close as possible to the natural language of mathematics.

3 The Premise Selection Algorithm

When verifying the correctness of a proof, mathematicians basically face the same problem as ATPs. They have a given set of premises, i.e. all their mathematical knowledge, and have to derive the conjecture from these premises. Understanding the proof means knowing which premises were used in each step. While the human selection process as a whole is very complicated, there are three parts that can easily be used for automated premise selection.

– Explicit References:
  Explicit references like "*by theorem 4*" are often used in mathematical texts. Such a reference is a clear indication that the referenced object is useful, or even necessary.
– Textual Adjacency:
  While human proofs are not as detailed as formal derivations, they are still done step by step. Usually, the proof steps just before a statement are relevant. The most common example of this are assumptions: *Assume* φ. *Then* ψ. Here, the proof step before *Then* ψ is *Assume* φ, and φ will most likely be needed to prove ψ.
– Logical Relevance:
  Quite often, ideas that were needed in one part of a proof are also needed in another part. I.e. if a definition was needed for the first proof step, it will probably be needed again later in the proof.

In order to capture these ideas we developed Proof Graphs. Each statement of the proof becomes a node in this graph. Two nodes are connected by an (untyped) edge if they are textually or logically close to each other, or if there is an explicit reference from one to the other. We define the distance between two statements as the geodesic distance. i.e. the length of the shortest path from one statement to the other.

Based on Proof Graphs, we can define a premise selection algorithm. Given a proof obligation, the premise selection algorithm determines which of the available premises are given to the ATP. The algorithm was implemented as part of the Naproche system.

Explicit references and textual adjacency are calculated during the linguistic analysis of the text. We say that φ is logically close to ψ if φ was used in the proof on ψ. In the implementation, we use GEOFF SUTCLIFFEs program *Proof Summary* which analyses ATP proofs.

The premises selection algorithm proceeds as follows:

– Input: *Conjecture*, *Axioms*, *Distance* and *Time*
1 Determine the distance between the *Conjecture* and the *Axioms*.
2 Select all axioms whose nodes have distance less than *Distance* from the conjecture Node.
3 Create a TPTP problem with the selected axioms and the *Conjecture*.
4 Run an ATP on the problem with time limit *Time*.
5 If the ATP cannot prove the conjecture from the axioms, the starting distance is less than the predefined maximum distance, and the starting time is less than the predefined maximum time, define a new time limit and a new starting distance (e.g. $\mathit{NewTime} = 2 * \mathit{Time}$ and $\mathit{NewDistance} = 2 * \mathit{Distance}$) and try again.
6 If the ATP finds a proof, use the proof given by the ATP to find out which axioms where actually used. Determine the maximum distance of the used axiom to define the new starting distance (e.g. $\mathit{NextDistance} = (4 * \mathit{Distance} + \mathit{MaxUsedDistance})/5$) and update the proof graph with this new information.

4 Results

To test the algorithm we checked a Naproche CNL version of the first chapter of LANDAU's *Grundlagen der Analysis* [11] with and without the premises selection algorithm. This text contains 228 proof obligation with a total number of 7602 premises.

Currently *Proof Summary* [16] only supports two ATPs, Metis [9] and EP [15]. Both were used during testing. MaxDistance was set to 20, MaxTime was set to 5 seconds, the start Distance was set to 1, and the start Time set to 1 sec. The other values were defined as follows:

$$\begin{aligned} \text{NewTime} &= 2 * \text{Time} \\ \text{NewDistance} &= 2 * \text{Distance} \\ \text{NextDistance} &= \left\lceil \frac{4 * \text{Distance} + \text{MaxUsedDistance}}{5} \right\rceil \end{aligned}$$

Table 1 shows the results for EP. Without the premises selection algorithm, seven obligations could not be discharged by EP. With the premise selection algorithm enabled, EP was able to discharge all 228 obligations.

Table 1. Results for EP 1.0

		Without PS		With PS	
	Total	Theorem	No Proof	Theorem	No Proof
obligations	228	221	7	228	0
premises	7602	7235	367	3964	0
premises/obligations		32.74	52.43	17.39	N/A
used premises/obligation		2.99	N/A	2.93	N/A
unused premises/obligation		29.75	52.43	5.96	N/A
average distance		8.2	8.15	5.53	N/A
average used distance		3.46	N/A	3.38	N/A
average unused distance		8.68	8.15	5.96	N/A

We also determined the average distance of the used (3.46) and unused premises (8.68). These numbers are a clear indicator that our distance definition is indeed useful.

In Table 2 we see the results for Metis. Without the premises selection algorithm, 44 obligations could not be discharged by Metis. With the premise selection algorithm enabled, 26 obligations could not be discharged.

Table 2. Results for Metis 2.2

		Without PS		With PS	
	Total	Theorem	No Proof	Theorem	No Proof
obligations	228	184	44	202	26
premises	7602	5630	1972	2412	1176
premises/obligations		30.6	44.82	11.94	45.23
used premises/obligation		2.16	N/A	2.09	N/A
unused premises/obligation		28.43	44.42	9.85	45.23
average distance		8.98	9.64	5.22	9.49
average used distance		3.49	N/A	3.22	N/A
average unused distance		9.39	9.64	5.64	9.49

The reason why Metis ends up with a lower number of total premises with premises selection enabled is that the fewer obligations an ATP is able to discharge, the less information we have about logical relevance. This affects the subsequent obligations since fewer formulas are within the search distance. Similar to the EP results, the average distance of used premises is much lower (3.49) than the average distance of unused premises (9.39).

For further testing, we created two problem batches. The first one contained the original 228 problems. For the second batch we took all the modified problems that were created when using EP 1.0 and the premises selection algorithm. We sent the problems to Geoff Sutcliffe and he used his TPTP infrastructure to run seven ATP systems on the problems with a time limit of 300 seconds per obligation. The results can be seen in Table 3.

Table 3. Results with more ATPs and time limit 300 sec

ATP	Solved(Modified)	Solved(Original)
Bliksem 1.12	225	222
E 1.1	228	228
Geo 2007f	219	210
iProver 0.7	223	222
Metis 2.2	205	193
Prover9 0908	213	221
Vampire 11.0	227	227

182 of the modified problems were solved by all the systems. Of the remaining 46, 39 were solved by 6 out of the 7 systems. 169 of the original problems were solved by all the systems. Of the remaining 59, 50 were solved by 6 out of the 7 systems. 6 of the 9 are also in the set of 7 hard ones from the modified versions of the problems.

Of all ATPs tested, only Prover9 performs worse on the modified problem set. We assume that this is due to the fact that we used EP to create the modified problems. We hope that Prover9 would also perform better with premise selection when being used directly in the Naproche system. Unfortunately this cannot be tested at the moment since Prover9 does not provide *Proof Summary* parseable output.

5 Conclusion and Future Work

While the details of the implementation are and should be up for discussion, the results we received suggest that the ideas behind the premise selection algorithm: textual adjacency, references and reusing the same ideas do seem to work and improve the ATP performance.

For further testing, we would like to compare and combine our approach with other axiom selection algorithms. Furthermore, it would be interesting to see how important the different aspects of the proof graph are.

The main focus for the future will be to create longer and more mathematical texts. Once we do have more material, we will experiment with different modification of the presented Proof Graph.

References

1. Barker-Plummer, D.: Gazing: An Approach to the Problem of Definition and Lemma Use. J. Autom. Reasoning 8(3), 311–344 (1992)
2. Benzmüller, C., Schiller, M., Siekmann, J.: Resource-bounded modelling and analysis of human-level interactive proofs. In: Crocker, M., Siekmann, J. (eds.) Resource Adaptive Cognitive Processes. Cognitive Technologies Series. Springer, Heidelberg (2010) (in print)

3. Cramer, M., Fisseni, B., Koepke, P., Kühlwein, D., Schröder, B., Veldman, J.: The Naproche Project: Controlled Natural Language Proof Checking of Mathematical Texts. In: Fuchs, N.E. (ed.) CNL 2009 Workshop. LNCS, vol. 5972, pp. 170–186. Springer, Heidelberg (2010)
4. Cramer, M.: Mathematisch-logische Aspekte von Beweisrepresentationsstrukturen. Master's thesis, University of Bonn (2009)
5. Koepke, P., Kühlwein, D., Cramer, M., Schröder, B.: The Naproche System (2009)
6. Ganesalingam, M.: The Language of Mathematics. PhD thesis, University of Cambridge (2009)
7. Heath, T.L., Euclid: The Thirteen Books of Euclid's Elements, Books 1 and 2. Dover Publications, New York (1956) (Incorporated)
8. Hoder, K.: Automated Reasoning in Large Knowledge Bases. Master's thesis, Charles University (2008)
9. Hurd, J.: First-Order Proof Tactics in Higher-Order Logic Theorem Provers. In: Design and Application of Strategies/Tactics in Higher Order Logics, number NASA/CP-2003-212448 in NASA Technical Reports, pp. 56–68 (2003)
10. Kuehlwein, D.: A Calculus for Proof Representation Structures. Master's thesis, University of Bonn (2008)
11. Landau, E.: Grundlagen der Analysis. Chelsea Publishing Company (1930)
12. Matuszewski, R., Rudnicki, P.: Mizar: the first 30 years. Mechanized Mathematics and Its Applications 4 (2005)
13. Meng, J., Paulson, L.C.: Lightweight relevance filtering for machine-generated resolution problems. J. Applied Logic 7(1), 41–57 (2009)
14. Nipkow, T., Paulson, L.C., Wenzel, M. (eds.): Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)
15. Schulz, S.: E – A Brainiac Theorem Prover. Journal of AI Communications 15(2/3), 111–126 (2002)
16. Sutcliffe, G.: The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. Journal of Automated Reasoning 43(4), 337–362 (2009)
17. Sutcliffe, G., Puzis, Y.: SRASS - A Semantic Relevance Axiom Selection System. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 295–310. Springer, Heidelberg (2007)
18. Urban, J., Sutcliffe, G., Pudlák, P., Vyskocil, J.: MaLARea SG1- Machine Learner for Automated Reasoning with Semantic Guidance. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 441–456. Springer, Heidelberg (2008)
19. Verchinine, K., Lyaletski, A., Paskevich, A.: System for Automated Deduction (SAD): A Tool for Proof Verification. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 398–403. Springer, Heidelberg (2007)
20. Vershinin, K., Paskevich, A.: ForTheL - the language of formal theories. International Journal of Information Theories and Applications 7(3), 120–126 (2000)
21. Wenzel, M.: Isabelle/Isar - a generic framework for human-readable proof documents. Studies in Logic, Grammar and Rhetoric, vol. 10(23). University of Białystok (2007)
22. Wos, L.: The problem of definition expansion and contraction. J. Autom. Reason. 3(4), 433–435 (1987)
23. Zinn, C.: Understanding Informal Mathematical Discourse. PhD thesis, Friedrich-Alexander-Universitt Erlangen Nürnberg (2004)

On the Saturation of YAGO

Martin Suda, Christoph Weidenbach, and Patrick Wischnewski

Max Planck Institute for Informatics, Saarbrücken, Germany
{suda,weidenbach,wischnew}@mpi-inf.mpg.de

Abstract. YAGO is an automatically generated ontology out of Wikipedia and WordNet. It is eventually represented in a proprietary flat text file format and a core comprises 10 million facts and formulas. We present a translation of YAGO into the Bernays-Schönfinkel Horn class with equality. A new variant of the superposition calculus is sound, complete and terminating for this class. Together with extended term indexing data structures the new calculus is implemented in SPASS-YAGO. YAGO can be finitely saturated by SPASS-YAGO in about 1 hour. We have found 49 inconsistencies in the original generated ontology which we have fixed. SPASS-YAGO can then prove non-trivial conjectures with respect to the resulting saturated and consistent clause set of about 1.4 GB in less than one second.

1 Introduction

YAGO (Yet Another Great Ontology) has been developed by our colleagues from the database/information retrieval group at the Max Planck Institute for Informatics [10]. It attracted a lot of attention in the information retrieval community because it was the first automatically retrieved ontology with both an accuracy of about 97% and a high coverage as it includes a unification of Wikipedia and WordNet. It contains about 20 million "facts" of the YAGO language. A detailed introduction to YAGO containing a comparison to other well-known ontologies can be found in [11].

After a close inspection of the YAGO language it turned out that the Bernays-Schoenfinkel Horn class with equality, abbreviated *BSHE* from now on, is sufficiently expressive to cover a core of YAGO. In 2008 the idea was born to write a translation procedure from YAGO into BSHE and then use SPASS in order to find *all* inconsistencies in YAGO and to answer queries. The translation procedure is described in Section 3. We then started running SPASS on the resulting formulas in a kind of "test and refine" loop, eventually leading to the SPASS-YAGO variant of SPASS, a new superposition calculus for BSHE, an extension to context tree indexing, and this paper.

The first step was actually to make SPASS ready for handling really big formula and clause sets. Some of this work went already into SPASS 3.5 [16], the basis for SPASS-YAGO, but further refinements were needed in order to actually start the experiments on YAGO. The engineering steps taken are explained in Section 6.

After the first experiments on smaller fragments of YAGO it immediately became clear that the standard superposition calculus does not work sufficiently well on BSHE. We started searching for a calculus that is sound, complete and terminating on BSHE and at the same time generates "small" saturations. The YAGO language assumes a unique name assumption, i.e., all constants are different. This can be translated into first-order logic by enumerating disequations $a \not\approx b$ for all different constants a, b. For several million constants this translation is not tractable. Bonancina and Schulz [7] therefore suggested additional inference rules instead of adding the disequations. We followed this approach and further refined one of their rules according to the BSHE fragment and the rest of our calculus. The BSH fragment can be decided by positive hyper resolution. Hyper resolution is a good choice anyway, because it prevents the prolific generation of intermediate resolvents of the form $\neg A_1 \vee \ldots \neg A_n \vee B$ that would be generated and kept by (ordered) binary resolution if there are no resolution partners for some $\neg A_i$. Experiments showed that this works nicely for most types of clauses resulting from the translation. For example, in YAGO a relation Q can be defined to be functional, translated into the clause $\neg Q(x, y) \vee \neg Q(x, z) \vee y \approx z$. If hyper resolution succeeds on generating a ground clause $(y \approx z)\sigma$ out of this clause, it is either a tautology or the unique name assumption rule mentioned above will refute the clause. The search space generated by hyper resolution out of subsort definitions and transitive relations contained in YAGO turned out to be too prolific. Therefore, we further composed our calculus by adding chaining for transitive relations [1] and sort reasoning [15]. The latter is available in Spass anyway, whereas for chaining we added a novel implementation. All details on the BSHE fragment generated out of YAGO and the eventual calculus including proofs for completeness, soundness, and termination plus implementation aspects are discussed in Section 4.

Thirdly, it turned out that the well-known indexing solutions for first-order theorem proving [6] are too inefficient for the size and structure of the YAGO BSHE fragment. The problem is that for example unifiability queries with a query atom $Q(x, a)$ need an index to both discriminate on the signature symbols Q and a without explicitly looking at all potential partner atoms in the index. In Section 5 we present an extension to context tree indexing [2] called *Filtered Context Trees* that discriminate for the above example on Q and a in logarithmic time in the number of symbols, i.e. in logarithmic time the filtered context tree index gives access to a structure that contains all potential partners containing these symbols. Context trees are a generalization of substitution trees used in Spass. In Spass-YAGO the context tree extension is finally implemented as an extension to substitution tree indexing.

Eventually, Spass-YAGO saturates the BSHE translation of YAGO in 1 hour, generating 26379349 clauses. The generated saturated clause set consists of 9943056 clauses. We found 49 inconsistencies which we resolved by hand. With respect to saturated clause set we can prove queries in less than one second (Section 7). The paper ends with a summary of the obtained results and directions for future work (Section 8). Detailed proofs and algorithms are available in a technical report [13]. Spass-YAGO and all input files are available from the Spass homepage `http://www.spass-prover.org/` in section prototypes and experiments.

2 Preliminaries

We follow the notation from [15]. A first-order language is constructed over a signature Σ. We assume Σ to be a finite set of function symbols. In addition to the signature Σ we assume that there is an infinite set $\mathcal{V}$ of variables. The set of terms $\mathcal{T}(\Sigma, \mathcal{X})$ over a signature Σ and a set of variables $\mathcal{X}$ with $\mathcal{X} \subset \mathcal{V}$ is recursively defined: $\mathcal{X} \subseteq \mathcal{T}(\Sigma, \mathcal{X})$ and for every function symbol $f \in \Sigma$ with arity zero (*a constant*) $f \in \mathcal{T}(\Sigma, \mathcal{X})$ and if f has arity n and $t_1, \dots t_n \in \mathcal{T}(\Sigma, \mathcal{X})$ then also $f(t_1, \dots t_n) \in \mathcal{T}(\Sigma, \mathcal{X})$. The variables $\mathcal{V} \setminus \mathcal{X}$ are used as meta variables in context tree indexing. Let $\text{vars}(t)$ for a term $t \in \mathcal{T}(\Sigma, \mathcal{X})$ be the set of all variables occurring in t. If $t = f(t_1, \dots, t_n)$ then $\text{top}(t) = f$.

A substitution $\sigma : \mathcal{V} \to \mathcal{T}(\Sigma, \mathcal{X})$ is a mapping from the set of variables into the set of terms such that $x\sigma \neq x$ for only finitely many $x \in \mathcal{V}$. The domain of a substitution σ is defined as $\text{dom}(\sigma) = \{x \mid x\sigma \neq x\}$ and the codomain is defined as $\text{cod}(\sigma) = \{x\sigma \mid x\sigma \neq x\}$. Substitutions are lifted to terms as usual. Given two terms s,t, a substitution σ is called a *unifier* if $s\sigma = t\sigma$ and *most general unifier* (mgu) if, in addition, for any other unifier τ of s and t there exists a substitution λ with $\sigma\lambda = \tau$. A substitution σ is called a *matcher* from s to t if $s\sigma = t$. The term s is then called a *generalization* of t and t an *instance* of s. A substitution σ is a *unifier for substitutions* τ and ρ if σ is a unifier of $x\tau$ and $x\rho$ for all $x \in \text{dom}(\tau)$. The definitions for matcher, generalization and instance can be lifted to substitutions analogously.

3 Translation of YAGO into BSHE

From a logical perspective, YAGO [10, 11] consists of about 20 million ground atoms of second-order logic. However, most of the second-order content is actually "syntactic sugar" that can be eventually translated into first-order logic. For example, subsort relations are represented as facts over the involved sort predicates. Representative examples of YAGO facts we have considered together with our clause translation are

bornIn AlbertEinstein Ulm $\mathit{bornIn}(\mathit{AlbertEinstein}, \mathit{Ulm})$	type bornIn yagoFunction $\neg \mathit{bornIn}(x, y) \lor \neg \mathit{bornIn}(x, z) \lor y \approx z$
locatedIn Ulm Germany $\mathit{locatedIn}(\mathit{Ulm}, \mathit{Germany})$	type locatedIn yagoTransitiveRelation $\neg \mathit{locatedIn}(x, y) \lor \neg \mathit{locatedIn}(y, z) \lor \mathit{locatedIn}(x, z)$
type AngelaMerkel human $\mathit{human}(\mathit{AngelaMerkel})$	subClassOf human mammal $\neg \mathit{human}(x) \lor \mathit{mammal}(x)$

The above kind of facts make up about half of YAGO, i.e., about 10 million facts translated into ground atoms and clauses of the above form. The translation results in first-order ground facts and non-unit clauses one half each. For this paper we left out YAGO facts about the source of information as well as confidence values attached to the facts. For example, in YAGO for each relation occurring in a YAGO fact there is also a fact relating it to the link of the website it was extracted from as well as further facts relating to links of other websites containing the same relation.

4 A New Calculus for BSHE

We translated YAGO into the Bernays-Schönfinkel Horn class with equality where all the clauses are range restricted. This means that any clause has the form $C \vee A$ or just C where

- Horn clauses: C contains only negative literals and A is a positive literal,
- range restricted: $Var(A) \subseteq Var(C_n)$, where C_n is the subclause of C consisting of all the non-equality atoms of C,
- Bernays-Schönfinkel: the signature Σ contains only constant symbols,
- equality ($\approx$) is present among the predicate symbols.

By using the unique name assumption, which is in our case imposed on all the constant symbols from Σ, the given set of clauses can be further simplified before starting the actual reasoning process. Each clause of the form $C \vee a \not\approx b$ is a tautology and can therefore be removed. If it is of the form $C \vee a \not\approx a$ the literal $a \not\approx a$ can be deleted. Moreover, clauses of the form $C \vee x \not\approx t$, for variable x and term t (either a variable or a constant) can be simplified to $C[x \leftarrow t]$. Thus we may assume that the clause set does not contain disequation literals. When we look at the positive occurrences of the equality predicate, we can do yet another simplification: a clause of the form $C \vee a \approx b$ can be simplified to C, because $a \approx b$ is false in any interpretation satisfying the unique name assumption. Thus, when starting our saturation process, we can assume that the given clause set only contains positive occurrences of the equality predicate. As noted in the introduction, we used the refinement of the calculus presented in [7] to deal with the unique name assumption.

Another key ingredient in the process of saturation of YAGO is the chaining calculus, a refinement of superposition designed to deal efficiently with transitive relations [1]. It is well known that the axiom stating that a relation Q is transitive,

$$Q(x,y) \wedge Q(y,z) \rightarrow Q(x,z),$$

may be a source of non-termination in resolution proving. This is because the transitivity axiom clause may be resolved with (a variant of) itself to yield a new clause $Q(x,y) \wedge Q(y,z) \wedge Q(z,w) \rightarrow Q(x,w)$. Evidently, such process can be arbitrarily iterated. Even if we use selection of negative literals or hyperresolution to block the self-inference, (hyper)resolution will eventually explicitly compute the whole transitive closure of the relation Q.

The idea of chaining is to remove the prolific transitivity axiom from the given clause set, and instead to introduce a couple of specialized inference rules that encode the logical consequences of transitivity in a controlled way. The crucial restriction lies in requiring that the two literals $Q(u,v)$ and $Q(v,w)$ chain together only if $v \succeq u$ and $v \succeq w$, where $\succ$ is a standard superposition term ordering. In order to show that such a restricted version of the rule is still complete techniques from term rewriting are employed.

An important step in introducing the chaining calculus to a theorem prover is the implementation of a new literal ordering. In the standard superposition setting literal ordering is typically defined as the two-fold multiset extension of the term ordering on the so called *occurrences* of equations/atoms (see e.g. [15] for details). This, for instance, entails that $\neg A \succ A$ for any atom A, a property essential for the completeness of the calculus. Unfortunately, however, stronger properties are required for the chaining to work, namely to ensure that the chaining inferences are decreasing, i.e. that the conclusion of an inference is always smaller than the maximal premise. These properties are integrated under the notion of *admissibility* of the literal ordering.

Definition 1. *An ordering $\succ$ on ground terms and literals is called* admissible [1] *if*

– *it is well-founded and total on ground terms and literals,*
– *it is* compatible with reduction on maximal subterms, *i.e. $L \succ L'$ whenever L and L' contain the same transitive predicate symbol Q, and the maximal subterm of L' is strictly smaller than the maximal subterm of L,*
– *it is* compatible with goal reduction, *i.e.*
  - $\neg A \succ A$ *for all ground atoms A,*
  - $\neg A \succ B$ *whenever A is an atom $Q(s,t)$ and B is an atom $Q(s',t')$, such that Q is a transitive predicate and $max(s,t) \succeq max(s',t')$,*
  - $\neg A \succ \neg B$ *whenever A is an atom $Q(s,s)$ and B atom $Q(s,t)$ or $Q(t,s)$, where Q is a transitive predicate and $s \succ t$.*

An ordering on ground clauses is called admissible *if it is the multiset extension of an admissible ordering on literals.*

In order to actually implement an admissible ordering on literals, we can work as follows. We associate to each L a tuple (max_L, p_L, min_L) and compare these lexicographically, using the superposition term ordering $\succ$ in the first and last component, and the ordering $1 > 0$ in the middle component. The individual members of the tuple are defined as follows: If L is of the form $Q(s,t)$ for a transitive predicate Q we set max_L to the maximum of s and t, and min_L to the minimum of the two terms (with respect to $\succ$). If L is of the form A or $\neg A$ for some atom A the top symbol of which is not a transitive predicate, we set $max_L = A$ and $min_L = \top$, where $\top$ is special symbol minimal in the term ordering $\succ$. We set $p_L = 1$, if L is negative, and 0 otherwise.

We use $\succ$ to denote both the standard term ordering, which is as usual assumed to be total on ground terms, and the just described admissible ordering on literals and clauses. Context should always make clear what instance of $\succ$ is meant.

Lifting the lexicographic ordering of the tuples to the non-ground level is a non-trivial task. For instance, the maximum of s and t may not be unique, because the term ordering $\succ$ cannot be total on non-ground terms. Nevertheless, it is possible to proceed by simultaneously considering both cases. Then it can happen that we produce a distinctive result, as opposed to just reporting incomparability of the two literals in question, which is always a sound solution, because it only means that more inference will potentially have to be done. For example, comparing the two non-ground literals $L_1 = \neg R(s_1, t_1)$ and $L_2 = R(s_2, t_2)$ where the term pairs s_1, t_1, s_2, t_2, and t_1, s_2 are incomparable respectively, but $s_1 \succ s_2$ and $t_1 = t_2$, we can report that $L_1 \succ L_2$ although we don't know whether max_{L_1} is s_1 or t_1. In the second case the comparison of the p_L-member of the tuple would take over. Obviously, we try to identify as many such cases as possible, to be able to restrict applicability of the inferences.

4.1 The Proof System

Here we present the inference rules of our calculus. They are refinements of calculi presented in [1] and [7] composed and specialized for BSHE. For the chaining rules, we assume that Q is a transitive predicate.

Ordered Chaining

$$\frac{Q(l,s) \quad Q(t,r)}{Q(l,r)\sigma}$$

where σ is the most general unifier of s and t, $l\sigma \not\preceq s\sigma$, and $r\sigma \not\preceq t\sigma$.

Negative Chaining

$$\frac{Q(l,s) \quad D \vee \neg Q(t,r)}{D\sigma \vee \neg Q(s,r)\sigma}$$

where σ is the most general unifier of l and t, $s\sigma \not\preceq l\sigma$, and $r\sigma \not\preceq t\sigma$, and

$$\frac{Q(l,s) \quad D \vee \neg Q(t,r)}{D\sigma \vee \neg Q(t,l)\sigma}$$

where σ is the most general unifier of s and r, $l\sigma \not\preceq s\sigma$, and $t\sigma \not\prec r\sigma$.

Hyperresolution

$$\frac{A_1 \quad \ldots \quad A_n \quad \neg B_1 \vee \ldots \vee \neg B_n \vee P}{P\sigma},$$

where $n \geq 1$, $A_1, \ldots, A_n$ are unit clauses, P is a positive literal or false, and σ is the simultaneous most general unifier of A_i and B_i respectively, for all $i \in \{1, \ldots n\}$.

OECut [7]

$$\frac{a \approx b}{\perp},$$

where a and b are two different constants.

In negative chaining, the case $t\sigma = r\sigma$ needs to be dealt with by only one of the two negative chaining rules. We do not impose maximality restrictions on the negative literal as this would cause incompleteness in the combination with hyperresolution. Positive hyperresolution alone is known to decide Horn function-free clauses, but with respect to YAGO the search space becomes too prolific. Therefore, we developed the above calculus where transitivity is replaced by the specific chaining rules.

4.2 Completeness, Soundness, and Termination

In this section we show that our calculus is sound and terminating for the Bernays-Schönfinkel Horn class with equality with range restricted clauses. Before that, however, we explain how the completeness of the calculus can be established.

The completeness proof is based on the ideas from [1] adapted to our special case. It incorporates the notion of redundancy, so the standard elimination rules like subsumption and tautology deletion can be added to the calculus. The model construction itself proceeds along standard lines. One takes the set of all ground instances of the given saturated clause set, and uses the clause ordering which is total and well-founded on the ground level to inductively build partial interpretations. In order to satisfy all the clauses in the final interpretation, some of the clauses are designated as productive, which means they contribute with a positive atom to the interpretation. A specialty of our case is that we additionally need to consider a closure of the contributed atoms in order to obtain the right interpretation. Moreover, we only allow positive unit clauses to potentially become productive (this can be justified by viewing all the negative literals as implicitly selected). The full proof of the model construction lemma along with all the necessary definitions can be found in [13].

Theorem 2 (Completeness). *If a set of Horn clauses N is saturated up to redundancy then the set $N \cup \mathrm{TRANS} \cup \mathrm{UNA}$ is unsatisfiable if and only if N contains the empty clause, where* TRANS *is the set of transitivity axioms for predicates $Q_1, \ldots, Q_l$, and* $\mathrm{UNA} = \{a \not\approx b | a \neq b, a \in \Sigma, b \in \Sigma\}$.

Proof. If N does not contain the empty clause, we claim that the Herbrand interpretation I constructed from the set of all ground instances (see [13] for the details about the construction) of N is a model of $N \cup \mathrm{TRANS} \cup \mathrm{UNA}$. Via the usual lifting argument[1] the set of all ground instances is saturated as well. By the model construction lemma, every ground instance C of a clause in N is true in I, and in addition I is a transitivity interpretation and satisfies the unique name assumption.

Theorem 3 (Soundness). *The presented calculus is sound. Conclusion of any inference is logically entailed by the premises of the inference and the theory* $(\mathrm{TRANS} \cup \mathrm{UNA})$.

Proof. The claim is obvious for hyperresolution, and also for the OECut rule, where we use the unique name assumption. Finally, all the chaining rules can be simulated as two resolution steps between the participating premises and the appropriate transitivity axiom clause.

Theorem 4 (Termination). *The calculus terminates on the set of Horn clauses from Bernays-Schönfinkel class.*

[1] Note that we only consider ground version of the OECut rule. Nevertheless, it does not need to be lifted in our case. It is because our clauses are range restricted, and therefore we can never generate a non-ground positive (unit) clause.

Proof. No inference rule produces a longer clause than any of its premises. There are only finitely many clauses of given length (up to variable renaming) as all the function symbols are constants.

5 Filtered Context Trees

The translation of YAGO into the BSHE class results in several million clauses. In order to reason about these formulas, a fast indexing mechanism is vital. The atoms occurring in the clauses are of the form: $Q(a, b)$, $Q(a, x)$, $Q(x, b)$, $Q(x, y)$, $S(a)$ and $S(x)$, where Q is a binary predicate symbol, a, b are constants and S is a monadic predicate (sort symbol) from the signature. In order to perform retrieval operations on an index containing such atoms, we have to discriminate efficiently on all available signature symbols. In order to achieve this we develop a filtering mechanism for the context tree indexing [2]. The filtering performs in logarithmic time and filters out subtrees of the indexing that do not lead to a success with respect to the current retrieval operation.

Context tree indexing is a generalization of substitution tree indexing [3]. Compared to substitution trees, context trees can additionally share common subterms even if they occur below different function symbols via the introduction of extra variables for function symbols which we call function variables. For example, the terms $f(s, t)$ and $g(s, t)$ can be represented as $F(s, t)$ with children $F = f$ and $F = g$. The function variable F represents a single function symbol. In the context of deep formulas, this potentially increases the degree of sharing in the index structure. Figure 1 depicts a context tree containing the terms $f(c, a)$, $h(d, a)$, $g(a, d)$, $g(d, b)$, and $g(e, b)$.

We assume a set of function variables $\mathcal{U} \subset \mathcal{V}$ which is disjoint from the set of variables $\mathcal{X}$. The set of terms $\mathcal{T}(\Sigma \cup \mathcal{U}, \mathcal{X})$ are terms built over the signature Σ, the function variables $\mathcal{U}$ and the variables $\mathcal{X}$. The notion of a substitution can be adapted accordingly. In addition, we assume a set of index variables $\mathcal{W} \subset \mathcal{V}$ which is pairwise disjoint from $\mathcal{X}$ and $\mathcal{U}$. Index variables are internal variables of a context tree. Index variables are denoted by w_i. We also assume a set of index function variables which are denoted by F_i.

In the context of YAGO, the notion of function variables provides an extended query mechanism. For example, we can query the index for terms that contain the symbol a as their second argument without fixing the top symbol. The respective query term is $F(x, a)$. Applying this query to the context tree given in Figure 1, returns the terms $f(c, a)$ and $h(d, a)$.

The following example demonstrates a retrieval operation on a context tree. Consider the context tree of Figure 1 and the retrieval of terms unifiable with the term $g(e, x)$. The retrieval operation works with respect to a query substitution containing the query term. The query substitutions ρ for $g(e, x)$ is $\rho = \{w_0 \mapsto g(e, x)\}$ where w_0 is the first indexing variable occurring in the context tree. The algorithm starts with the query substitution ρ at the node whose substitution is τ_0. The substitution τ_0 is unifiable with ρ using the substitution $\sigma = \{w_1 \mapsto e, w_2 \mapsto x, F_1 \mapsto g\}$. Descending the indexing further requires to

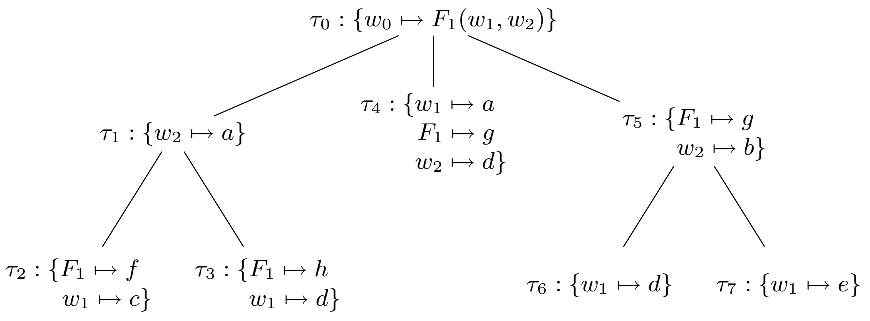

$\tau_0 : \{w_0 \mapsto F_1(w_1, w_2)\}$
$\tau_1 : \{w_2 \mapsto a\}$
$\tau_4 : \{w_1 \mapsto a, F_1 \mapsto g, w_2 \mapsto d\}$
$\tau_5 : \{F_1 \mapsto g, w_2 \mapsto b\}$
$\tau_2 : \{F_1 \mapsto f, w_1 \mapsto c\}$
$\tau_3 : \{F_1 \mapsto h, w_1 \mapsto d\}$
$\tau_6 : \{w_1 \mapsto d\}$
$\tau_7 : \{w_1 \mapsto e\}$

Fig. 1. Context Tree

check all subnodes. In this case, these are the nodes containing τ_1, τ_4 and τ_5. Unifiable under the current substitution $\rho \circ \sigma$ are the substitutions τ_1 and τ_5. At first, the algorithm proceeds by inspecting the subtree starting at the node with τ_1. The substitution τ_1 is unifiable with $\rho \circ \sigma$ using $\sigma' = \{x \mapsto a\}$. Continuing with the subnodes, the algorithms recognizes that neither τ_2 nor τ_3 are unifiable with $\rho \circ \sigma \circ \sigma'$. Then the algorithm backtracks, proceeds with τ_5 and eventually finds a leaf where all substitutions along the path τ_0, τ_5, τ_7 are unifiable under the respective substitution ρ and returns the desired term which is $w_0\tau_0\tau_5\tau_7$.

In this example, after examining the node containing the substitution τ_0, the retrieval procedure proceeds by examining all subnodes. These subnodes are the nodes containing the substitutions τ_1, τ_4 and τ_5. However, if we inspect the query substitution and the subtree starting at the node with the substitution τ_1 we recognize that the symbol g does not occur in any substitution of this subtree. Consequently, the retrieval procedure does not need to process this subtree.

In the case of Spass-YAGO, it is vital to efficiently exclude subtrees that do not contribute to the current retrieval operation because one node may have millions of subnodes and the term indexing is processed several thousand times in a reasoning loop. Therefore, we develop in the following an improvement of the indexing which filters out subtrees from the indexing that do not contribute. In the case of YAGO, the filtering avoids inspecting several million nodes during one retrieval operation. Without our filtering technique, Spass was even unable to load the core of YAGO into the index in reasonable time.

The following gives an overview over context trees and introduces the new filtered context trees. Additionally, we present the retrieval algorithm together with some details about the implementation of filtered context trees in Spass. For a complete introduction of filtered context trees and the respective algorithms we refer to the technical report [13].

Definition 5 (Context Tree). *A context tree is a tree $T = (V, E, \mathrm{subst}, v_r)$ where V is a set of vertexes, $E \subset V \times V$ is the edge relation, the function* subst *assigns to each vertex a substitution, $v_r \in V$ is the root node of T and the following properties hold:*

1. *each node is either a leaf or an inner node with at least two children.*

2. for every path $v_1 \dots v_n$ from the root ($v_1 = v_r$) to any node it holds:

$$\mathrm{dom}(\mathrm{subst}(v_i)) \cap \bigcup_{1 \leq j < i} \mathrm{dom}(\mathrm{subst}(v_j)) = \emptyset$$

3. for every path $v_1 \dots v_n$ from the root ($v_1 = v_r$) to a leaf v_n

$$\mathrm{vars}(\mathrm{cod}(\mathrm{subst}(v_1) \circ \dots \circ \mathrm{subst}(v_n))) \subset \mathcal{X}$$

Each node in a context tree which is not a leaf node, must have at least two subtrees due to the first condition. The second condition ensures that each variable is bound at most once along a path. The third condition assures that all terms represented by a path from the root to a leaf are from $\mathcal{T}(\Sigma, \mathcal{X})$.

Definition 6 (Variables of a path). *Let $v_1, \dots, v_n$ be a path from the root of a context tree to a node v_n then the set of variables of this path is*

$$\mathrm{vars}(v_1, \dots, v_n) \ = \bigcup_{i \in \{1 \dots n\}} \mathrm{vars}(\mathrm{cod}(\mathrm{subst}(v_i))) \setminus \bigcup_{i \in \{1 \dots n\}} \mathrm{dom}(\mathrm{subst}(v_i))$$

For obtaining filtered context trees, we extend the notion of context trees. To each node v we add a map M from any symbol s to each subnode of v that contains s in one substitution along a path starting at v including v itself.

Reconsider the above example with the unification retrieval operation for the query substitution $\rho = \{w_0 \mapsto g(e, x)\}$. Extending Figure 1 with the map M yields the filtered context tree depicted in Figure 2. The retrieval algorithm applied to Figure 2 examines the node containing the substitution τ_0. As we have seen in the above example, only those subtrees can contribute to the current retrieval operation that contain g in the codomain of the substitution of any of its nodes. The function M contains exactly this information. If we apply g to M, the function M returns those subtrees; in our example these are the subtrees starting at the nodes containing the substitution τ_4 and τ_5. Consequently, the node containing the substitutions τ_1 is not considered during the retrieval.

A mapping mechanism has also been used for discrimination trees. In discrimination tree indexing the mapping assigns to a given label the respective successor node of the discrimination tree. For example, this has been added to the indexing of the theorem prover E [9].

In order to be able to use a mapping mechanism for context trees we have to define a function that assigns a set of signature symbols to a given substitution. We define the characteristic function for a substitution as the set of top symbols occurring in its codomain. In order to also characterize a substitution containing only variables in its codomain we assume a symbol $\perp$ with $\perp \notin \Sigma$.

Definition 7 (Characteristic Function). *Let σ be a substitution and $\mathcal{O}$ be a set of variables. The set of top symbols of σ and $\mathcal{O}$ is defined as*

$$ts(\sigma, \mathcal{O}) = \{f \mid \exists x \in \mathrm{dom}(\sigma) \cap \mathcal{O} \textit{ with } x\sigma = f(t_1, \dots, t_n)\}$$

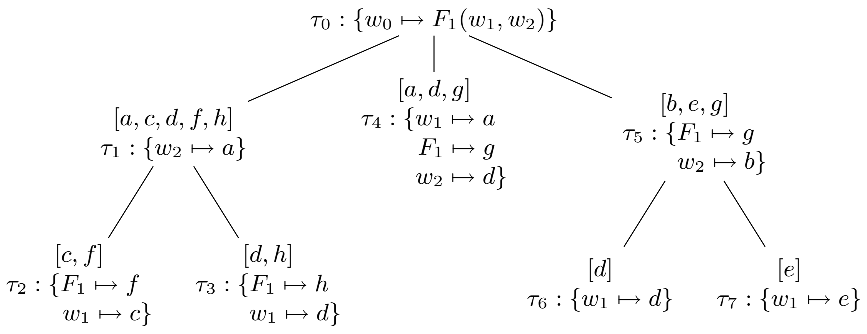

$\tau_0 : \{w_0 \mapsto F_1(w_1, w_2)\}$
$[a, c, d, f, h]$
$\tau_1 : \{w_2 \mapsto a\}$
$[a, d, g]$
$\tau_4 : \{w_1 \mapsto a$ $F_1 \mapsto g$ $w_2 \mapsto d\}$
$[b, e, g]$
$\tau_5 : \{F_1 \mapsto g$ $w_2 \mapsto b\}$
$[c, f]$
$\tau_2 : \{F_1 \mapsto f$ $w_1 \mapsto c\}$
$[d, h]$
$\tau_3 : \{F_1 \mapsto h$ $w_1 \mapsto d\}$
$[d]$
$\tau_6 : \{w_1 \mapsto d\}$
$[e]$
$\tau_7 : \{w_1 \mapsto e\}$

Fig. 2. Filtered Context Tree

The characteristic function $\mathrm{chr}(\sigma, \mathcal{O})$ *of a substitution* σ *with respect to the set of variables* $\mathcal{O}$ *is defined as follows:*

$$\mathrm{chr}(\sigma, \mathcal{O}) = \begin{cases} ts(\sigma, \mathcal{O}) & \text{if } ts(\sigma, \mathcal{O}) \neq \emptyset \\ \{\bot\} & \text{if } ts(\sigma, \mathcal{O}) = \emptyset \wedge \exists x \in \mathrm{dom}(\sigma) \text{ with} \\ & \quad x\sigma \in \mathcal{X} \vee x\sigma \in \mathcal{T}(\Sigma \cup \mathcal{U}, \mathcal{X}) \setminus \mathcal{T}(\Sigma, \mathcal{X}) \vee x \in \mathcal{X} \\ \emptyset & \text{otherwise} \end{cases}$$

Note that this definition also includes the cases where $x\sigma$ is a constant or $x\sigma$ is a function symbol mapped from a function variable.

Reconsider the above example with query substitution $\rho = \{w_0 \mapsto g(e, x)\}$. The characteristic function of ρ is $\mathrm{chr}(\rho, \{w_0\}) = \{g\}$. Note that g is the only symbol of the characteristic function of ρ because this is the top symbol of the term $g(e, x)$. A term that is unifiable with $g(e, x)$ is of the form $g(y, x)$, where y is either a variable or the constant e. Consequently, the symbol g is the only symbol characterizing ρ.

Once we have defined the characteristic function for a substitution, we can extend the definition of context trees with a function M that assigns to a given node v and a symbol s a set of successor nodes. For each node v' in the set of successor nodes it holds that there is a node on a path, starting at v', which contains the symbol s in the characteristic function of its substitution. This lifts the characteristic function of a substitution of one node to the characteristic of a subtree of a context tree.

Definition 8 (Filtered Context Tree). *A* filtered context tree $FT = (V, E, \mathrm{subst}, v_r, M)$ *is a context tree* $(V, E, \mathrm{subst}, v_r)$ *together with a function* $M : V \times (\Sigma \cup \{\bot\}) \to 2^V$ *from nodes and function symbols to a subset of* V *such that* $v_{k+1} \in M(v_k, s)$ *iff there is a path* $v_1, \ldots, v_k, v_{k+1}, \ldots, v_n$ *where* $v_1 = v_r$ *is the root node with*

$$s \in \bigcup_{i \in \{k+1, \ldots, n\}} \mathrm{chr}(\mathrm{subst}(v_i), \mathrm{vars}(v_1, \ldots, v_k))$$

Algorithms for Filtered Context Trees. The procedure FilteredLookup (Algorithm 1) depicts the function performing the lookup operation on a given filtered context tree FT, a node v_n, a query substitution ρ and a function Test. The node v_n is the current examined node of FT during the recursive application of FilteredLookup. Initially, v_n is the root node v_r. The function Test is one of the test functions unification, generalization or instantiation. These test functions get two substitutions as their arguments and check if these substitutions are unifiable, are generalizations or are instances of each other, respectively. Consequently, these are independent of the underlying indexing structure and we can use the standard algorithms of these tests for the implementation of filtered context tree indexing.

Algorithm 1. FilteredLookup

```
   Input: $FT = (V, E, \text{subst}, v_r, M)$, $v_n \in V$, substitution $\rho$, function Test
1  $HITS = \emptyset$;
   /* $v_r = v_1, ..., v_n$ path from the root $v_r$ to $v_n$                    */
2  $C = \text{chr}(\rho, \text{vars}(v_r, ..., v_n))$;
3  if $C = \{\perp\}$ then $N = \{v' | (v_n, v') \in E\}$;
4  else if $C \neq \emptyset$ then $N = \bigcup_{s \in C \cup \{\perp\})} M(v_n, s)$;
5  else $N = \emptyset$;
6  foreach $v' \in N$ do
7      if $Test(\text{subst}(v'), \rho) = (true, \sigma)$ then
8          if $isLeaf(v')$ then return $\{v'\}$;
9          $HITS = HITS \cup \text{FilteredLookup}(FT, v', \rho \circ \sigma, \text{Test})$;
10     end
11 end
12 return $HITS$
```

FilteredLookup (Algorithm 1) first computes the characteristic function in line 2. If the characteristic function returns $\{\perp\}$ then the algorithm inspects all subnodes of the given node v_n. Otherwise, the algorithm looks for the symbols in M and considers only those nodes which are returned by M (line 4). From this point on FilteredLookup is exactly the implementation of the corresponding retrieval operation of context tree indexing.

Computing the characteristic of the substitution ρ in line 2 is in time $O(|\operatorname{dom}(\rho)|)$, where $|\operatorname{dom}(\rho)|$ is the number of elements of the domain of ρ. As a result, obtaining the set N from M in line 4 is in time $O(|\operatorname{dom}(\rho)| * \log |\Sigma|)$ where $|\Sigma|$ is the number of symbols in the signature. Hence, the overhead for the filtering is in $O(|\operatorname{dom}(\rho)| * \log |\Sigma|)$.

The algorithms for insertion and deletion of the context tree indexing have to be adapted to work with filtered context trees. The search for the correct insert and deletion position, respectively, have to be adapted analogously to FilteredLookup. Additionally, the procedures have to maintain the map M. This works in time $O(n * |\operatorname{dom}(\tau)| * \log |\Sigma|)$ where n is the path length from the root to the insert (deletion) position, $|\operatorname{dom}(\tau)|$ is the number of elements of the domain of τ and $|\Sigma|$ is the number of symbols in the signature.

Implementation in Spass-YAGO. Since context trees are a generalization of substitution trees and Spass has an implementation of substitution tree indexing, our implementation of Spass-YAGO contains the substitution tree indexing of Spass together with the above described filter techniques.

In Spass, symbols are internally represented as integers. Consequently, they can be compared with respect to their integer value. So we implemented the lookup function M using CSB^+-trees [8].

For further details about the algorithms, the implementation in Spass and further optimization techniques of the filtered context tree indexing we refer to the technical report [13].

6 Engineering

In order to accommodate Spass to the new indexing technique and the calculus for BSHE, a lot of extra engineering had to be performed. We increased the maximal number of signature symbols that Spass can handle to 19M. The parsing module was modified, so that originally quadratic manipulations on the lists of input clauses now only take linear time. Algorithms for manipulating clause sets holding Spass's search state, such as loading the usable clauses, or sorting clause lists, were sped up from $O(n^2)$ to $O(n * \mathsf{log}(n))$. Hashmaps used in the clausification process in Flotter had to be extended to reduce the number of hash-conflicts. The structure for storing superterms in the sharing was changed from lists to maps. Newly derived clauses are now inserted at the first possible position with respect to weight in the list of usable clauses, instead of also considering search space depth. Finally, Spass-YAGO skips auto-configuration and instead uses a standard complete flag setting in the input files according to our calculus (Section 4).

There is still plenty of room for speed ups via further engineering. Our motivation was not on getting a much faster system but to advance Spass such that it can cope with the size of YAGO.

7 Experiments

We ran our experiments on 4 x Intel Xeon Processor X5560 (8M Cache, 2.80 GHz) Debian Linux machine with 48 GB RAM. We compared Spass-YAGO with iProver version 0.7 [5], E version 1.1 [9], and Spass version 3.5 [16] including the before mentioned engineering improvements. The reason for this comparison is only to show that our new calculus, filtered context tree indexing and improved implementation advances the state of the art in automated reasoning on large ontologies. None of the above systems has been specifically designed to fit the BSHE theory created out of YAGO. All the provers were recompiled for the above 64 bit architecture to better cope with the large inputs.

First we evaluated the task of showing satisfiability of (slices of) YAGO after having removed all inconsistencies by hand on the basis of Spass-YAGO runs. The examples are in favor of iProver, E, and Spass 3.5 as we did not include

Slices	Input size [F]	Time to saturate	Output size [F]	Other provers
S_0	136808	12.5	+1768	fail
S_1	132080	9.7	+16060	fail
S_2	96454	9.9	+1768	fail
S_3	114527	10.6	+4769	fail
S_4	4891235	37:11.1	+24123	fail
Full	9918933	1:03:24.0	+24123	fail

Fig. 3. Saturating YAGO

the unique name assumption units for those provers, whereas Spass-YAGO tests the corresponding inference rule. The results are given in Figure 3.

The second column shows the number of formulas (clauses), the third the time needed for saturation, and the fourth the number of additionally eventually kept clauses by Spass-YAGO. All other provers fail on showing any of the examples due to timing constraints of 60 min for the first 4 slices and due to running out of (internal) memory (except for Spass and E running out of time) for S_4 and the full set.

Note that showing satisfiability is the more difficult problem compared to actually proving queries. All provers can successfully solve queries with respect to at least one of the S_0-S_4 slices.

Since none of the other provers could handle the overall core, we only carried out the second experiment on queries using Spass-YAGO. We evaluated the following two queries on the saturated core of YAGO, where we applied the now complete SOS strategy.

$$Q_1 \qquad \exists x.\mathit{politician}(x) \wedge \mathit{physicist}(x) \wedge \mathit{bornIn}(x, \mathit{Hamburg}) \wedge \mathit{hasSuccessor}(\mathit{Helmut\_Schmidt}, x)$$

$$Q_2 \qquad \exists x, y, z.\mathit{diedIn}(x, y) \wedge \mathit{hasChild}(x, z) \wedge \mathit{bornIn}(z, y) \wedge \mathit{locatedIn}(y, \mathit{New\_York})$$

The results of the querying are shown in the table below.

Query	Derived	Kept	Proof length	Reasoning	Total
Q_1	1	1	18	0:00.1	9:37.8
Q_2	9	0	6	0:00.1	9:38.3

The table shows the number of derived, kept clauses and the length of the proof found by Spass-YAGO. Actually, almost all of the time is spent on loading the overall clause set, the difference between total time and reasoning time. The time for answering the queries is below one second. The difference between derived/kept clauses and proof length is the result of simplification, in particular sort simplification exploring subsort relationships. Recall that in the saturated core not all ground consequences of YAGO are explicitly represented. So the involved reasoning goes beyond simple data base style joins but involves reasoning about transitivity and subsort relationships.

8 Conclusion

The saturation of large ontologies is a challenge for first-order reasoning. The core of the YAGO ontology can be saturated by Spass-YAGO in about 1 hour (Section 7) due to a new complete, sound, and terminating variant of the superposition calculus (Section 4) accompanied by filtered context tree indexing (Section 5) and improved implementations (Section 6). Spass-YAGO significantly advances the state of the art in theorem proving on large ontologies (Section 7). It complements other efforts in this direction. The yearly CASC division on ontology reasoning [14] as well as approaches on combining theorem provers with other sources of knowledge [12] concentrate on finding proofs (answers, contradictions), not saturations, i.e. models of an overall ontology as we have studied in this paper for a core of YAGO. One of the first contributions on applying theorem proving to large ontologies is [4] where a number of engineering questions are discussed.

Most importantly, we showed that standard automated reasoning tools such as Spass are able to cope with large ontologies such as a core of YAGO if the calculus and implementation are adopted accordingly. Currently, our implementation does not directly give answers but shows proofs. This can be straightforwardly extended to an answer mechanism. The queries we considered are solely existentially quantified. This can be extended to arbitrary quantifier prefixes, because we are considering a finite domain only. However, it needs further research in order to cope with the potential search space spanned by such a query. Here an even more refined calculus, e.g. by integrating chaining directly into the hyper resolution inference is instrumental. Finally, reasoning with respect to the confidence values attached to facts in YAGO that are ignored for this paper could be added to the calculus, e.g. in the style of a multi-valued logic aggregating formulas at their respective confidence values.

References

[1] Bachmair, L., Ganzinger, H.: Ordered chaining calculi for first-order theories of transitive relations. Journal of the ACM (JACM) 45(6), 1007–1049 (1998)

[2] Ganzinger, H., Nieuwenhuis, R., Nivela, P.: Context trees. In: Goré, R., Leitsch, A., Nipkow, T. (eds.) IJCAR 2001. LNCS (LNAI), vol. 2083, pp. 242–256. Springer, Heidelberg (2001)

[3] Graf, P.: Term Indexing. LNCS, vol. 1053. Springer, Heidelberg (1996)

[4] Horrocks, I., Voronkov, A.: Reasoning support for expressive ontology languages using a theorem prover. In: Dix, J., Hegner, S.J. (eds.) FoIKS 2006. LNCS, vol. 3861, pp. 201–218. Springer, Heidelberg (2006)

[5] Korovin, K.: iProver – An Instantiation-Based Theorem Prover for First-Order Logic (System Description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 292–298. Springer, Heidelberg (2008)
[6] Ramakrishnan, I.V., Sekar, R.C., Voronkov, A.: Term indexing. In: Robinson, J.A., Voronkov, A. (eds.) Handbook of Automated Reasoning, pp. 1853–1964. Elsevier/MIT Press (2001)
[7] Schulz, S., Bonacina, M.P.: On Handling Distinct Objects in the Superposition Calculus. In: Konev, B., Schulz, S. (eds.) Proc. of the 5th International Workshop on the Implementation of Logics, Montevideo, Uruguay, pp. 66–77 (2005)
[8] Rao, J., Ross, K.A.: Making B$^+$-trees cache conscious in main memory. In: ACM SIGMOD International Conference on Management of Data, pp. 475–486 (2000)
[9] Schulz, S.: E - a brainiac theorem prover. AI Communication 15(2-3), 111–126 (2002)
[10] Suchanek, F.M., Kasneci, G., Weikum, G.: Yago: A Core of Semantic Knowledge. In: 16th international World Wide Web conference (WWW 2007), Banff, Canada, pp. 697–706. ACM Press, New York (2007)
[11] Suchanek, F.M., Kasneci, G., Weikum, G.: YAGO: A Large Ontology from Wikipedia and WordNet. J. Web Sem. 6(3), 203–217 (2008)
[12] Suda, M., Sutcliffe, G., Wischnewski, P., Lamotte-Schubert, M., de Melo, G.: External sources of axioms in automated theorem proving. In: Mertsching, B., Hund, M., Aziz, M.Z. (eds.) KI 2009. LNCS, vol. 5803, pp. 281–288. Springer, Heidelberg (2009)
[13] Suda, M., Weidenbach, C., Wischnewski, P.: On the Saturation of YAGO. Research Report MPI-I-2010-RG1-001, Max-Planck-Institut für Informatik, Saarbrücken (2010)
[14] Sutcliffe, G.: The 4th IJCAR Automated Theorem Proving System Competition - CASC-J4. AI Communication 22(1), 59–72 (2009)
[15] Weidenbach, C.: Combining superposition, sorts and splitting. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, ch. 27, vol. 2, pp. 1965–2012. Elsevier, Amsterdam (2001)
[16] Weidenbach, C., Dimova, D., Fietzke, A., Suda, M., Wischnewski, P.: SPASS Version 3.5. In: Schmidt, R.A. (ed.) CADE 2009. LNCS, vol. 5663, pp. 140–145. Springer, Heidelberg (2009)

Optimized Description Logic Reasoning via Core Blocking

Birte Glimm, Ian Horrocks, and Boris Motik

Oxford University Computing Laboratory, UK

Abstract. State of the art reasoners for expressive description logics, such as those that underpin the OWL ontology language, are typically based on highly optimized implementations of (hyper)tableau algorithms. Despite numerous optimizations, certain ontologies encountered in practice still pose significant challenges to such reasoners, mainly because of the size of the model abstractions that they construct. To address this problem, we propose a new blocking technique that tries to identify and halt redundant construction at a much earlier stage than standard blocking techniques. An evaluation of a prototypical implementation in the HermiT reasoner shows that our technique can dramatically reduce the size of constructed model abstractions and reduce reasoning time.

1 Introduction

Description logics (DLs) are a family of logic based knowledge representation formalisms [1] widely used in conceptual modeling, and they provide the formal basis for the OWL 2 ontology language [2]. DL reasoners support the development and deployment of ontologies in numerous tools and applications. State of the art reasoners for expressive DLs are typically based on highly optimized variants of (hyper)tableau algorithms—model building procedures that decide the (un)satisfiability of a knowledge base $\mathcal{K}$ via a constructive search for an abstraction of a model for $\mathcal{K}$. Examples of such reasoners include FaCT++ [3], HermiT [4], Pellet [5], and Racer [6].

Despite numerous optimizations, certain existing and emerging knowledge bases still pose significant challenges to such reasoners. In particular, state of the art (hyper)tableau reasoners use a cycle detection technique called *blocking* to ensure that only finite model abstractions are constructed. These abstractions can, however, be very large, which can be problematical with respect to space and time: the available memory may be exhausted, and even if this is not the case, building such large structures, and potentially performing backtracking search over them, can be time consuming.

It has already been demonstrated that using a more fine-grained blocking condition can make the constructed abstractions smaller, resulting in a significant speedup [7]. Even with such a blocking condition, however, the constructed model abstractions can be very large; furthermore, checking such fine-grained conditions can itself be costly.

To address these problems, we propose a new *core blocking* technique. Our technique first employs an easy-to-check and very "aggressive" blocking condition that can halt the model construction much earlier than existing techniques. This condition is so aggressive that, if used alone, it is not necessarily the case that the constructed abstraction can be expanded into a model. Therefore, after a model abstraction has been constructed, a detailed check is performed to ensure that all blocks are indeed valid, and model construction terminates only if all blocks pass this check. Checking blocks for validity can be costly, but it has to be performed relatively rarely, sometimes only once.

We present our technique in the context of the hypertableau calculus [4] and show that the resulting calculus is sound, complete, and terminating. As we discuss in Section 3.4, however, our idea can easily be transferred to standard tableau calculi.

To make our technique effective in practice, one must strike a balance between model expansion and blocking validation: if the core blocking condition is too strict, then it will offer little advantage over standard blocking, and model abstractions may still grow very large; if the condition is not strict enough, then it may stop the construction too early, and the reasoner will spend most of its time in validating blocks. We therefore present several variants of core blocking inspired by our observations about reasoning algorithms for $\mathcal{EL}$-like DLs [8,9], and we present an empirical evaluation using a prototypical implementation of our technique in the HermiT reasoner. The evaluation compares the performance of the hypertableau algorithm employing the original blocking condition and each of the core blocking variants on several widely used ontologies. As might be expected, the effects of core blocking are most pronounced with large and complex ontologies such as DOLCE or GALEN, on which it significantly reduces the sizes of the constructed model abstractions. Furthermore, core blocking allows HermiT to classify an OWL version of the FMA ontology [10], whereas with standard blocking the reasoner runs out or memory. Finally, with simple ontologies such as Wine, core blocking may have little or no effect on the size of abstractions, but it incurs little or no additional cost compared to the standard blocking technique.

2 Preliminaries

We present our results in the context of the hypertableau calculus, which preprocesses a DL knowledge base into a particular clausal form. Therefore, the precise definitions of DLs [1] are not relevant. The basic elements of DL knowledge bases are *atomic concepts*, *atomic roles*, and *individuals*, which correspond to unary predicates, binary predicates, and constants, respectively. Using various *constructors*, one can construct complex *concepts*, which can be transformed into formulae of two-variable first-order logic with counting. For example, the concept $\mathit{Man} \sqcap \exists \mathit{hasChild}.\mathit{Man}$, representing the set of all men with a male child, corresponds to $\mathit{Man}(x) \wedge \exists y : \mathit{hasChild}(x,y) \wedge \mathit{Man}(y)$. Concepts can be used in axioms of the form $C \sqsubseteq D$, which state that the extension of C is contained in the extension of D. For example, $\mathit{Man} \sqsubseteq \mathit{Person}$ expresses the fact that all men are persons. Various DLs allow for other types of axioms, such as statements that a particular role is reflexive, symmetric, or transitive. A TBox $\mathcal{T}$ is a set of axioms. An ABox $\mathcal{A}$ is a set of assertions of the form $C(a)$, $R(a,b)$, $a \approx b$, and $a \not\approx b$, where C is a concept, R is an atomic role, and a and b are individuals. A knowledge base is a pair $\mathcal{K} = (\mathcal{T}, \mathcal{A})$; it is satisfiable if a first-order interpretation satisfying $\mathcal{T}$ and $\mathcal{A}$ exists.

2.1 Model Construction Calculi

Model construction calculi, such as tableau and hypertableau [4], decide the satisfiability of $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ by trying to construct a model of $\mathcal{K}$. To this end, they use *derivation* *rules* that, given an ABox $\mathcal{A}_\ell$, derive an ABox $\mathcal{A}_{\ell+1}$ and thus explicate information implicit in $\mathcal{T}$ and $\mathcal{A}_\ell$. Derivation rules can be nondeterministic—that is, they can derive more than one $\mathcal{A}_{\ell+1}$ from $\mathcal{A}_\ell$. To check the satisfiability of $\mathcal{K}$, model construction calculi construct a *derivation* for $\mathcal{K}$, which is a sequence of ABoxes $\mathcal{A}_0, \ldots, \mathcal{A}_k$ such that $\mathcal{A}_0 = \mathcal{A}$, the ABox $\mathcal{A}_{\ell+1}$ is obtained from $\mathcal{A}_\ell$ by applying a derivation rule for each $0 \leq \ell < k$, and no derivation rule is applicable to $\mathcal{A}_k$. The satisfiability of $\mathcal{K}$ can be decided by checking whether a derivation exists such that $\mathcal{A}_k$ does not contain an obvious contradiction; such an $\mathcal{A}_k$ is called a *pre-model*. Given such an $\mathcal{A}_k$, a model for $\mathcal{K}$ can be constructed by a process called *unraveling* [4].

2.2 The Hypertableau Calculus

The formal definition of the hypertableau calculus is technically involved; therefore, we present here only the aspects that are needed to understand the idea behind core blocking. For further details and the precise definitions, please refer to [4].

The calculus is applicable to a knowledge base $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ expressed in $\mathcal{SROIQ}$ [11]—the DL underpinning OWL 2. The calculus does not operate on $\mathcal{K}$ directly; rather, in order to reduce nondeterminism, it first translates $\mathcal{K}$ into a set of clauses $\mathcal{C}$ and an ABox $\mathcal{A}$. The class of clauses on which the hypertableau calculus operates is called HT-clauses. An HT-clause is an implication of the form $\bigwedge_{i=1}^{m} U_i \rightarrow \bigvee_{j=1}^{n} V_j$, where U_i and V_j are called the *antecedent* and the *consequent* atoms, respectively. Most notably, the atoms in an HT-clause must satisfy the following restrictions, where r is an atomic role, s is a role, A is an atomic concept, and B is a possibly negated atomic concept.

- Each *antecedent* atom is of the form $A(x)$, $r(x, x)$, $r(x, y_i)$, $r(y_i, x)$, $A(y_i)$, or $A(z_j)$.
- Each *consequent* atom is of the form $B(x)$, $\geqslant n\, s.B(x)$, $B(y_i)$, $r(x, x)$, $r(x, y_i)$, $r(y_i, x)$, $r(x, z_j)$, $r(z_j, x)$, $x \approx z_j$, or $y_i \approx y_j$.

These syntactic restrictions reflect the structure of DL axioms and ultimately ensure termination of the calculus. HT-clauses are straightforwardly interpreted in first-order logic, and they intuitively state that at least one consequent atom must be true whenever all atoms in the antecedent are true. We next present the derivation rules of the calculus.

The Hyp-rule is the main derivation rule. For σ a mapping of variables to individuals and U an atom, let $\sigma(U)$ be the atom obtained from U by replacing each variable x with $\sigma(x)$. The Hyp-rule is applicable to an HT-clause $\bigwedge_{i=1}^{m} U_i \rightarrow \bigvee_{j=1}^{n} V_j$ and an ABox $\mathcal{A}_\ell$ if a mapping σ from the variables in the clause to the individuals in $\mathcal{A}_\ell$ exists such that $\sigma(U_i) \in \mathcal{A}_\ell$ for each $1 \leq i \leq m$, but $\sigma(V_j) \notin \mathcal{A}_\ell$ for each $1 \leq j \leq n$. If that is the case, the rule nondeterministically derives $\mathcal{A}_{\ell+1} = \mathcal{A}_\ell \cup \{\sigma(V_j)\}$ for some $1 \leq j \leq n$. For example, when applied to the HT-clause $r(x, y) \rightarrow (\geqslant 1\, r.A)(x) \vee D(y)$ and an ABox $\mathcal{A}_\ell$ containing $r(a, b)$, the Hyp-rule extends $\mathcal{A}_\ell$ with either $(\geqslant 1\, r.A)(a)$ or $D(b)$.

The $\geqslant$-rule deals with existential quantifiers and number restrictions. Intuitively, an assertion $(\geqslant n\, s.B)(a)$ implies the existence of distinct individuals $b_1, \ldots, b_n$ such that role s connects a with each b_i and that $B(b_i)$ holds. Thus, the $\geqslant$-rule is applicable to $(\geqslant n\, s.B)(a)$ in $\mathcal{A}_\ell$ if no individuals $b_1, \ldots, b_n$ exist such that $\mathsf{ar}(s, a, b_i) \in \mathcal{A}_\ell$ and $B(b_i) \in \mathcal{A}_\ell$ for each $1 \leq i \leq n$, and $b_i \not\approx b_j \in \mathcal{A}_\ell$ for each $1 \leq i < j \leq n$, where

$\mathsf{ar}(s,a,b) = s(a,b)$ if s is an atomic role and $\mathsf{ar}(s,a,b) = r(b,a)$ if s is an inverse role such that $s = r^-$. If that is the case, then $\mathcal{A}_\ell$ is extended to $\mathcal{A}_{\ell+1}$ by introducing fresh individuals $c_1, \ldots, c_n$ and adding assertions $\mathsf{ar}(s,a,c_i)$ and $B(c_i)$ for $1 \leq i \leq n$, and $c_i \not\approx c_j$ for $1 \leq i < j \leq n$. The individual a is called a *predecessor* of each c_i; each c_i is called a *successor* of a; and *ancestor* and *descendant* are transitive closures of the predecessor and successor relations, respectively.

The $\approx$-rule deals with equality. Intuitively, an assertion $a \approx b$ states that individuals a and b are equal, so one can be treated as a synonym for the other. The $\approx$-rule is applicable to an assertion $a \approx b$ in $\mathcal{A}_\ell$ if $a \neq b$, in which case it derives $\mathcal{A}_{\ell+1}$ from $\mathcal{A}_\ell$ by a process called *merging*: a is replaced with b in all assertions, certain assertions are removed, and some bookkeeping information is added in order to ensure termination.

The NI-rule deals with certain complications due to an interaction between number restrictions, nominals, and inverse roles. The details of this derivation rule are not relevant for core blocking, so we omit it for the sake of brevity.

The $\bot$-rule detects obvious contradictions. If $\mathcal{A}_\ell$ contains $A(a)$ and $\neg A(a)$, or $a \not\approx a$, the rule derives $\mathcal{A}_{\ell+1} = \mathcal{A}_\ell \cup \{\bot\}$; then $\mathcal{A}_{\ell+1}$ is said to contain a *clash*.

2.3 Blocking

The $\geqslant$-rule is found in virtually all DL model construction calculi. Unrestricted application of the $\geqslant$-rule can lead to the introduction of infinitely many fresh individuals and thus prevent the calculus from terminating. To counteract that, the $\geqslant$-rule is applied to an assertion $(\geqslant n\, r.B)(a)$ only if the individual a is not *blocked*, as described next.

To apply blocking, the individuals are split into two sets. *Root* individuals either occur in the input or are introduced by the NI-rule, and they are never blocked. In contrast, *blockable* individuals are introduced by the $\geqslant$-rule and they can be blocked. Blocking uses labels of an individual and an individual pair defined as follows, where A is an atomic concept and r is an atomic role:

$$\mathcal{L}_\mathcal{A}(s) = \{A \mid A(s) \in \mathcal{A}\} \qquad\qquad \mathcal{L}_\mathcal{A}(s,t) = \{r \mid r(s,t) \in \mathcal{A}\}$$

To prevent cyclic blocks, one needs to pick a strict order $\prec$ over all individuals satisfying certain restrictions. In practice, $\prec$ usually coincides with the order in which individuals are inserted into an ABox.

Pairwise anywhere blocking is necessary for knowledge bases that use inverse roles and number restrictions. Each individual s in an ABox $\mathcal{A}$ is assigned by induction on $\prec$ a status as follows: s is *blocked* if it is directly or indirectly blocked; s is *indirectly blocked* if it has a blocked ancestor; and s is *directly blocked* by an individual t if, for s' and t' the predecessors of s and t, respectively, s, t, s', and t' are all blockable, t is not blocked, $t \prec s$, and (1)–(4) hold.

$$\mathcal{L}_\mathcal{A}(s) = \mathcal{L}_\mathcal{A}(t) \quad (1) \qquad\qquad \mathcal{L}_\mathcal{A}(s') = \mathcal{L}_\mathcal{A}(t') \quad (2)$$
$$\mathcal{L}_\mathcal{A}(s,s') = \mathcal{L}_\mathcal{A}(t,t') \quad (3) \qquad\qquad \mathcal{L}_\mathcal{A}(s',s) = \mathcal{L}_\mathcal{A}(t',t) \quad (4)$$

For an efficient implementation, we build, for each individual, a blocking signature that consist of the four label sets. A hash table containing the blocking signatures for possible blockers can then be used to cheaply look-up a blocker for an unblocked individual before the $\geqslant$-rule is applied.

The simpler *single anywhere blocking* can be used on knowledge bases without inverse roles, and it differs from the above definition in that s is *directly blocked* by an individual t if s and t are blockable, t is not blocked, $t \prec s$, and (1) holds.

A pre-model $\mathcal{A}'$ can be extended to a model for $(\mathcal{A}, \mathcal{C})$ by unraveling. Roughly speaking, each individual s that is directly blocked in $\mathcal{A}'$ by t is replaced by a "copy" of t; a precise account of this process is given in [4].

Tableau algorithms for DLs without inverse roles can use *single subset blocking*, in which s is directly blocked by t if $\mathcal{L}_{\mathcal{A}}(s) \subseteq \mathcal{L}_{\mathcal{A}}(t)$. This can result in smaller pre-models; however, in the hypertableau calculus single subset blocking does not guarantee that a pre-model can be expanded into a model [4].

3 Optimized Blocking Strategies

On complex knowledge bases, reasoners based on model construction calculi may construct very large pre-models. This is one of the main limiting factors in practical DL reasoning [4], as the construction may be time-consuming, and the reasoner may even run out of memory before terminating. Blocking prevents the application of the $\geqslant$-rule and thus constrains the size of pre-models, so some effort has been devoted to detecting situations in which a block can be established earlier than with the standard single or pairwise blocking, but without sacrificing soundness.

For tableau algorithms that normally require pairwise blocking, Horrocks and Sattler proposed a more precise blocking condition [7], which amounts to single subset blocking with additional constraints on the predecessor of the individual that is to be blocked and on the blocker itself. For example, if an individual s with predecessor s' is to be blocked by an individual t, and $\forall r.C$ is in the label of the blocker t, then either s should be not an r^{-}-successor of s' or C should be in the label of s'. Although checking the blocking conditions is quite expensive, the optimization exhibits substantial improvements in reasoning performance due to the significantly smaller pre-models.

Related blocking optimizations were proposed in the context of first-order theorem proving [12]; however, these techniques do not guarantee termination for DLs such as $\mathcal{SROIQ}$ that provide for nominals, number restrictions, and inverse roles.

Caching [13] is an orthogonal approach for reducing the pre-model size by reusing already constructed pre-model fragments. In fact, caching techniques can be used to obtain a worst-case optimal algorithm for certain DLs [14,15]; in contrast, standard (hyper)tableau algorithms are usually not worst-case optimal.

3.1 Core Blocking

Unlike existing blocking techniques, core blocking is approximate rather than exact: applying core blocking alone does not guarantee that a pre-model can indeed be unraveled into a model. To ensure the latter, a pre-model needs to be checked to discover invalid blocks; if such blocks are found, the derivation is continued until either a contradiction is derived or all blocks become valid. The latter condition is satisfied at latest when the standard blocking condition is satisfied, which ensures termination.

To formalize the process of discovering approximate blocks, we assume that each assertion α in an ABox is associated with a Boolean flag that determines whether α is a *core assertion*. A *core blocking policy* will be used to determine which assertions are core. A credulous policy would make no assertions core, thus allowing any individual to potentially block any other individual. While this might generate very small pre-models, it is unlikely to be practicable because most blocks are likely to be invalid, so a reasoner would spend a lot of time in validation. In contrast, a conservative policy would make all assertions core, thus making core blocking exact. In Section 3.3 we present two policies that strike a balance between the potential for reduction in the pre-model size and the cost of validating blocks. Before that, however, we introduce a general notion of core blocking that is applicable to any policy.

Definition 1. *For an ABox $\mathcal{A}$ and a pair of individuals s and t, let*

$$\mathcal{L}^{\mathsf{core}}_{\mathcal{A}}(s) = \{A \mid A \in \mathcal{L}_{\mathcal{A}}(s) \textit{ and } A(s) \textit{ is a core assertion in } \mathcal{A}\} \textit{ and}$$
$$\mathcal{L}^{\mathsf{core}}_{\mathcal{A}}(s,t) = \{r \mid r(s,t) \in \mathcal{L}_{\mathcal{A}}(s,t) \textit{ and } r(s,t) \textit{ is a core assertion in } \mathcal{A}\}.$$

Single *and* pairwise core blocking *are obtained from the respective definitions given in Section 2.3 by using $\mathcal{L}^{\mathsf{core}}_{\mathcal{A}}$ instead of $\mathcal{L}_{\mathcal{A}}$ in conditions* (1)–(4)*; furthermore, in single core blocking, for s to be directly blocked by t we additionally require both s and t to be successors of blockable individuals.*

On knowledge bases that normally require pairwise anywhere blocking (i.e., that contain inverse roles), the model construction from [4] requires individuals s and t involved in direct blocking to be successors of blockable individuals. This is reflected in the notion of single core blocking in Definition 1, which allows single core blocking to be used with knowledge bases that contain inverse roles.

Since core blocking may halt the model expansion too early, we introduce a blocking validation test that checks whether any of the derivation rules would be applicable if we were to unravel a candidate pre-model $\mathcal{A}_\ell$ to a model. To this end, we define an ABox $\mathsf{val}_{\mathcal{A}_\ell}(s)$ for a blockable individual s that, intuitively, contains the assertions from the unraveling of $\mathcal{A}_\ell$ that affect inferences involving s. If no derivation rule is applicable to $\mathsf{val}_{\mathcal{A}_\ell}(s)$, we can conclude that no derivation rule is applicable to the model constructed from $\mathcal{A}_\ell$ as discussed in [4].

Definition 2. *Let $\mathcal{C}$ be a set of HT clauses, and let $\mathcal{A}_\ell$ be an ABox. For an individual w, let $|w| = w$ if w is not blocked in $\mathcal{A}_\ell$, and $|w| = w'$ if w is blocked in $\mathcal{A}_\ell$ by w'. For a blockable individual s, the ABox $\mathsf{val}_{\mathcal{A}_\ell}(s)$ is the union of the sets shown in the following table, where u denotes the predecessor of s, v denotes a successor of $|s|$, b denotes a root individual, C denotes a concept, and r denotes an atomic role.*

1	2	3
$\{C(u) \mid C(u) \in \mathcal{A}_\ell\}$	$\{r(u,s) \mid r(u,s) \in \mathcal{A}_\ell\}$	$\{r(s,u) \mid r(s,u) \in \mathcal{A}_\ell\}$
$\{C(s) \mid C(\lvert s\rvert) \in \mathcal{A}_\ell\}$		
$\{C(v) \mid C(\lvert v\rvert) \in \mathcal{A}_\ell\}$	$\{r(s,v) \mid r(\lvert s\rvert,v) \in \mathcal{A}_\ell\}$	$\{r(v,s) \mid r(v,\lvert s\rvert) \in \mathcal{A}_\ell\}$
$\{C(b) \mid C(b) \in \mathcal{A}_\ell\}$	$\{r(s,b) \mid r(\lvert s\rvert,b) \in \mathcal{A}_\ell\}$	$\{r(b,s) \mid r(b,\lvert s\rvert) \in \mathcal{A}_\ell\}$

A blockable individual s is safe for blocking *in an ABox $\mathcal{A}_\ell$ if the following conditions are satisfied:*

- *the* Hyp*-rule is not applicable to an HT-clause $\gamma \in \mathcal{C}$ and* $\mathsf{val}_{\mathcal{A}_\ell}(s)$ *with a mapping σ such that $\sigma(x) = s$, and*
- *the $\geqslant$-rule is not applicable to an assertion $(\geqslant n\, r.B)(s)$ in* $\mathsf{val}_{\mathcal{A}_\ell}(s)$.

A directly blocked individual s with predecessor s' is validly blocked *in $\mathcal{A}_\ell$ if both s and s' are safe for blocking.*

We finish this section with a note that, on knowledge bases that normally require single blocking (i.e., that do not contain inverse roles), Definitions 1 and 2 can be simplified. By the model construction from [4], $\mathsf{val}_{\mathcal{A}_\ell}(s)$ then needs to contain only sets from columns 1 and 2 in Definition 2; this, in turn, allows us to drop the extra requirement on the predecessors of s and t in Definition 1 in the case of single core blocking.

3.2 Applying Core Blocking in a Derivation

If an individual s is core-blocked by an individual t but the block is identified as invalid, one should reconsider t as a potential blocker for s only after $\mathsf{val}_{\mathcal{A}_\ell}(s)$ changes; otherwise, the calculus might get stuck in an endless loop trying to block s by t and subsequently discovering the block to be invalid. We deal with this problem by associating with each individual s in $\mathcal{A}_\ell$ a Boolean flag $\mathsf{mod}_{\mathcal{A}_\ell}(s)$ that is updated as the derivation progresses. Intuitively, $\mathsf{mod}_{\mathcal{A}_\ell}(s) = \mathsf{true}$ means that $\mathsf{val}_{\mathcal{A}_\ell}(s)$ has changed since the last time blocks were checked for validity. We also maintain a set S of pairs of validly blocked and blocking individuals, which we to ensure that the calculus terminates only when all blocks are valid.

Definition 3. *Let S be a set of pairs of individuals; let $\mathcal{A}_\ell$ be an ABox; and let s and t be individuals occurring in $\mathcal{A}_\ell$. Then, s is directly blocked by t in $\mathcal{A}_\ell$ for S-*core blocking *iff s is directly blocked by t in $\mathcal{A}_\ell$ for core blocking and*

$$\langle s,t\rangle \in S \quad \textit{or} \quad \mathsf{mod}_{\mathcal{A}_\ell}(s) = \mathsf{true} \quad \textit{or} \quad \mathsf{mod}_{\mathcal{A}_\ell}(t) = \mathsf{true}.$$

A derivation by the hypertableau calculus with core blocking *for a set of HT-clauses $\mathcal{C}$ and an ABox $\mathcal{A}$ is constructed by applying the following steps.*

1. *Set $S := \emptyset$, $\mathcal{A}^{\mathsf{a}} := \mathcal{A}$, and* $\mathsf{mod}_{\mathcal{A}^{\mathsf{a}}}(s) := \mathsf{true}$ *for each individual s in $\mathcal{A}^{\mathsf{a}}$.*
2. *Apply the hypertableau calculus exhaustively to $\mathcal{A}^{\mathsf{a}}$ and $\mathcal{C}$ while using S-core blocking in the $\geqslant$-rule; furthermore, whenever $\mathcal{A}_{\ell+1}$ is derived from $\mathcal{A}_\ell$, for each individual s in $\mathcal{A}_{\ell+1}$ set*
   (a) $\mathsf{mod}_{\mathcal{A}_{\ell+1}}(s) := \mathsf{true}$ *if* $\mathsf{val}_{\mathcal{A}_{\ell+1}}(s) \neq \mathsf{val}_{\mathcal{A}_\ell}(s)$ *or if s does not occur in $\mathcal{A}_\ell$, and*
   (b) $\mathsf{mod}_{\mathcal{A}_{\ell+1}}(s) := \mathsf{mod}_{\mathcal{A}_\ell}(s)$ *otherwise.*
   Let $\mathcal{A}^{\mathsf{b}}$ be a resulting ABox to which no derivation rule is applicable.
3. *Set S to be equal to the set of pairs $\langle s,t\rangle$ of individuals such that s is directly blocked in $\mathcal{A}^{\mathsf{b}}$ by t and s is validly blocked in $\mathcal{A}^{\mathsf{b}}$.*
4. *Set* $\mathsf{mod}_{\mathcal{A}^{\mathsf{b}}}(s) := \mathsf{false}$ *for each individual s in $\mathcal{A}^{\mathsf{b}}$.*
5. *If an individual s exists such that s is core blocked in $\mathcal{A}^{\mathsf{b}}$ by t but $\langle s,t\rangle \notin S$, then set $\mathcal{A}^{\mathsf{a}} := \mathcal{A}^{\mathsf{b}}$ and go to Step 2.*
6. *Return $\mathcal{A}^{\mathsf{b}}$.*

Roughly speaking, our algorithm first applies the derivation rules as usual, with the difference that core blocking is used (this is because $S = \emptyset$ in Step 1). After computing a candidate pre-model $\mathcal{A}^{\mathsf{b}}$ in Step 2, in Step 3 the algorithm updates S to the set of pairs of valid blocks, and in Step 4 it marks all individuals in $\mathcal{A}^{\mathsf{b}}$ as not changed. In Step 5, the algorithm checks whether $\mathcal{A}^{\mathsf{b}}$ contains invalid blocks. If that is the case, the process is repeated; but then, S-core blocking ensures that only those blocks are considered that are known to be valid or for which at least one of the individuals has changed since the last validation. Theorem 1 shows that the calculus is sound, complete, and terminating.

Theorem 1. *Let $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ be a $\mathcal{SROIQ}$ knowledge base and $\mathcal{C}$ the set of HT-clauses for $\mathcal{K}$.*

1. *The hypertableau calculus with core blocking terminates.*
2. *If $\mathcal{C}$ and $\mathcal{A}$ are satisfiable, then $\bot \notin \mathcal{A}^{\mathsf{b}}$ for some $\mathcal{A}^{\mathsf{b}}$ computed by the calculus.*
3. *If $\mathcal{C}$ and $\mathcal{A}$ are unsatisfiable, then $\bot \in \mathcal{A}^{\mathsf{b}}$ for each $\mathcal{A}^{\mathsf{b}}$ computed by the calculus.*

Proof (Sketch). For the first claim, assume that $\mathcal{A}^{\mathsf{b}}$ is an ABox computed in Step 2 such that, whenever s is directly blocked in $\mathcal{A}^{\mathsf{b}}$ by t for core blocking, then s is directly blocked in $\mathcal{A}^{\mathsf{b}}$ by t for standard blocking. Each individual s is then validly blocked in $\mathcal{A}^{\mathsf{b}}$, so $\langle s, t\rangle \in S$ at Step 3 and the condition at Step 5 is not satisfied, so the calculus terminates. Thus, in the worst case, core blocking reduces to standard blocking, which implies a bound on the size of $\mathcal{A}^{\mathsf{b}}$ in the usual way [4]. Furthermore, if an individual t does not validly block s in an ABox $\mathcal{A}^{\mathsf{b}}$, then t can be considered again as a blocker for s only after $\mathsf{val}_{\mathcal{A}^{\mathsf{b}}}(s)$ or $\mathsf{val}_{\mathcal{A}^{\mathsf{b}}}(t)$ changes. Since $\mathcal{A}^{\mathsf{b}}$ is bounded in size, $\mathsf{val}_{\mathcal{A}^{\mathsf{b}}}(s)$ and $\mathsf{val}_{\mathcal{A}^{\mathsf{b}}}(t)$ can change only a bounded number of times; hence, t is considered as a candidate blocker for s only a finite number of times, which implies termination.

The second claim holds in the same way as in [4]. Finally, for the third claim, given an ABox $\mathcal{A}^{\mathsf{b}}$ computed by the calculus such that $\bot \notin \mathcal{A}^{\mathsf{b}}$, we unravel $\mathcal{A}^{\mathsf{b}}$ into an interpretation in the standard way [4]. From the definition of unraveling in [4], one can see that, for each blockable individual s, the ABox $\mathsf{val}_{\mathcal{A}^{\mathsf{b}}}(s)$ contains the assertions that correspond to the part of the unraveled interpretation involving s. Since s is validly blocked in $\mathcal{A}^{\mathsf{b}}$, all the relevant restrictions are satisfied for s. Since all blocks are valid in $\mathcal{A}^{\mathsf{b}}$, the unraveled interpretation is a model of $\mathcal{C}$ and $\mathcal{A}$. $\square$

3.3 Core Blocking Policies

We now present two policies for identifying core assertions. Each policy can be used with either single or pairwise core blocking.

The *simple core policy* is inspired by the following observation. Let $\mathcal{A}$ be a potentially infinite ABox obtained by applying the hypertableau calculus without blocking to an $\mathcal{EL}$ knowledge base $\mathcal{K}$, and let s and t be two individuals introduced by applying the $\geqslant$-rule to assertions of the form $(\geqslant n\, r.B)(s')$ and $(\geqslant m\, r'.B)(t')$. Then, $\mathcal{L}_{\mathcal{A}}(s) = \mathcal{L}_{\mathcal{A}}(t)$; in fact, the concept labels $\mathcal{L}_{\mathcal{A}}(s)$ and $\mathcal{L}_{\mathcal{A}}(t)$ depend only on the concept B. The policy thus makes such assertions $B(s)$ and $B(t)$ core in the hope that, if a knowledge base is sufficiently "$\mathcal{EL}$-like," then s would validly block t.

Definition 4. *The* simple core *policy marks all assertions as not core unless they are covered by one of the following rules.*

- *Each assertion $B(c_j)$ derived by applying the $\geqslant$-rule to an assertion of the form $(\geqslant n\, r.B)(a)$ is marked as core.*
- *Each assertion α' derived by the $\approx$-rule from an assertion α via merging is marked as core if and only if α is core.*
- *If an ABox contains α as a noncore assertion but some derivation rule derives α as a core assertion, the former assertion is replaced with the latter.*

Simple core blocking generates very small cores, but it can be imprecise and can therefore lead to frequent validation of blocks. For example, if s and t are individuals introduced by applying the $\geqslant$-rule as above, then inferences involving the predecessor of s can cause the propagation of new concepts to s, which might invalidate blocking. Furthermore, if the knowledge base contains nondeterministic concepts, then nondeterministic inferences involving s and t may cause $\mathcal{L}_{\mathcal{A}}(s)$ and $\mathcal{L}_{\mathcal{A}}(t)$ to diverge, which can also invalidate blocking. We therefore define the following, stronger notion of cores.

Definition 5. *The* complex core *policy is the extension of the simple core policy in which, whenever the* Hyp*-rule derives an assertion $\sigma(V_j)$ using a mapping σ and an HT-clause $\gamma = \bigwedge_{i=1}^{m} U_i \rightarrow \bigvee_{j=1}^{n} V_j$, the assertion $\sigma(V_j)$ is marked as core if and only if $\sigma(V_j)$ is a concept assertion and*

- $n > 1$, *or*
- *$\sigma(V_j)$ is of the form $B(s)$ with s a successor of $\sigma(w)$ for some variable w in γ.*

The complex core policy is motivated by the fact that, when $\mathcal{EL}$-style algorithms are extended to expressive but deterministic DLs such as Horn-$\mathcal{SHIQ}$ [9], the concepts that are propagated to an individual from its predecessor uniquely determine the individual's label, so we mark all such assertions as core.

From the above discussion, one might expect simple core blocking to be exact on HT-clauses obtained from an $\mathcal{EL}$ knowledge base. The knowledge base consisting of axioms (1)–(6), however, shows that this is not the case.

$$A(x) \rightarrow (\geqslant 1\, r.C)(x) \qquad (1) \qquad\qquad t(x,y) \wedge D(y) \rightarrow E(x) \qquad (4)$$
$$B(x) \rightarrow (\geqslant 1\, s.C)(x) \qquad (2) \qquad\qquad B(x) \wedge s(x,y) \wedge E(y) \rightarrow \bot \qquad (5)$$
$$C(x) \rightarrow (\geqslant 1\, t.D)(x) \qquad (3) \qquad\qquad A(a) \quad B(a') \qquad (6)$$

The algorithm initially produces the pre-model shown on the left-hand side of Figure 1, in which b directly core-blocks b' (since there are no inverse roles, it does not matter that a and a' are root individuals). If single core blocking were exact, the algorithm would terminate and declare the knowledge base satisfiable. This block, however, is invalid since the Hyp-rule is applicable to HT-clause (5) and ABox $\mathsf{val}_{\mathcal{A}_\ell}(b')$ shown below (since there are no inverse roles, $\mathsf{val}_{\mathcal{A}_\ell}(b')$ does not contain assertions from column 3 of Definition 2) for $\sigma(x) = a'$ and $\sigma(y) = b'$. Therefore, the model expansion continues as shown on the right-hand side of Figure 1, which ultimately leads to a contradiction.

$$\mathsf{val}_{\mathcal{A}_\ell}(b') = \{\ B(a'),\ \ \geqslant 1\, s.C(a'),\ \ s(a',b'),\ \ C(b'),\ \ \geqslant 1\, t.D(b'),\ \ E(b'),\ \ D(c),\ \ t(b',c),\ \ A(a),\ \ \geqslant 1\, r.C(a)\ \} \qquad (7)$$

$$\begin{tikzcd}
A, \geqslant 1\,r.C \;\; a \arrow[d, "r"'] & a' \;\; B, \geqslant 1\,s.C \arrow[d, "s"] \\
\mathbf{C}, \geqslant 1\,t.D, E \;\; b \arrow[d, "t"'] \arrow[r, dotted, "\text{blocks}"] & b' \;\; \mathbf{C}, \geqslant 1\,t.D \\
\mathbf{D} \;\; c &
\end{tikzcd}
\qquad
\begin{tikzcd}
A, \geqslant 1\,r.C \;\; a \arrow[d, "r"'] & a' \;\; B, \geqslant 1\,s.C, \bot \arrow[d, "s"] \\
\mathbf{C}, \geqslant 1\,t.D, E \;\; b \arrow[d, "t"'] & b' \;\; \mathbf{C}, \geqslant 1\,t.D, E \arrow[d, "t"] \\
\mathbf{D} \;\; c & c' \;\; \mathbf{D}
\end{tikzcd}$$

Fig. 1. Left is the first pre-model for the knowledge base (1)–(6) and single simple core blocking, and right is the final pre-model. Core concepts are shown in bold.

3.4 Applying Core Blocking in Tableau Calculi

Although we presented our idea in the context of the hypertableau calculus, core blocking can be straightforwardly adapted to tableau calculi. This would require adapting the definition of $\mathsf{val}_{\mathcal{A}_\ell}(s)$ to reflect the model construction used in the soundness proof of the calculus, and adapting the notion of safe blocking and Definition 5 to reflect the inference rules of the calculus. For example, most tableau calculi use a $\forall$-rule that from $(\forall r.A)(a)$ and $r(a, b)$ derives $A(b)$. Consequently, an individual a should be considered safe for blocking if the $\forall$-rule is not applicable to an assertion in $\mathsf{val}_{\mathcal{A}_\ell}(a)$ involving a, and $A(b)$ should be made core if b is a successor of a.

4 Empirical Evaluation

We implemented the different core blocking strategies in our HermiT reasoner and carried out a preliminary empirical evaluation. For the evaluation, we selected several ontologies commonly used in practice. We classified each ontology and tested the satisfiability of all concepts from the ontologies with the different blocking strategies. We were mainly interested in the number of individuals in a candidate pre-model, which we expected to be smaller due to core blocking. Since the number of individuals in a pre-model directly relates to the amount of memory required by the reasoner, smaller pre-models can make the difference between being able to process an ontology or not.

We conducted our tests on a 2.6 GHz Windows 7 Desktop machine with 8 GB of RAM. We used Java 1.6 allowing for 1 GB of heap space in each test. All tested ontologies, a version of HermiT that supports core blocking, and Excel spreadsheets containing test results are available online.[1]

Figures 2–6 contain concepts on the x-axis; however, concept names are not shown due to the high number of concepts. The concepts are ordered according to the performance of the standard blocking strategy reasoner. The y-axis either displays the number of individuals in the pre-models or the reasoning times in milliseconds. All reasoning times exclude loading and preprocessing times, since these are independent of the blocking strategy. Some figures employ a logarithmic scale to improve readability. The label *standard pairwise* refers to the standard pairwise anywhere blocking strategy, *complex pairwise* refers to pairwise core blocking with the complex core policy, etc.

Tables 1–3 show average measurements taken while testing the satisfiability of all concepts in an ontology. The meaning of various rows is as follows: *final pre-model size* shows the average number of individuals in the final pre-model; *finally blocked* shows

[1] `http://www.hermit-reasoner.com/coreBlocking.html`

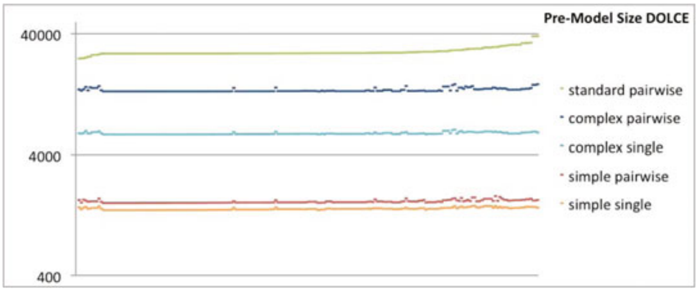


Fig. 2. The number of individuals in the pre-models for all concepts in DOLCE

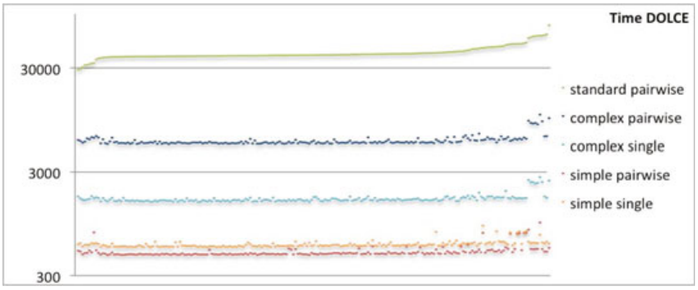


Fig. 3. The reasoning times in ms for testing the satisfiability of all concepts in DOLCE

the average number of blocked individuals in the final pre-model; *number of validations* shows the average number of validations before a pre-model was found in which all blocks are valid; *time in ms* shows the average time to test concept satisfiability; and *validation part* shows the percentage of this time taken to validate blocks. Finally, all tables show the time needed to classify the ontology in the format *hours:minutes:seconds*.

DOLCE is a small but complex $\mathcal{SHOIQ}(\mathcal{D})$ ontology containing 209 concepts and 1,537 axioms that produce 2,325 HT-clauses. Core blocking works particularly well on DOLCE. The pre-model sizes (see Figure 2) and the reasoning times (see Figure 3) for all core blocking variants are consistently below those obtained with the standard anywhere blocking strategy. The simple single core blocking strategy gives the smallest pre-models but the reasoning times are slightly smaller for the simple pairwise strategy. This is because the simple single strategy produces more invalid blocks and, consequently, requires more expansion and (expensive) validation cycles before a final pre-model is found (see Table 1). Overall, the strategies work very well because DOLCE does not seem to be highly constrained and many blocks are valid immediately.

Wine has often been used to demonstrate various DL features. The ontology contains 139 concepts and 393 $\mathcal{SHOIN}(\mathcal{D})$ axioms that are converted to 627 HT-clauses. State of the art reasoners can routinely process the ontology. Satisfiability for almost all concepts can be checked using pre-models with very few blocks and almost all blocks are valid immediately, so we have not provided a figure showing the pre-model sizes.

Table 1. Average measurements over all concepts in DOLCE and the classification time

	standard pairwise	complex pairwise	complex single	simple pairwise	simple single
final pre-model size	28,310	13,583	5,942	1,634	1,426
finally blocked	19,319	9,341	4,241	1,207	1,046
number of validations	—	1.03	1.06	1.09	2.09
time in ms	41,821	5,970	1,663	511	601
validation part	—	2.17%	3.98%	48.84%	65.63%
classification time	01:18:32	00:24:03	00:08:43	00:03:45	00:05:29

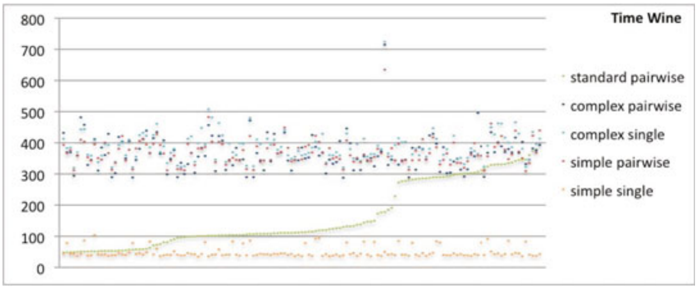


Fig. 4. The reasoning times for testing the satisfiability of all concepts in the Wine ontology

Different blocking strategies (cf. Figure 4 and Table 2) exhibit little difference in performance, and slight variations can be attributed to Java's inconsistent runtime behavior.

GALEN is the original version of the GALEN medical ontology dating from about 10 years ago. Apart from CB [9], which implements an extension of an $\mathcal{EL}$-style algorithm to Horn-$\mathcal{SHIQ}$ [9], HermiT is currently the only reasoner that can classify this ontology. GALEN is a Horn-$\mathcal{SHIF}$ ontology containing 2,748 concepts and 4,979 axioms that produce 8,189 HT-clauses, and it normally requires pairwise blocking. GALEN is unusual in that it contains 2,256 "easy" concepts that are satisfied in very small pre-models ($<$ 200 individuals) and 492 "hard" concepts that are satisfied in very large pre-models ($>$ 35,000 individuals) for the standard blocking strategy. The classification times in Table 3 take all concepts into account; in all other cases we omit the measurements for the "easy" concepts since they do not show much difference between the different blocking strategies and just clutter the presentation. As for DOLCE, the simple single core blocking strategy produces the most significant reduction in model size (see Figure 5). Although this strategy requires the most validation rounds, and these take up 86% of the overall reasoning time, this strategy is still the fastest (see Figure 6) since the reduction in model sizes compensates for the expensive block validations.

The only optimization in HermiT that needs adapting in order to work with core blocking is the *blocking cache*: once a pre-model for a concept is constructed, parts of the pre-model are reused in the remaining subsumption tests [4]. This dramatically reduces the overall classification time. The blocking cache can only be used on ontologies without nominals; in out test suite only GALEN falls into that category. Although the blocking cache could in principle be adapted for use with core blocking, this has not yet been implemented, so we switched this optimization off.

Table 2. Average measurements over all concepts in Wine and the classification time

	standard pairwise	complex pairwise	complex single	simple pairwise	simple single
final pre-model size	204.56	204.56	204.56	204.56	219.40
finally blocked	0	0	0	0	0
number of validations	—	1.00	1.00	1.01	1.01
time in ms	108	152	154	141	168
validation part	—	0.00%	0.01%	0.01%	0.06%
classification time	00:00:38	00:00:41	00:00:42	00:00:40	00:00:41

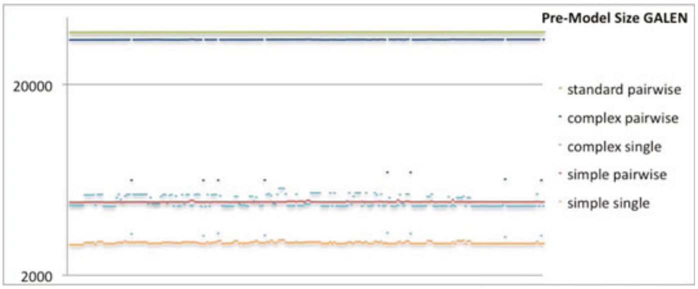


Fig. 5. The number of individuals in the pre-models for the (hard) concepts in GALEN

The foundational model of anatomy (FMA) is a domain ontology about human anatomy [10]. The ontology is one of the largest OWL ontologies available, containing 41,648 concepts and 122,617 $\mathcal{ALCOIF}(\mathcal{D})$ axioms, and it is transformed into 125,346 HT-clauses and 3,740 ABox assertions. We initially tried to take the same measurements for FMA as for the other ontologies; however, after 20 hours we processed only about 10% of the concepts (5,288 out of 41,648), so we aborted the test. Only the single simple core blocking strategy was able to process all 5,288 concepts. The pairwise simple core strategy stayed within the memory limit, but it was significantly slower and it suffered from 5 timeouts due to our imposition of a 2 minute time limit per concept. The standard blocking strategy exceeded either the memory or the time limit on 56 concepts, the pairwise complex core strategy on 70, and the single complex core strategy on 37 concepts. Therefore, we produced complete measurements only with single simple core blocking, using which HermiT was able to classify the entire ontology in about 5.5 hours, discovering 33,431 unsatisfiable concepts. The ontology thus seems to contain modeling errors that went undetected so far due to lack of adequate tool support. The unsatisfiability of all of these concepts was detected before blocking validation was required. The sizes of the ABoxes constructed while processing unsatisfiable concepts is included in the final pre-model size in Table 4, although these are not strictly pre-models since they contain a clash.

We also tested how much memory is necessary to construct all pre-models for DOLCE and GALEN under different blocking strategies. Starting with 16MB, we doubled the memory until the tested strategy could build all pre-models. The simple and complex core blocking strategies require as little as 64MB and 128MB of memory, respectively, whereas the standard blocking technique requires 512MB.

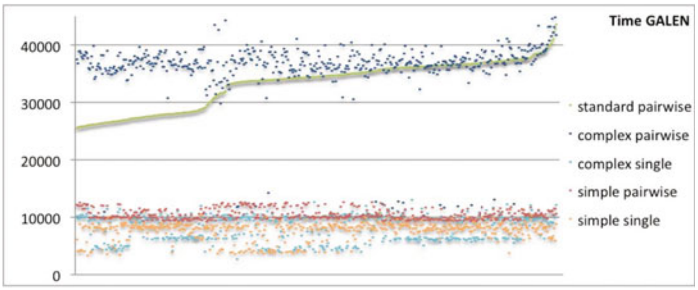


Fig. 6. The reasoning times for testing the satisfiability of the (hard) concepts in GALEN

Table 3. Average measurements over (hard) concepts in GALEN and the classification time

	standard pairwise	complex pairwise	complex single	simple pairwise	simple single
final pre-model size	37,747	33,557	4,876	4,869	2,975
finally blocked	19,290	19,726	2,234	1,896	1,247
number of validations	—	9.18	12.65	8.87	13.91
time in ms	33,293	36,213	8,050	10,485	7,608
validation part	—	27.47%	74.91%	81.47%	86.78%
classification time	03:50:01	04:35:12	01:07:18	01:27:50	01:02:44

Table 4. Average measurements over FMA with the single simple core strategy

final pre-model size	1,747	finally blocked	1,074
time in ms	518	validation part	0.00%
number of validations	0.2	classification time	05:31:23

5 Discussion

In this paper we presented several novel blocking strategies that can improve the performance of DL reasoners by significantly reducing the size of the pre-models generated during satisfiability tests. Although we expected complex core blocking to work better on knowledge bases in expressive DLs, the evaluation shows that the simple core policy clearly outperforms the complex core policy regarding space and time on all tested ontologies. On more complex ontologies, the memory requirement with core blocking seems to decrease significantly.

On ontologies such as Wine where very few individuals are blocked, the new strategies cannot really reduce the sizes of the pre-models; however, they do not seem to have a negative effect on the reasoning times either. On more complex ontologies, the memory requirement with core blocking seems to decrease significantly.

The current publicly available version of HermiT (1.2.2) uses simple single core blocking as its default blocking strategy for ontologies with nominals; for ontologies without nominals it uses standard anywhere blocking with the blocking cache optimization, an optimization that has not yet been extended to core blocking.

Blocking validation is not highly optimized in our prototypical implementation. This is most apparent for the single simple core strategy that causes the most invalid blocks and where block validation takes 86% of the time for GALEN. Only the significant model size reductions allows this strategy to nevertheless be the fastest. We believe that we can significantly improve the performance in the future. We identified the two most common reasons for invalid blocks: the $\geqslant$-rule is applicable to an assertion from $\mathsf{val}_{\mathcal{A}_\ell}(s)$ of a blocked individual, or the Hyp-rule is applicable to the assertions from the temporary ABox of the predecessor of a directly blocked individual. Testing for these two cases first should reduce the overall time of validity tests. Finally, we shall adapt the blocking cache technique to core blocking.

Acknowledgements. The presented work is funded by the EPSRC project HermiT: Reasoning with Large Ontologies.

References

1. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F.: The Description Logic Handbook. Cambridge University Press, Cambridge (2003)
2. Motik, B., Patel-Schneider, P.F., Parsia, B.: OWL 2 web ontology language document overview (2009), `http://www.w3.org/TR/owl2-overview/`
3. Tsarkov, D., Horrocks, I.: FaCT++ description logic reasoner: System description. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 292–297. Springer, Heidelberg (2006)
4. Motik, B., Shearer, R., Horrocks, I.: Hypertableau reasoning for description logics. J. of Artificial Intelligence Research 173(14), 1275–1309 (2009)
5. Sirin, E., Parsia, B., Cuenca Grau, B., Kalyanpur, A., Katz, Y.: Pellet: A practical OWL-DL reasoner. J. of Web Semantics 5(2) (2007)
6. Haarslev, V., Möller, R.: Description of the RACER system and its applications. In: Proc. of DL-01 (2001)
7. Horrocks, I., Sattler, U.: Optimised reasoning for $\mathcal{SHIQ}$. In: Proc. of ECAI-02, pp. 277–281 (2002)
8. Baader, F., Brandt, S., Lutz, C.: Pushing the $\mathcal{EL}$ envelope. In: : Proc. of IJCAI-05, vol. 19, pp. 364–369 (2005)
9. Kazakov, Y.: Consequence-driven reasoning for horn SHIQ ontologies. In: : Proc. of IJCAI-09, pp. 2040–2045 (2009)
10. Golbreich, C., Zhang, S., Bodenreider, O.: The foundational model of anatomy in owl: Experience and perspectives. J. of Web Semantics: Science, Services and Agents on the World Wide Web 4(3), 181–195 (2006)
11. Horrocks, I., Kutz, O., Sattler, U.: The even more irresistible $\mathcal{SROIQ}$. In: : Proc. of KR-06, pp. 57–67 (2006)
12. Baumgartner, P., Schmidt, R.A.: Blocking and other enhancements for bottom-up model generation methods. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 125–139. Springer, Heidelberg (2006)
13. Ding, Y., Haarslev, V.: Tableau caching for description logics with inverse and transitive roles. In: Proc. of DL-06 (2006)
14. Goré, R., Widmann, F.: Sound global state caching for *ALC* with inverse roles. In: Giese, M., Waaler, A. (eds.) TABLEAUX 2009. LNCS, vol. 5607, pp. 205–219. Springer, Heidelberg (2009)
15. Donini, F.M., Massacci, F.: EXPTIME tableaux for $\mathcal{ALC}$. Artificial Intelligence Journal 124(1), 87–138 (2000)

An Extension of Complex Role Inclusion Axioms in the Description Logic $\mathcal{SROIQ}$

Yevgeny Kazakov

Oxford University Computing Laboratory

Abstract. We propose an extension of the syntactic restriction for complex role inclusion axioms in the description logic $\mathcal{SROIQ}$. Like the original restriction in $\mathcal{SROIQ}$, our restrictions can be checked in polynomial time and they guarantee regularity for the sets of role chains implying roles, and thereby decidability for the main reasoning problems. But unlike the original restrictions, our syntactic restrictions can represent any regular compositional properties on roles. In particular, many practically relevant complex role inclusion axioms, such as those describing various parthood relations, can be expressed in our extension, but could not be expressed in the original $\mathcal{SROIQ}$.

1 Introduction

The description logic (DL) $\mathcal{SROIQ}$ [11] provides a logical foundation for the new version of the web ontology language OWL 2.[1] In comparison to the DL $\mathcal{SHOIN}$ which underpins the first version of OWL,[2] $\mathcal{SROIQ}$ provides several new constructors for classes and axioms. One of the new powerful features of $\mathcal{SROIQ}$ are so-called complex role inclusion axioms (RIAs) which allow for expressing implications between role chains and roles: $R_1 \cdots R_n \sqsubseteq R$. It is well-known that unrestricted usage of such axioms can easily lead to undecidability for modal and description logics [6,8,9,12], with a notable exception of the DL $\mathcal{EL}^{++}$ [2]. Therefore, certain syntactic restrictions are imposed on RIAs in $\mathcal{SROIQ}$ to regain decidability. Specifically, the restrictions ensure that RIAs $R_1 \cdots R_n \sqsubseteq R$ when viewed as production rules for context-free grammars $R \to R_1 \ldots R_n$ induce a regular language. In fact, the reasoning procedures for $\mathcal{SROIQ}$ [11,13] do not use the syntactic restrictions directly, but take as an input the resulting non-deterministic finite automata for RIAs. This means that it is possible to use exactly the same procedure for any set of RIAs for which the corresponding regular automata can be constructed.

Unfortunately, the syntactic restrictions on RIAs in $\mathcal{SROIQ}$ are not satisfied in all cases when the language induced by the RIAs is regular. In this paper we analyze several common use cases of RIAs which correspond to regular languages but cannot be expressed within $\mathcal{SROIQ}$. To extend the expressive power of RIAs, we introduce a notion of stratified set of RIAs and demonstrate that it can be used to express the required axioms. Our restrictions have several nice properties, which could allow for their seamless integration into future revisions of OWL:

[1] http://www.w3.org/TR/owl2-overview/

[2] http://www.w3.org/TR/owl-ref/

Table 1. The syntax and semantics of $\mathcal{SROIQ}$

Name	Syntax	Semantics
Concepts		
atomic concept	A	$A^{\mathcal{I}}$ (given)
nominal	$\{a\}$	$\{a^{\mathcal{I}}\}$
top concept	$\top$	$\Delta^{\mathcal{I}}$
negation	$\neg C$	$\Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$
conjunction	$C \sqcap D$	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$
existential restriction	$\exists R.C$	$\{x \mid R^{\mathcal{I}}(x, C^{\mathcal{I}}) \neq \emptyset\}$
min cardinality	$\geqslant nS.C$	$\{x \mid \|S^{\mathcal{I}}(x, C^{\mathcal{I}})\| \geq n\}$
exists self	$\exists S.\mathsf{Self}$	$\{x \mid \langle x, x\rangle \in S^{\mathcal{I}}\}$
Axioms		
complex role inclusion	$\boldsymbol{\rho \sqsubseteq R}$	$\boldsymbol{\rho^{\mathcal{I}} \subseteq R^{\mathcal{I}}}$
disjoint roles	$\mathsf{Disj}(S_1, S_2)$	$S_1^{\mathcal{I}} \cap S_2^{\mathcal{I}} = \emptyset$
concept inclusion	$C \sqsubseteq D$	$C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$
concept assertion	$C(a)$	$a^{\mathcal{I}} \in C^{\mathcal{I}}$
role assertion	$R(a, b)$	$\langle a, b\rangle \in R^{\mathcal{I}}$

1. Our restrictions are *conservative* over the current restrictions in $\mathcal{SROIQ}$. That is, every set of RIAs that satisfies the current restriction in $\mathcal{SROIQ}$ will automatically satisfy our restrictions.
2. Our restrictions are *tractable*, that is, they can be verified in polynomial time in the size of the input set of RIAs.
3. Our restrictions are *constructive*, which means that there is a procedure that builds the corresponding regular automaton for every set of RIAs that satisfies our restrictions.
4. Finally, unlike the original restrictions in $\mathcal{SROIQ}$, our restrictions are *complete* w.r.t. regular compositional properties. This means that any regular compositional properties on roles can be expressed using a stratified set of RIAs.

2 Preliminaries

In this section we introduce syntax and semantics of the DL $\mathcal{SROIQ}$ [11]. A $\mathcal{SROIQ}$ *vocabulary* consists of countably infinite sets N_C of *atomic concepts*, N_R of *atomic roles*, and N_I of *individuals*. A $\mathcal{SROIQ}$ *role* is either $r \in \mathsf{N}_R$, an *inverse role* r^- with $r \in \mathsf{N}_R$, or the *universal role* U. A *role chain* is a sequence of roles $\rho = R_1 \cdots R_n$, $n \geq 0$, where $R_i \neq U$, $1 \leq i \leq n$; in this case we denote by $\|\rho\| := n$ the *size* of ρ; when $n = 0$, ρ is called the *empty role chain* and is denoted by ϵ. With $\rho_1\rho_2$ we denote the *concatenation* of role chains ρ_1 and ρ_2, and with ρR ($R\rho$) we denote the role chain obtained by appending (prepending) R to ρ. We denote by $\mathsf{Inv}(R)$ the *inverse of a role* R defined by $\mathsf{Inv}(R) := r^-$ when $R = r$, $\mathsf{Inv}(R) := r$ when $R = r^-$, and $\mathsf{Inv}(R) := U$ when $R = U$. The *inverse of a role chain* $\rho = R_1 \cdots R_n$ is a role chain $\mathsf{Inv}(\rho) := \mathsf{Inv}(R_n) \cdots \mathsf{Inv}(R_1)$.

The syntax and semantics of $\mathcal{SROIQ}$ is summarized in Table 1. The set of $\mathcal{SROIQ}$ concepts is recursively defined using the constructors in the upper part of the table, where $A \in \mathsf{N}_C$, C, D are concepts, R, S roles, a an individual, and n a positive integer.

A *regular order on roles* is an irreflexive transitive binary relation $\prec$ on roles such that $R_1 \prec R_2$ iff $\mathsf{Inv}(R_1) \prec R_2$. A (complex) *role inclusion axiom* (RIA) $R_1 \cdots R_n \sqsubseteq R$ is said to be $\prec$*-regular*, if either: (i) $n = 2$ and $R_1 = R_2 = R$, or (ii) $n = 1$ and $R_1 = \mathsf{Inv}(R)$, or (iii) $R_i \prec R$ for $1 \leq i \leq n$, or (iv) $R_1 = R$ and $R_i \prec R$ for $1 < i \leq n$, or (v) $R_n = R$ and $R_i \prec R$ for $1 \leq i < n$. A set $\mathcal{R}$ of RIAs is $\prec$*-regular* if every RIA in $\mathcal{R}$ is $\prec$-regular.

A $\mathcal{SROIQ}$ *ontology* is a set $\mathcal{O}$ of axioms listed in the lower part of Table 1, where ρ is a role chain, $R_{(i)}$ and $S_{(i)}$ are roles, C, D concepts, and a, b individuals, such that the set of all RIAs in $\mathcal{O}$ is $\prec$-regular for some regular order $\prec$ on roles.

For a RIA $\alpha = (\rho \sqsubseteq R)$ and role chains ρ' and ρ'', we write $\rho' \sqsubseteq_\alpha \rho''$ if $\rho' = \rho'_1 \rho \rho'_2$ and $\rho'' = \rho'_1 R \rho'_2$ for some ρ'_1 and ρ'_2. To indicate a position where α was used, we also write $\rho' \sqsubseteq_{\alpha,k} \rho''$ where $k = \|\rho'_1\|$. For a set of RIAs $\mathcal{R}$, we write $\rho' \sqsubseteq_\mathcal{R} \rho''$ ($\rho' \sqsubseteq_{\mathcal{R},k} \rho''$) if $\rho' \sqsubseteq_\alpha \rho''$ ($\rho' \sqsubseteq_{\alpha,k} \rho''$) for some $\alpha \in \mathcal{R}$. We denote by $\sqsubseteq^*_\mathcal{R}$ ($\sqsubseteq^*_{\mathcal{R},k}$) the reflexive transitive closure of $\sqsubseteq_\mathcal{R}$ ($\sqsubseteq_{\mathcal{R},k}$). The sequence $\rho_0 \sqsubseteq_{\alpha_1} \rho_1 \cdots \sqsubseteq_{\alpha_n} \rho_n$ ($\rho_0 \sqsubseteq_{\alpha_1,k_1} \rho_1 \cdots \sqsubseteq_{\alpha_n,k_n} \rho_n$), $n \geq 0$, $\alpha_i \in \mathcal{R}$ $(1 \leq i \leq n)$ is called a *proof for* $\rho_0 \sqsubseteq \rho_n$ *in* $\mathcal{R}$. In this case we also say that $\rho_0 \sqsubseteq \rho_n$ *is provable in* $\mathcal{R}$.

We denote by $\bar{\mathcal{R}}$ the extension of $\mathcal{R}$ with *inverses* $\mathsf{Inv}(\rho) \sqsubseteq \mathsf{Inv}(R)$ of RIAs $\rho \sqsubseteq R \in \mathcal{R}$. Let $\mathcal{O}$ be a $\mathcal{SROIQ}$ ontology and $\mathcal{R}$ the set of RIAs in $\mathcal{O}$. A role S is *simple* if $\rho \sqsubseteq^*_{\bar{\mathcal{R}}} S$ implies $\|\rho\| \leq 1$. It is required that all roles $S_{(i)}$ in Table 1 are simple w.r.t. $\mathcal{R}$. Other constructors of $\mathcal{SROIQ}$ [11] can be expressed using those in Table 1. The *bottom concept* $\bot$ stands for $\neg\top$, *disjunction* $C \sqcup D$ for $\neg(\neg C \sqcap \neg D)$, *universal restriction* $\forall R.C$ for $\neg(\exists R.\neg C)$, *max cardinality* $\leqslant nS.C$ for $\neg(\geqslant(n+1)S.C)$, *role transitivity* $\mathsf{Tra}(S)$ for $S \cdot S \sqsubseteq S$, *role reflexivity* $\mathsf{Ref}(R)$ for $\epsilon \sqsubseteq R$, *role symmetry* $\mathsf{Sym}(R)$ for $\mathsf{Inv}(R) \sqsubseteq R$, *role irreflexivity* $\mathsf{Irr}(S)$ for $\exists S.\mathsf{Self} \sqsubseteq \bot$, *role asymmetry* $\mathsf{Asy}(S)$ for $\mathsf{Disj}(S, \mathsf{Inv}(S))$, *concept equivalence* $C \equiv D$ for $C \sqsubseteq D$ and $D \sqsubseteq C$, and *negative role assertion* $\neg S(a,b)$ for $S_S(a,b)$ and $\mathsf{Disj}(S, S_S)$, where S_S is a fresh (simple) role for S. Of all constructors and axioms, only RIAs are of a primary focus in this paper.

The semantics of $\mathcal{SROIQ}$ is defined using interpretations. An *interpretation* is a pair $\mathcal{I} = (\Delta^\mathcal{I}, \cdot^\mathcal{I})$ where $\Delta^\mathcal{I}$ is a non-empty set called *the domain of the interpretation* and $\cdot^\mathcal{I}$ is the *interpretation function*, which assigns to every $A \in \mathsf{N}_C$ a set $A^\mathcal{I} \subseteq \Delta^\mathcal{I}$, to every $r \in \mathsf{N}_R$ a relation $r^\mathcal{I} \in \Delta^\mathcal{I} \times \Delta^\mathcal{I}$, and to every $a \in \mathsf{N}_I$ an element $a^\mathcal{I} \in \Delta^\mathcal{I}$. $\mathcal{I}$ is extended to roles by $U^\mathcal{I} := \Delta^\mathcal{I} \times \Delta^\mathcal{I}$ and $(r^-)^\mathcal{I} := \{\langle x,y\rangle \mid \langle y,x\rangle \in r^\mathcal{I}\}$, and to role chains by $(R_1 \cdots R_n)^\mathcal{I} := R_1^\mathcal{I} \circ \cdots \circ R_n^\mathcal{I}$ where $\circ$ is the composition of binary relations. The empty role chain ϵ is interpreted by $\epsilon^\mathcal{I} := \{\langle x,x\rangle \mid x \in \Delta^\mathcal{I}\}$.

The interpretation of concepts is defined according to the right column of the upper part of Table 1, where $\delta(x,V)$ for $\delta \subseteq \Delta^\mathcal{I} \times \Delta^\mathcal{I}$, $V \subseteq \Delta^\mathcal{I}$, and $x \in \Delta^\mathcal{I}$ denotes the set $\{y \mid \langle x,y\rangle \in \delta \wedge y \in V\}$, and $\|V\|$ denotes the cardinality of a set $V \subseteq \Delta^\mathcal{I}$. An interpretation $\mathcal{I}$ *satisfies an axiom* α (written $\mathcal{I} \models \alpha$) if the respective condition to the right of the axiom in Table 1 holds; $\mathcal{I}$ is a *model of an ontology* $\mathcal{O}$ (written $\mathcal{I} \models \mathcal{O}$) if $\mathcal{I}$ satisfies every axiom in $\mathcal{O}$. We say that α is a *(logical) consequence* of $\mathcal{O}$ or *is entailed* by $\mathcal{O}$ (written $\mathcal{O} \models \alpha$) if every model of $\mathcal{O}$ satisfies α.

3 Regularity for Sets of Role Inclusion Axioms

Given a set of RIAs $\mathcal{R}$, for every role R, define the following language $L_{\mathcal{R}}(R)$ of role chains (viewed as words over roles):

$$L_{\mathcal{R}}(R) := \{\rho \mid \rho \sqsubseteq^*_{\mathcal{R}} R\} \tag{1}$$

We say that $\mathcal{R}$ is *regular* if the language $L_{\mathcal{R}}(R)$ is regular for every role R. It has been shown [12] that $\prec$-regularity for $\mathcal{R}$ implies regularity for $\bar{\mathcal{R}}$. The converse of this property, however, does not always hold, as demonstrated in the following example.

Consider the following set $\mathcal{R}$ of RIAs:

$$\mathsf{isProperPartOf} \sqsubseteq \mathsf{isPartOf} \tag{2}$$
$$\mathsf{isPartOf} \cdot \mathsf{isPartOf} \sqsubseteq \mathsf{isPartOf} \tag{3}$$
$$\mathsf{isPartOf} \cdot \mathsf{isProperPartOf} \sqsubseteq \mathsf{isProperPartOf} \tag{4}$$

$\mathcal{R}$ expresses properties of parthood relations $\mathsf{isPartOf}$ and $\mathsf{isProperPartOf}$: RIA (2) says that $\mathsf{isProperPartOf}$ is a sub-relation of $\mathsf{isPartOf}$; RIA (3) says that $\mathsf{isPartOf}$ is transitive; RIA (4) says that if x is a part of y which is a proper part of z, then x is a proper part of z. Since any role chain consisting of $\mathsf{isPartOf}$ and $\mathsf{isProperPartOf}$ can be reduced using (2) and (3) to $\mathsf{isPartOf}$, it is easy to see that:

$$L_{\bar{\mathcal{R}}}(\mathsf{isPartOf}) = (\mathsf{isPartOf} \mid \mathsf{isProperPartOf})^+ \tag{5}$$

Since $\mathsf{isProperPartOf}$ is only implied by (4), we also have:

$$L_{\bar{\mathcal{R}}}(\mathsf{isProperPartOf}) = (\mathsf{isPartOf}^* \cdot \mathsf{isProperPartOf})^+ \tag{6}$$

Thus, the languages (5) and (6) induced by RIAs (2)–(4) are regular. However, there is no order $\prec$ for which RIAs (2)–(4) are $\prec$-regular. Indeed, by conditions (i)–(v) of $\prec$-regularity, it follows from (2) that $\mathsf{isProperPartOf} \prec \mathsf{isPartOf}$, and from (4) that $\mathsf{isPartOf} \prec \mathsf{isProperPartOf}$, which is not possible if $\prec$ is a transitive irreflexive relation.

In fact, there is no set of RIAs $\mathcal{R}$, possibly with additional roles, that could express properties (2)–(4) using only $\prec$-regular RIAs. It is easy to show by induction over the definition of $\sqsubseteq^*_{\mathcal{R}}$ that if the RIAs of $\mathcal{R}$ are $\prec$-regular, then $R_1 \cdots R_n \sqsubseteq^*_{\bar{\mathcal{R}}} R$ implies that for every i with $1 \leq i \leq n$, either $R_i = R$, or $R_i = \mathsf{Inv}(R)$, or $R_i \prec R$. This means that for every role R, the language $L_{\bar{\mathcal{R}}}(R)$ contains only words over R, $\mathsf{Inv}(R)$, or R' with $R' \prec R$. Clearly, this is not possible if $L_{\bar{\mathcal{R}}}(\mathsf{isPartOf})$ and $L_{\bar{\mathcal{R}}}(\mathsf{isProperPartOf})$ are extensions of the languages defined in (5) and (6).

Axioms such as (2)–(4) naturally appear in ontologies describing parthood relations, such as those between anatomical parts of the human body. For example, release 7 of the GRAIL version of the OpenGALEN ontology[3] contains the following axioms, which are analogous to (2)–(4):

$$\mathsf{isNonPartitivelyContainedIn} \sqsubseteq \mathsf{isContainedIn} \tag{7}$$
$$\mathsf{isContainedIn} \cdot \mathsf{isContainedIn} \sqsubseteq \mathsf{isContainedIn} \tag{8}$$
$$\mathsf{isNonPartitivelyContainedIn} \cdot \mathsf{isContainedIn} \sqsubseteq \mathsf{isNonPartitivelyContainedIn} \tag{9}$$

[3] http://www.opengalen.org/

Complex RIAs such as (7)–(9) are used in OpenGALEN to propagate properties over chains of various parthood relations. For example, the next axiom taken from OpenGALEN expresses that every instance of body structure contained in spinal canal is a structural component of nervous system:

$$\begin{aligned}&\mathsf{BodyStructure} \sqcap \exists \mathsf{isContainedIn.SpinalCanal}\\ &\qquad\qquad \sqsubseteq \exists \mathsf{isStructuralComponentOf.NervousSystem}\end{aligned} \tag{10}$$

Recently, complex RIAs over parthood relations have been proposed as an alternative to SEP-triplet encoding [19]. The SEP-triplet encoding was introduced [17] as a technique to enable the propagation of some properties over parthood relations, while ensuring that other properties are not propagated. For example, if a finger is defined as part of a hand, then any injury to a finger should be classified as an injury to the hand, however, the amputation of a finger should not be classified as an amputation of the hand. The proposed new encoding makes use of complex RIAs such as (2)–(4) to express propagation properties. For example, propagation of the injury property over the proper-part relation can be expressed using the following RIA:

$$\mathsf{isInjuryOf} \cdot \mathsf{isProperPartOf} \sqsubseteq \mathsf{isInjuryOf}. \tag{11}$$

It was argued that the usage of complex RIAs can eliminate many potential problems with the existing SEP-triplet encoding, used, e.g., in SNOMED CT[4] and can dramatically reduce the size of the ontology. However, since RIAs (2)–(4) do not satisfy $\prec$-regularity, this technique is currently limited to $\mathcal{EL}^{++}$ ontologies where $\prec$-regularity is not required, and can be problematic when an expressivity beyond $\mathcal{EL}^{++}$ is required, such as for translating OpenGALEN into OWL 2. In this paper we propose an extension of regularity conditions, which, in particular, can handle axioms such as (2)–(4).

4 Stratified Sets of Role Inclusion Axioms and Regularity

As can be seen from example RIAs (2)–(4), one limitation of $\prec$-regularity is that it cannot deal with cyclic dependencies on roles. Our first step is to relax this requirement by considering arbitrary (i.e., not necessarily strict) orders on roles.

Definition 1. *Let $\precsim$ be a preorder (a transitive reflexive relation) on roles. We write $R_1 \approx R_2$ if $R_1 \precsim R_2$ and $R_2 \precsim R_1$, and $R_1 \prec R_2$ if $R_1 \precsim R_2$ and $R_2 \not\precsim R_1$. The* level $l_{\precsim}(R)$ of R *w.r.t. $\precsim$ is the largest n such that there exists roles $R_1 \prec R_2 \prec \cdots \prec R_n \prec R$. We say that a RIA $R_1 \cdots R_n \sqsubseteq R$ is $\precsim$-*admissible *if $R_i \precsim R$ $(1 \le i \le n)$.*

Unlike $\prec$-regularity, however, $\precsim$-admissibility is not sufficient for regularity since every RIA is $\precsim$-admissible for the *total* preorder $\precsim$, i.e., the one such that $R_1 \precsim R_2$ holds for all roles R_1 and R_2. Note $l_{\precsim}(R) = 0$ for every role R w.r.t. to this preorder. To regain regularity, we impose an additional condition on the set of RIAs $\mathcal{R}$ as a whole.

[4] http://www.ihtsdo.org/

Definition 2. *Given a set of $\precsim$-admissible RIAs $\mathcal{R}$, we say that a RIA $\rho \sqsubseteq R'$ is $\precsim$-*stratified *in $\mathcal{R}$, if for every $R \approx R'$ such that $\rho = \rho_1 R \rho_2$, there exists R_1 such that $\rho_1 R \sqsubseteq^*_{\mathcal{R}} R_1$ and $R_1 \rho_2 \sqsubseteq^*_{\mathcal{R}} R'$. We say that $\mathcal{R}$ is $\precsim$-*stratified *if every RIA $\rho \sqsubseteq R$ provable in $\mathcal{R}$, is $\precsim$-stratified in $\mathcal{R}$.*

Intuitively, a set of RIAs $\mathcal{R}$ is $\precsim$-stratified, if every RIA $\rho \sqsubseteq R'$ provable in $\mathcal{R}$ is always provable in $\mathcal{R}$ when reducing the left-most roles of the maximal level first. For example, consider the set $\mathcal{R}$ consisting of RIAs (2)–(4) and (11) and the preorder $\precsim$ such that $\mathsf{isPartOf} \approx \mathsf{isProperPartOf} \precsim \mathsf{isInjuryOf}$ w.r.t. which $\mathcal{R}$ is clearly stratified. Then both of the following RIAs are provable in $\mathcal{R}$ and are $\precsim$-stratified in $\mathcal{R}$:

$$\mathsf{isPartOf} \cdot \mathsf{isPartOf} \cdot \mathsf{isPropertPartOf} \sqsubseteq \mathsf{isProperPartOf}, \tag{12}$$

$$\mathsf{isInjuryOf} \cdot \mathsf{isPartOf} \cdot \mathsf{isPropertPartOf} \sqsubseteq \mathsf{isInjuryOf}. \tag{13}$$

RIA (12) is stratified because for $\rho_1 := \mathsf{isPartOf}$, $R := \mathsf{isPartOf} \approx \mathsf{isProperPartOf} =: R'$, and $\rho_2 := \mathsf{isProperPartOf}$, we have $\rho_1 R = \mathsf{isPartOf}\cdot\mathsf{isPartOf} \sqsubseteq_{(3)} \mathsf{isPartOf} := R_1$ and $R_1 \rho_2 = \mathsf{isPartOf} \cdot \mathsf{isProperParOf} \sqsubseteq_{(4)} \mathsf{isProperPartOf} = R'$. Note that when either $\rho_1 = \epsilon$ or $\rho_2 = \epsilon$, the conditions of Definition 2 hold trivially. RIA (13) is stratified because $R \approx R' := \mathsf{isInjuryOf}$ holds only for $R = \mathsf{isInjuryOf}$, in which case $\rho_1 = \epsilon$. It can be similarly shown that every RIA provable in $\mathcal{R}$, is $\precsim$-stratified in $\mathcal{R}$, so $\mathcal{R}$ is $\precsim$-stratified. If we, however, extend the preorder $\precsim$ such that $\mathsf{isPartOf} \approx \mathsf{isProperPartOf} \approx \mathsf{isInjuryOf}$, i.e., take the total preorder $\precsim$, RIA (13) will be no longer $\precsim$-stratified. Indeed, for $\rho_1 := \mathsf{isInjuryOf}$, $R := \mathsf{isPartOf} \approx \mathsf{isInjuryOf} := R'$, and $\rho_2 := \mathsf{isProperPartOf}$, there does not exist R_1 such that $\rho_1 R = \mathsf{isInjuryOf} \cdot \mathsf{isPartOf} \sqsubseteq^*_{\mathcal{R}} R_1$.

As seen from this example, the choice of the preorder $\precsim$ has an impact on whether $\mathcal{R}$ is $\precsim$-stratified or not. As we pointed out, every RIA is $\precsim$-admissible for the total preorder $\precsim$. However, since all roles R have the same maximal level $L_{\precsim}(R) = 0$ for the total preorder $\precsim$, to check if $\rho \sqsubseteq R'$ is $\precsim$-stratified, one has to consider every role R in ρ, and prove that $\rho_1 R \sqsubseteq^*_{\mathcal{R}} R_1$ and $R_1 \rho_2 \sqsubseteq^*_{\mathcal{R}} R'$ hold for the respective prefix ρ_1 and suffix ρ_2. On the other hand, by taking the *smallest* preorder $\precsim_{\mathcal{R}}$ for which the RIAs in $\mathcal{R}$ are $\precsim_{\mathcal{R}}$-admissible, one can avoid many of these tests. The *smallest preorder $\precsim_R$ for $\mathcal{R}$* can be defined as the transitive reflexive closure of the relation $\prec_{\mathcal{R}}$ such that $R_1 \prec_{\mathcal{R}} R_2$ iff $\rho_1 R_1 \rho_2 \sqsubseteq R_2 \in \mathcal{R}$ for some ρ_1 and ρ_2. It can easily be shown using Definition 1 and Definition 2 that for every preorder $\precsim$, (i) all RIAs in $\mathcal{R}$ are $\precsim$-admissible iff $\precsim$ extends $\precsim_{\mathcal{R}}$, and (ii) if $\mathcal{R}$ is $\precsim$-stratified then $\mathcal{R}$ is $\precsim_{\mathcal{R}}$-stratified. From (ii) it follows, in particular, that $\mathcal{R}$ is $\precsim$-stratified for *some* order $\precsim$ iff $\mathcal{R}$ is $\precsim_{\mathcal{R}}$-stratified.

Our next goal is to prove that every $\precsim$-stratified set of RIAs $\mathcal{R}$ induces a regular language $L_{\mathcal{R}}(R)$ for every role R. From now on, we assume that we are given a fixed preorder $\precsim$ and a set of $\precsim$-admissible RIAs $\mathcal{R}$. So, when we say that $\mathcal{R}$ is stratified or a RIA is stratified, we mean $\mathcal{R}$ is $\precsim$-stratified and the RIA is $\precsim$-stratified in $\mathcal{R}$.

First, we distinguish two types of RIAs according to the levels of their roles:

Definition 3. *The* level *of a RIA $\alpha = (R_1 \cdots R_n \sqsubseteq R) \in \mathcal{R}$ (w.r.t. $\precsim$) is $l_{\precsim}(\alpha) := l_{\precsim}(R)$. We say that α is* simple *if $R_i \prec R$ for all i with $1 \leq i \leq (n-1)$; otherwise we say that α is* complex. *For $n \geq 0$, define $\mathcal{R}_n := \{\alpha \in \mathcal{R} \mid l_{\precsim}(\alpha) = n\}$, $\mathcal{R}_{<n} := \bigcup_{k<n} \mathcal{R}_k$, and define $\mathcal{R}^s_n$ to be the set of simple RIAs in $\mathcal{R}_n$.*

In the next lemma, we demonstrate that for every stratified set of RIAs w.l.o.g. one can assume a certain precedence on RIAs in proofs: RIAs of smaller level are applied first; simple RIAs are applied before complex RIAs of the same level; and complex RIAs are only applied to the prefix of the role chain, i.e, at the position 0.

Lemma 1. *For every ρ and R' such that $\rho \sqsubseteq^*_{\mathcal{R}} R'$, there exist ρ^1 and ρ^2 such that $\rho \sqsubseteq^*_{\mathcal{R}_{<n}} \rho^1 \sqsubseteq^*_{\mathcal{R}^s_n} \rho^2 \sqsubseteq^*_{\mathcal{R}_{n},0} R'$, where $n = l_{\precsim}(R')$.*

Proof. W.l.o.g., one can assume that $\mathcal{R}$ does not contain RIAs of the form $\epsilon \sqsubseteq R$. Indeed, otherwise for $\mathcal{R}^\epsilon := \{\epsilon \sqsubseteq R \in \mathcal{R}\}$ and $\mathcal{R}' := \mathcal{R} \setminus \mathcal{R}^\epsilon$, we have $\rho \sqsubseteq^*_{\mathcal{R}} R'$ iff $\rho \sqsubseteq^*_{\mathcal{R}^\epsilon} \rho_1 \sqsubseteq^*_{\mathcal{R}'} R'$ for some ρ_1. Now if the lemma holds for $\mathcal{R}'$, then there exist ρ^1_1 and ρ^2_1 such that $\rho_1 \sqsubseteq^*_{\mathcal{R}'_{<n}} \rho^1_1 \sqsubseteq^*_{\mathcal{R}'^s_n} \rho^2_1 \sqsubseteq^*_{\mathcal{R}'_{n,0}} R'$. In this case, it can be readily seen that $\rho \sqsubseteq^*_{\mathcal{R}^\epsilon_{<n}} \rho^1 \sqsubseteq^*_{\mathcal{R}'_{<n}} \rho^2 \sqsubseteq^*_{\mathcal{R}^\epsilon_n} \rho^3 \sqsubseteq^*_{\mathcal{R}'^s_n} \rho^2_1 \sqsubseteq^*_{\mathcal{R}'_{n,0}} R'$ for some ρ^1, ρ^2, and ρ^3.

Now, consider all ρ' such that $\rho \sqsubseteq^*_{\mathcal{R}} \rho' \sqsubseteq^*_{\mathcal{R}} R'$. Since $\mathcal{R}$ does not contain RIAs of the form $\epsilon \sqsubseteq R$, the number of all such ρ' is bounded. From all such ρ', select all $\rho' = \rho'_0 R_1 \rho'_1 \cdots R_m \rho'_m$ with the maximal number of occurrences $R_1, \ldots, R_m$ of roles of level n, and from them select one of the largest length. Then $\rho = \rho_0 \rho^1_1 \rho_1 \cdots \rho^1_m \rho_m$ such that $\rho_i \sqsubseteq^*_{\mathcal{R}_{<n}} \rho'_i$ $(0 \leq i \leq m)$ and $\rho^1_i \sqsubseteq^*_{\mathcal{R}_{<n}} \rho^2_i \sqsubseteq^*_{\mathcal{R}^s_n} R_i$ $(1 \leq i \leq m)$. Otherwise one could find ρ' with more occurrences of roles of level n or the same number of occurrences but of a larger length.

Since $\rho'_0 R_1 \rho'_1 \cdots R_m \rho'_m \sqsubseteq^*_{\mathcal{R}} R'$ and $\mathcal{R}$ is stratified, there exist R'_i $(1 \leq i \leq m)$ such that $\rho'_0 R_1 \sqsubseteq^*_{\mathcal{R}} R'_1$, $R'_i \rho'_i R_{i+1} \sqsubseteq^*_{\mathcal{R}} R'_{i+1}$ $(1 \leq i < m)$, and $R'_m \rho'_m \sqsubseteq^*_{\mathcal{R}} R'$. In particular, there exist ρ'^1_i $(1 \leq i \leq m)$, ρ'^2_i $(0 \leq i < m)$ that do not contain roles of level n, and R^1_i $(1 < i \leq m)$ such that $\rho'_0 \sqsubseteq^*_{\mathcal{R}_{<n}} \rho'^2_0$, $\rho'^2_0 R_1 \sqsubseteq^*_{\mathcal{R}^s_n} R'_1$, $\rho'_i \sqsubseteq^*_{\mathcal{R}_{<n}} \rho'^1_i \rho'^2_i$, $\rho'^2_i R_{i+1} \sqsubseteq^*_{\mathcal{R}^s_n} R^1_{i+1}$, $R'_i \rho'^1_i R^1_{i+1} \sqsubseteq^*_{\mathcal{R}_{n},0} R'_{i+1}$ $(1 \leq i < m)$, $\rho'_m \sqsubseteq^*_{\mathcal{R}_{<n}} \rho'^1_m$, and $R'_m \rho'^1_m \sqsubseteq^*_{\mathcal{R}_{n},0} R'$. Otherwise one could again find ρ' with more roles of level n or with the same number of roles but of a larger length.

Summing up, we obtain the required ρ^1 and ρ^2 as follows:

$$\begin{aligned}\rho =& \rho_0 \rho^1_1 \rho_1 \cdots \rho^1_m \rho_m \sqsubseteq^*_{\mathcal{R}_{<n}} \rho^1 := \rho'^2_0 \rho^2_1 \rho'^1_1 \rho'^2_1 \cdots \rho^2_m \rho'^1_m \\ & \sqsubseteq^*_{\mathcal{R}^s_n} \rho^2 := R'_1 \rho'^1_1 R^1_2 \rho'^1_2 \cdots R^1_m \rho'^1_m \sqsubseteq^*_{\mathcal{R}_{n},0} R'.\end{aligned}$$ □

We are now in a position to prove that every stratified set of RIAs $\mathcal{R}$ is regular.

Theorem 1. *For every $\precsim$-stratified set of RIAs $\mathcal{R}$ and every role R, one can construct a non-deterministic finite automaton (NFA) that recognizes the language $L_{\mathcal{R}}(R)$. The size (i.e., the number of transitions) of the automaton is bounded by $(c \cdot m)^{2 \cdot n}$ where $m := \|\mathcal{R}\|$, $n := l_{\precsim}(R)$, and c is some fixed constant.*

Proof. By Lemma 1, for every role R we have:

$$L_{\mathcal{R}}(R) = \{\rho \mid \exists \rho^1 \exists \rho^2 : \rho \sqsubseteq^*_{\mathcal{R}_{<n}} \rho^1 \sqsubseteq^*_{\mathcal{R}^s_n} \rho^2 \sqsubseteq^*_{\mathcal{R}_{n},0} R\}. \tag{14}$$

We first show that the languages $L_{\mathcal{R}^s_n}(R)$ and $L_{\mathcal{R}_{n},0}(R) := \{\rho \mid \rho \sqsubseteq^*_{\mathcal{R}_{n},0} R\}$ can be recognized by NFAs. For every role R, introduce a terminal symbol a_R and a non-terminal symbol A_R. For $\mathcal{R}^s_n$, consider a grammar containing production rules

$A_R \to a_R$ for every role R and $A_R \to a_{R_1} \cdots a_{R_{k-1}} A_{R_k}$ for every RIA $R_1 \cdots R_k \sqsubseteq R \in \mathcal{R}_n^s$. Since for every simple RIA $R_1 \cdots R_k \sqsubseteq R \in \mathcal{R}_n^s$ the level of the roles R_i with $1 \leq i < k$ is smaller than n, it is easy to show by induction that $R_1 \cdots R_m \in L_{\mathcal{R}_n^s}(R)$ iff $A_R \to^* a_{R_1} \ldots a_{R_m}$. Since the grammar is right-linear, the language $L_{\mathcal{R}_n^s}(R)$ is regular. Similarly, the language $L_{\mathcal{R}_n,0}(R)$ is regular since it corresponds to the left-linear grammar containing production rules $A_R \to a_R$ for every role R and $A_R \to A_{R_1} a_{R_2} \cdots a_{R_k}$ for every RIA $R_1 \cdots R_k \sqsubseteq R \in \mathcal{R}_n$. It is well-known that for left- and right-linear grammars one can construct an NFA with size linear in the size of the grammar, which in our cases is bounded by $c \cdot m$ where $m := \|\mathcal{R}\|$ and c a constant.

Assume, by induction hypothesis, that for every R such that $l_{\precsim}(R) < n$, $L_{\mathcal{R}}(R)$ can be recognized by an NFA of size at most $(c \cdot m)^{2\cdot(n-1)}$. Since $L_{\mathcal{R}_{<n}}(R) = \{R\}$ if $l_{\precsim}(R) \geq n$ and $L_{\mathcal{R}_{<n}}(R) = L_{\mathcal{R}}(R)$ if $l_{\precsim}(R) < n$, for $L_{\mathcal{R}_{<n}}(R)$ one can construct an NFA of size bounded by $(c \cdot m)^{2\cdot(n-1)}$. For the remaining case $l_{\precsim}(R) = n$, consider $L_n(R) := \{\rho^1 \mid \exists \rho^2 : \rho^1 \sqsubseteq^*_{\mathcal{R}_n^s} \rho^2 \sqsubseteq^*_{\mathcal{R}_n,0} R\}$. Clearly, $\rho^1 \in L_n(R)$ iff there exists $\rho^2 = R_1 \cdots R_k \in L_{\mathcal{R}_n,0}(R)$ such that $\rho^1 = \rho^1_1 \cdots \rho^1_k$ where $\rho^1_i \in L_{\mathcal{R}_n^s}(R_i)$ $(1 \leq i \leq k)$. Hence $L_n(R) = L_{\mathcal{R}_n,0}(R)[R_1/L_{\mathcal{R}_n^s}(R_1), \ldots, R_k/L_{\mathcal{R}_n^s}(R_k)]$ where R_i are all roles of level n and $L[R_1/L_1, \ldots, R_k/L_k]$ denotes the language obtained by substituting in every word from L the letters R_i with words from L_i in all possible ways. Since regular languages are closed under substitution and an NFA for $L[R_1/L_1, \ldots, R_m/L_m]$ can be constructed with the size bounded by the size of the NFA for L multiplied with the maximum size of the NFA for $L_1, \ldots, L_m$, the language $L_n(R)$ is regular and can be recognized by an NFA of size at most $(c \cdot m)^2$.

Similarly, by (14), we have $L_{\mathcal{R}}(R) = L_n(R)[R_1/L_{\mathcal{R}_{<n}}(R_1), \ldots, R_k/L_{\mathcal{R}_{<n}}(R_k)]$ where R_i for $1 \leq i \leq k$ are all roles of level n. Thus, one can construct an NFA of size bounded by $(c \cdot m)^2 \cdot (c \cdot m)^{2\cdot(n-1)} = (c \cdot m)^{2\cdot n}$ that recognizes $L_{\mathcal{R}}(R)$. □

5 Testing If a Set of RIAs Is Stratified

Up to now we have demonstrated that, similar to $\prec$-regularity, Definition 2 provides a sufficient condition for regularity of a set of RIAs $\mathcal{R}$. However, unlike $\prec$-regularity, Definition 2 does not provide for any effective means of testing this condition since it requires to test if all (of possibly infinitely many) RIAs $\rho \sqsubseteq R$ provable in $\mathcal{R}$, are stratified. Below we demonstrate that it suffices to test regularity only for finitely many RIAs that can be effectively computed from $\mathcal{R}$.

Definition 4. *Let $\mathcal{R}$ be a set of $\precsim$-admissible RIAs. RIA $\rho_1 R \rho_2 \sqsubseteq R'$ is an* overlap *of two RIAs $\rho_2^\epsilon R \rho_2 \sqsubseteq R_1$ and $\rho_1 R_2 \rho_1^\epsilon \sqsubseteq R'$ (w.r.t. $\mathcal{R}$ and $\precsim$) if $R \approx R'$, $\epsilon \sqsubseteq^*_{\mathcal{R}} \rho_2^\epsilon$, $\epsilon \sqsubseteq^*_{\mathcal{R}} \rho_1^\epsilon$, and $R_1 \sqsubseteq^*_{\mathcal{R}} R_2$. $\mathcal{R}$ is* weakly $\precsim$-stratified *if (i) every RIA in $\mathcal{R}$ is $\precsim$-stratified in $\mathcal{R}$ and (ii) the overlap of every two RIAs in $\mathcal{R}$ is $\precsim$-stratified in $\mathcal{R}$.*

Note that the overlap $\rho_1 R \rho_2 \sqsubseteq R'$ is provable in $\mathcal{R}$ in a such a way that only RIAs $\rho_2^\epsilon R \rho_2 \sqsubseteq R_1$ and $\rho_1 R_2 \rho_1^\epsilon \sqsubseteq R'$ involved in the overlap reduce the length of the role chains: $\rho_1(R\rho_2) \sqsubseteq^*_{\mathcal{R}} \rho_1(\rho_2^\epsilon R \rho_2) \sqsubseteq_{\mathcal{R}} \rho_1 R_1 \sqsubseteq^*_{\mathcal{R}} \rho_1 R_2 \sqsubseteq^*_{\mathcal{R}} \rho_1 R_2 \rho_1^\epsilon \sqsubseteq_{\mathcal{R}} R'$. Intuitively, to prove that $\mathcal{R}$ is weakly stratified, one has to consider only RIAs provable in $\mathcal{R}$ using at most two reducing steps, first, reducing the suffix of a role chain, and second, reducing the prefix of the role chain: $\rho_1(R\rho_2) \sqsubseteq^*_{\mathcal{R}} \rho_1 R_1 \sqsubseteq^*_{\mathcal{R}} R'$.

For example, the set $\mathcal{R}$ of RIAs (2)–(4) is weakly stratified. Indeed, every RIA in $\mathcal{R}$ is trivially stratified since no role chain in $\mathcal{R}$ has more than two roles. RIAs (3) and (4) can overlap (possibly with themselves) only in the following three cases:

$$\mathsf{isPartOf} \cdot (\mathsf{isPartOf} \cdot \mathsf{isPartOf}) \sqsubseteq_{(2)} \mathsf{isPartOf} \cdot \mathsf{isPartOf} \sqsubseteq_{(3)} \mathsf{isPartOf}, \tag{15}$$

$$\begin{aligned} \mathsf{isPartOf} \cdot (\mathsf{isPartOf} \cdot \mathsf{isProperPartOf}) &\sqsubseteq_{(4)} \\ \mathsf{isPartOf} \cdot \mathsf{isProperPartOf} &\sqsubseteq_{(4)} \mathsf{isProperPartOf}, \end{aligned} \tag{16}$$

$$\begin{aligned} \mathsf{isPartOf} \cdot (\mathsf{isPartOf} \cdot \mathsf{isProperPartOf}) &\sqsubseteq_{(4)} \\ \mathsf{isPartOf} \cdot \mathsf{isProperPartOf} &\sqsubseteq_{(2)} \\ \mathsf{isPartOf} \cdot \mathsf{isPartOf} &\sqsubseteq_{(3)} \mathsf{isPartOf}. \end{aligned} \tag{17}$$

The resulted overlaps are provable using (2)–(4) "left-to-right" and thus stratified:

$$(\mathsf{isPartOf} \cdot \mathsf{isPartOf}) \cdot \mathsf{isPartOf} \sqsubseteq_{(3),(3)} \mathsf{isPartOf}, \tag{18}$$

$$(\mathsf{isPartOf} \cdot \mathsf{isPartOf}) \cdot \mathsf{isProperPartOf} \sqsubseteq_{(3),(4)} \mathsf{isProperPartOf}, \tag{19}$$

$$(\mathsf{isPartOf} \cdot \mathsf{isPartOf}) \cdot \mathsf{isProperPartOf} \sqsubseteq_{(3),(4),(2)} \mathsf{isPartOf}. \tag{20}$$

Similarly, one can show that $\bar{\mathcal{R}}$ is weakly stratified by considering all overlaps between the inverses of (3) and (4).

The notion of overlap and conditions for a weakly stratified set of RIAs are reminiscent of the well-known notions of a *critical pair* and the *weak Church-Rosser property* from term rewriting [3]. Despite close resemblance, there seem, however, to be no direct correspondence between these properties—if to consider the entailment relation $\sqsubseteq_{\mathcal{R}}$ as a rewriting relation on chains of roles, the conditions of Definition 4 essentially mean that if a role chain can be rewritten to a role using at most two complex "rightmost" reductions, then it can also be rewritten to the same role chain using only "leftmost" reductions. Like in Church-Rosser theorem, however, it is possible to prove that every weakly stratified set of RIAs is stratified:

Theorem 2. *For every preorder $\precsim$ and every set of $\precsim$-admissible RIAs $\mathcal{R}$, $\mathcal{R}$ is $\precsim$-stratified iff $\mathcal{R}$ is weakly $\precsim$-stratified.*

Proof. As in the proof of Lemma 1, we first prove that w.l.o.g. one can assume that $\mathcal{R}$ does not contain RIAs of the form $\epsilon \sqsubseteq R$. Otherwise, we extend $\mathcal{R}$ by repeatedly adding for every $\epsilon \sqsubseteq R \in \mathcal{R}$ and $\rho_1 R \rho_2 \sqsubseteq R' \in \mathcal{R}$ a RIA $\rho_1\rho_2 \sqsubseteq R$. This transformation preserves the set of implied RIAs, and therefore the result $\mathcal{R}'$ is stratified iff $\mathcal{R}$ is stratified. It can also be shown that if $\rho \sqsubseteq R'$ is provable in $\mathcal{R}'$ and $\rho \neq \epsilon$ then $\rho \sqsubseteq R'$ is provable in $\mathcal{R}'$ without using RIAs of the form $\epsilon \sqsubseteq R$. Hence, if we prove that is $\mathcal{R}'$ weakly stratified iff $\mathcal{R}$ is weakly stratified, we can disregard all axioms of the form $\epsilon \sqsubseteq R$ in this proof (since all provable RIAs $\rho \sqsubseteq R$ for $\rho = \epsilon$ are trivially stratified).

It is clear that if $\mathcal{R}'$ is weakly stratified then $\mathcal{R}$ is as well since $\mathcal{R}'$ contains $\mathcal{R}$. To prove the converse, assume that $\mathcal{R}$ is weakly stratified. Let $\mathcal{R}^\epsilon := \{\epsilon \sqsubseteq R \in \mathcal{R}\}$. Note that if $\rho' \sqsubseteq_{R^\epsilon} \rho$ and $\rho \sqsubseteq R$ is stratified then $\rho' \sqsubseteq R$ is stratified as well (in both $\mathcal{R}$ and $\mathcal{R}'$). Hence the condition (i) of Definition 4 for $\mathcal{R}'$ is immediate from this property and the construction of $\mathcal{R}'$. To prove condition (ii) of Definition 4 for $\mathcal{R}'$, let

$\rho_1 R \rho_2 \sqsubseteq R'$ be an overlap of two RIAs $\rho_2^\epsilon R \rho_2 \sqsubseteq R_1$ and $\rho_1 R_2 \rho_1^\epsilon \sqsubseteq R'$ in $\mathcal{R}'$. From the construction of $\mathcal{R}'$, there should be RIAs $\rho_4^\epsilon R \rho_4 \sqsubseteq R_1$ and $\rho_3 R_2 \rho_3^\epsilon \sqsubseteq R'$ in $\mathcal{R}$ such that $\rho_2^\epsilon \sqsubseteq^*_{\mathcal{R}^\epsilon} \rho_4^\epsilon$, $\rho_2 \sqsubseteq^*_{\mathcal{R}^\epsilon} \rho_4$, $\rho_1 \sqsubseteq^*_{\mathcal{R}^\epsilon} \rho_3$, and $\rho_1^\epsilon \sqsubseteq^*_{\mathcal{R}^\epsilon} \rho_3^\epsilon$. In particular $\epsilon \sqsubseteq^*_{\mathcal{R}} \rho_2^\epsilon \sqsubseteq^*_{\mathcal{R}} \rho_4^\epsilon$, $\epsilon \sqsubseteq^*_{\mathcal{R}} \rho_1^\epsilon \sqsubseteq^*_{\mathcal{R}} \rho_3^\epsilon$, and so, $\rho_3 R \rho_4 \sqsubseteq R'$ is an overlap of RIAs in $\mathcal{R}$, which by condition (ii) should be stratified. Since $\rho_1 R \rho_2 \sqsubseteq^*_{\mathcal{R}^\epsilon} \rho_3 R \rho_4$, we obtain that $\rho_1 R \rho_2 \sqsubseteq R'$ is stratified as well.

So now, w.l.o.g., we can assume that $\mathcal{R}$ does not contain RIAs of the form $\epsilon \sqsubseteq R$.

The "only if" direction of the theorem is trivial since RIAs in $\mathcal{R}$ and overlaps of RIAs in $\mathcal{R}$ are provable in $\mathcal{R}$.

To prove the "if" direction, assume to the contrary that there exists ρ such that $\rho \sqsubseteq^*_{\mathcal{R}} R'$ but $\rho \sqsubseteq R'$ is not stratified w.r.t. $\mathcal{R}$ and $\precsim$.

Take such a ρ of the smallest length. Then $\rho = \rho_1 R \rho_2$ where $R \approx R'$ and there exists no R_1 such that $\rho_1 R \sqsubseteq^*_{\mathcal{R}} R_1$ and $R_1 \rho_2 \sqsubseteq^*_{\mathcal{R}} R'$. Clearly, $\rho_1 \neq \epsilon$ and $\rho_2 \neq \epsilon$.

Since $\rho = \rho_1 R \rho_2 \sqsubseteq^*_{\mathcal{R}} R'$, there exist ρ_1^1, ρ_1^2, ρ_2^1, ρ_2^2, R^1, and R^2 such that $\rho_1 \sqsubseteq^*_{\mathcal{R}} \rho_1^2 \rho_1^1$, $\rho_2 \sqsubseteq^*_{\mathcal{R}} \rho_2^1 \rho_2^2$, $R \sqsubseteq^*_{\mathcal{R}} R^1$, $\rho_1^1 R^1 \rho_2^1 \sqsubseteq R^2 \in \mathcal{R}$, $\rho_1^2 R^2 \rho_2^2 \sqsubseteq^*_{\mathcal{R}} R'$, and $\rho_1^1 \rho_2^1 \neq \epsilon$, so:

$$\rho_1 R \rho_2 \sqsubseteq^*_{\mathcal{R}} \rho_1^2 (\rho_1^1 R^1 \rho_2^1) \rho_2^2 \sqsubseteq_{\mathcal{R}} \rho_1^2 R^2 \rho_2^2 \sqsubseteq^*_{\mathcal{R}} R'.$$

Since $\mathcal{R}$ is weakly stratified and $\rho_1^1 R^1 \rho_2^1 \sqsubseteq R^2 \in \mathcal{R}$, by condition (i) in Definition 4, $\rho_1^1 R^1 \sqsubseteq^*_{\mathcal{R}} R_1^1$ and $R_1^1 \rho_2^1 \sqsubseteq^*_{\mathcal{R}} R^2$ for some R_1^1. In particular, $\rho_1^2 R_1^1 \rho_2^1 \rho_2^2 \sqsubseteq^*_{\mathcal{R}} R'$.

We prove that $\rho_1^1 = \epsilon$. If $\rho_1^1 \neq \epsilon$, then $\|\rho_1^2 R_1^1 \rho_2^1 \rho_2^2\| < \|\rho\|$, so, $\rho_1^2 R_1^1 \sqsubseteq^*_{\mathcal{R}} R_1$ and $R_1 \rho_2^1 \rho_2^2 \sqsubseteq^*_{\mathcal{R}} R'$ for some R_1. We obtain a contradiction since $\rho_1 R \sqsubseteq^*_{\mathcal{R}} \rho_1^2 (\rho_1^1 R^1) \sqsubseteq^*_{\mathcal{R}} \rho_1^2 R_1^1 \sqsubseteq^*_{\mathcal{R}} R_1$ and $R_1 \rho_2 \sqsubseteq^*_{\mathcal{R}} R_1 \rho_2^1 \rho_2^2 \sqsubseteq^*_{\mathcal{R}} R'$, but we assumed that no such R_1 exists. Thus $\rho_1^1 = \epsilon$.

Now we prove that $\rho_2^2 = \epsilon$. Since $\rho_1^1 \rho_2^1 \neq \epsilon$, $\|\rho_1^2 R^2 \rho_2^2\| < \|\rho\|$, so, $\rho_1^2 R^2 \sqsubseteq^*_{\mathcal{R}} R_1^2$ and $R_1^2 \rho_2^2 \sqsubseteq^*_{\mathcal{R}} R'$ for some R_1^2. In particular, $\rho_1^2 \rho_1^1 R^1 \rho_2^1 \sqsubseteq^*_{\mathcal{R}} R_1^2$. If $\rho_2^2 \neq \epsilon$ then $\|\rho_1^2 \rho_1^1 R^1 \rho_2^1\| < \|\rho\|$, so $\rho_1^2 \rho_1^1 R^1 \sqsubseteq^*_{\mathcal{R}} R_1$ and $R_1 \rho_2^1 \sqsubseteq^*_{\mathcal{R}} R_1^2$ for some R_1. We obtain a contradiction since $\rho_1 R \sqsubseteq^*_{\mathcal{R}} \rho_1^2 \rho_1^1 R^1 \sqsubseteq^*_{\mathcal{R}} R_1$ and $R_1 \rho_2 \sqsubseteq^*_{\mathcal{R}} (R_1 \rho_2^1) \rho_2^2 \sqsubseteq^*_{\mathcal{R}} R_1^2 \rho_2^2 \sqsubseteq^*_{\mathcal{R}} R'$, but we assumed that no such R_1 exists. Thus $\rho_2^2 = \epsilon$.

Now since $\rho_1^2 R^2 \sqsubseteq^*_{\mathcal{R}} R'$, there exist ρ_1^3, ρ_1^4, R^3, and R^4 such that $\rho_1^2 \sqsubseteq^*_{\mathcal{R}} \rho_1^4 \rho_1^3$, $R^2 \sqsubseteq^*_{\mathcal{R}} R^3$, $\rho_1^3 R^3 \sqsubseteq R^4 \in \mathcal{R}$, $\rho_1^4 R^4 \sqsubseteq^*_{\mathcal{R}} R'$, and $\rho_1^3 \neq \epsilon$. So the full picture is:

$$\begin{aligned} \rho_1 R \rho_2 \sqsubseteq^*_{\mathcal{R}} \rho_1^2 R^1 \rho_2^1 \sqsubseteq^*_{\mathcal{R}} \rho_1^4 \rho_1^3 (R^1 \rho_2^1) \sqsubseteq_{\mathcal{R}} & \\ \rho_1^4 \rho_1^3 R^2 \sqsubseteq^*_{\mathcal{R}} \rho_1^4 (\rho_1^3 R^3) \sqsubseteq_{\mathcal{R}} \rho_1^4 R^4 \sqsubseteq^*_{\mathcal{R}} R'. \end{aligned}$$

In this case, $\rho_1^3 R^1 \rho_2^1 \sqsubseteq R^4$ is an overlap of the RIAs $R^1 \rho_2^1 \sqsubseteq R^2$ and $\rho_1^3 R^3 \sqsubseteq R^4$ w.r.t. $\mathcal{R}$ and $\precsim$. Since $\mathcal{R}$ is weakly stratified, by condition (ii) in Definition 4, $\rho_1^3 R^1 \sqsubseteq^*_{\mathcal{R}} R_1^3$ and $R_1^3 \rho_2^1 \sqsubseteq R^4$ for some R_1^3. In particular, $\rho_1^4 R_1^3 \rho_2^1 \sqsubseteq^*_{\mathcal{R}} R'$. Since $p_1^3 \neq \epsilon$, $\|\rho_1^4 R_1^1 \rho_2^1\| < \|\rho\|$, so, $\rho_1^4 R_1^3 \sqsubseteq^*_{\mathcal{R}} R_1$ and $R_1 \rho_2^1 \sqsubseteq^*_{\mathcal{R}} R'$ for some R_1. We obtain a contradiction since $\rho_1 R \sqsubseteq^*_{\mathcal{R}} \rho_1^4 (\rho_1^3 R^1) \sqsubseteq^*_{\mathcal{R}} \rho_1^4 R_1^3 \sqsubseteq^*_{\mathcal{R}} R_1$ and $R_1 \rho_2 \sqsubseteq^*_{\mathcal{R}} R_1 \rho_2^1 \sqsubseteq^*_{\mathcal{R}} R'$, but we assumed that no such R_1 exists. □

Now, in order to present an algorithm for deciding whether a set of RIAs $\mathcal{R}$ is stratified, according to Theorem 2, it is sufficient to prove that one can effectively check the conditions of Definition 4.

Lemma 2. *Given a set of RIAs $\mathcal{R}$ and a RIA $\rho \sqsubseteq R$, it is possible to decide in polynomial time whether $\rho \sqsubseteq^*_{\mathcal{R}} R$.*

Proof. Define a context-free grammar with terminal symbols a_R and non-terminal symbols A_R for every role R, and production rules $A_R \rightarrow a_R$ for every role R and $A_R \rightarrow A_{R_1} \ldots A_{R_n}$ for every RIA $R_1 \cdots R_n \sqsubseteq R \in \mathcal{R}$. It is easy to show that $A_R \rightarrow^* a_{R_1} \ldots a_{R_n}$ w.r.t. this grammar iff $R_1 \cdots R_n \sqsubseteq^*_{\mathcal{R}} R$. Since the word problem (membership in the language) for context-free grammars is decidable in polynomial time (see, e.g. [10]), so is the property $\rho \sqsubseteq^*_{\mathcal{R}} R$. □

Corollary 1. *For every $\precsim$-admissible set of RIAs $\mathcal{R}$, one can check in polynomial time in $\|\mathcal{R}\|$ if $\mathcal{R}$ is $\precsim$-stratified.*

Proof. By Theorem 2, to check if $\mathcal{R}$ is stratified, it is sufficient to check if every RIA in $\mathcal{R}$ is stratified and every overlap of two RIAs is stratified. Hence there are only polynomially-many RIAs to test. In order to test whether $\rho_1 R \rho_2 \sqsubseteq R'$ is stratified for R, we enumerate all roles R_1 in $\mathcal{R}$ and check if $\rho_1 R \sqsubseteq^*_{\mathcal{R}} R_1$ and $R_1 \rho_2 \sqsubseteq^*_{\mathcal{R}} R'$ hold. By Lemma 2, each of these conditions can be checked in polynomial time. □

Using the criterion in Theorem 2, it is now possible to show that for every set $\mathcal{R}$ of $\prec$-regular RIAs (according to the original conditions in $\mathcal{SROIQ}$), $\bar{\mathcal{R}}$ is stratified w.r.t. $\precsim$ defined by $R_1 \precsim R_2$ if either $R_1 \prec R_2$, $R_1 = R_2$ or $R_1 = \mathsf{Inv}(R_2)$. Clearly, the conditions of $\prec$-regularity ensure that every $\rho \sqsubseteq R \in \bar{\mathcal{R}}$ is stratified. Note also that if $\rho R \sqsubseteq R' \in \bar{\mathcal{R}}$ or $R\rho \sqsubseteq R' \in \bar{\mathcal{R}}$ with $R \eqsim R'$ then either $R' = R$ or $\rho = \epsilon$. Now, if $\rho_1 R \rho_2 \sqsubseteq R'$ is an overlap of two RIAs $\rho_2^\epsilon R \rho_2 \sqsubseteq R_2$ and $\rho_1 R_1 \rho_1^\epsilon \sqsubseteq R'$ in $\bar{\mathcal{R}}$ with $\rho_1 \neq \epsilon$ and $\rho_2 \neq \epsilon$ (otherwise it is trivially stratified), then $R_2 \sqsubseteq^*_{\bar{\mathcal{R}}} R_1$, $R \eqsim R_2 \eqsim R_1 \eqsim R'$, so $\rho_1^\epsilon = \rho_2^\epsilon = \epsilon$, $R_2 = R$, $R' = R_1$, and either $R_1 = R_2$ or $R_1 = \mathsf{Inv}(R_2)$ (by definition of $\precsim$). From the last and the fact that $R_2 \sqsubseteq^*_{\bar{\mathcal{R}}} R_1$, it follows that $R_1 \sqsubseteq^*_{\bar{\mathcal{R}}} R_2$. Hence, $\rho_1 R = \rho_1 R_2 \sqsubseteq^*_{\bar{\mathcal{R}}} \rho_1 R_1 \sqsubseteq_{\bar{\mathcal{R}}} R' = R_1 \sqsubseteq^*_{\bar{\mathcal{R}}} R_2 = R$, $R\rho_2 \sqsubseteq_{\bar{\mathcal{R}}} R_2 \sqsubseteq^*_{\bar{\mathcal{R}}} R_1 = R'$, and so $\rho_1 R \rho_2 \sqsubseteq R'$ is stratified.

To illustrate the practical benefits of Definition 4, consider the set $\mathcal{R}$ of RIAs (2)–(4) and (11) for the total preorder $\precsim$. As we showed before, $\bar{\mathcal{R}}$ is not stratified because of RIA (13) provable in $\bar{\mathcal{R}}$. RIA (13) can be obtained as an overlap of RIAs (4) and (11):

$$\begin{aligned} &\mathsf{isInjuryOf} \cdot (\mathsf{isPartOf} \cdot \mathsf{isProperPartOf}) \sqsubseteq_{(4)} \\ &\qquad\qquad \mathsf{isInjuryOf} \cdot \mathsf{isProperPartOf} \sqsubseteq_{(11)} \mathsf{isInjuryOf}. \end{aligned} \tag{21}$$

RIA (13) is not stratified, because there is no R_1 such that $\mathsf{isInjuryOf} \cdot \mathsf{isPartOf} \sqsubseteq^*_{\bar{\mathcal{R}}} R_1$ and $R_1 \cdot \mathsf{isProperPartPf} \sqsubseteq^*_{\bar{\mathcal{R}}} \mathsf{isInjuryOf}$ hold. In our situation, the roles isInjuryOf and isPartOf can be "composed" together with the third role isProperPartOf, but cannot be composed directly. This typically indicates on some missing properties, which the domain expert, presented with this situation, could often easily identify. In our case, the set of RIAs becomes stratified as soon as we add the axiom propagating the injury relation over the part-of relation, which then subsumes (11) given (2):

$$\mathsf{isInjuryOf} \cdot \mathsf{isPartOf} \sqsubseteq \mathsf{isInjuryOf}. \tag{22}$$

Thus, Definition 4 has two practical benefits. First, it can be used to check automatically if the given set of RIAs is stratified. Second, in the case when the set of RIAs is not stratified, it is possible to use this definition in interactive setting when the user is presented with problematic overlaps and prompted to enter the missing RIAs.

It is a natural question, whether for any set of RIAs $\mathcal{R}$ that induces regular languages, there exists an extension $\mathcal{R}'$, as in this example, that is stratified. The following theorem gives a surprising positive answer to this question. It turns out, there always exists a stratified *conservative extension* of $\mathcal{R}$—a super-set $\mathcal{R}'$ of $\mathcal{R}$ possibly containing new roles, such that any model $\mathcal{I}$ of $\mathcal{R}$ can be extended to a model $\mathcal{J}$ of $\mathcal{R}'$ that interprets the roles occurring in $\mathcal{R}$ exactly as $\mathcal{I}$ does.

Theorem 3. *Let $\mathcal{R}$ be a set of RIAs such that $L_{\bar{\mathcal{R}}}(R)$ is regular for every role R. Then there exists a conservative extension $\mathcal{R}'$ of $\mathcal{R}$ such that $\bar{\mathcal{R}}'$ is $\precsim$-stratified for every $\precsim$.*

Proof. Let Σ be the set of all roles occurring in $\bar{\mathcal{R}}$. For every $R \in \Sigma$ and $\rho_1, \rho_2 \in \Sigma^*$, define the language $L_{\bar{\mathcal{R}}}(R, \rho_1, \rho_2) := \{\rho \mid \rho_1\rho\rho_2 \in L_{\bar{\mathcal{R}}}(R)\}$. It follows from Myhill-Nerode theorem (see, e.g., [18]) that $L_{\bar{\mathcal{R}}}(R)$ is regular iff there are only finitely many different languages $L_{\bar{\mathcal{R}}}(R, \rho_1, \rho_2)$ for all possible ρ_1 and ρ_2. Let $L_{\bar{\mathcal{R}}} := \{L_{\bar{\mathcal{R}}}(R, \rho_1, \rho_2) \mid R \in \Sigma, \rho_1, \rho_2 \in \Sigma^*\}$ be the set of all languages of this form, and $S_{\mathcal{R}} := \{\bigcap_{L \in S} L \mid S \subseteq L_{\bar{\mathcal{R}}}\}$ be the set of all their possible intersections (the empty intersection is Σ^*). Since every language $L_{\bar{\mathcal{R}}}(R)$ is regular, clearly, both sets $L_{\bar{\mathcal{R}}}$ and $S_{\mathcal{R}}$ are finite. Note that $L_{\bar{\mathcal{R}}}(R) = L_{\bar{\mathcal{R}}}(R, \epsilon, \epsilon) \in L_{\bar{\mathcal{R}}} \subseteq S_{\mathcal{R}}$. For languages $L_1, \dots, L_n \subseteq \Sigma^*$ (not necessarily in $S_{\mathcal{R}}$), let $L_1 \cdots L_n := \{\rho_1 \cdots \rho_n \mid \rho_i \in L_i, 1 \le i \le n\}$ if $n > 0$, or $\{\epsilon\}$ if $n = 0$. One nice feature of $S_{\mathcal{R}}$ is the following interpolation-like property:

Claim 1. *For every L_1, L_2, L such that $L_1 \cdot L_2 \subseteq L \in S_{\mathcal{R}}$, there exists $L_1' \in S_{\mathcal{R}}$ and $L_2' \in S_{\mathcal{R}}$ such that $L_i \subseteq L_i'$, $i = 1, 2$, and $L_1' \cdot L_2' \subseteq L$.*

Indeed, let $L = \bigcap_{1 \le i \le n} L_{\bar{\mathcal{R}}}(R_i, \rho_i^1, \rho_i^2)$ for some R_i, ρ_i^1, and ρ_i^2, $(1 \le i \le n)$. Define:

$$L_1' := \bigcap_{\rho_2 \in L_2, 1 \le i \le n} L_{\bar{\mathcal{R}}}(R_i, \rho_i^1, \rho_2\rho_i^2); \qquad L_2' := \bigcap_{\rho_1 \in L_1', 1 \le i \le n} L_{\bar{\mathcal{R}}}(R_i, \rho_i^1\rho_1, \rho_i^2).$$

To prove that $L_1 \subseteq L_1'$, take any $\rho_1 \in L_1$. Since $L_1 \cdot L_2 \subseteq L$, for every $\rho_2 \in L_2$, we have $\rho_1\rho_2 \in L$. By definition of L and $L_{\bar{\mathcal{R}}}(R_i, \rho_i^1, \rho_i^2)$, we have $\rho_i^1\rho_1\rho_2\rho_i^2 \in L_{\bar{\mathcal{R}}}(R_i)$, $(1 \le i \le n)$, so $\rho_1 \in L_{\bar{\mathcal{R}}}(R_i, \rho_i^1, \rho_2\rho_i^2)$, $(1 \le i \le n)$. Thus $\rho_1 \in L_1'$.

We now prove that $L_1' \cdot L_2 \subseteq L$. Take any any $\rho_1 \in L_1'$ and $\rho_2 \in L_2$. By definition of L_1', $\rho_1 \in L_{\bar{\mathcal{R}}}(R_i, \rho_i^1, \rho_2\rho_i^2)$, so $\rho_1\rho_2 \in L_{\bar{\mathcal{R}}}(R_i, \rho_i^1, \rho_i^2)$, $(1 \le i \le n)$. Thus $\rho_1\rho_2 \in L$.

Using $L_1' \cdot L_2 \subseteq L$, it is now easy to show that $L_2 \subseteq L_2'$ and $L_1' \cdot L_2' \subseteq L$ symmetrically to the proofs above.

To continue the proof of the theorem, for every $L \in S_{\mathcal{R}}$, we introduce a fresh role R_L. Consider the set $\mathcal{R}_1$ consisting of the following RIAs for every $L, L_1, L_2 \in S_{\mathcal{R}}$:

$$\epsilon \sqsubseteq R_L \qquad \text{if } \epsilon \in L, \tag{23}$$

$$R_{L_1} \sqsubseteq R_L \qquad \text{if } L_1 \subseteq L, \tag{24}$$

$$R_{L_1} \cdot R_{L_2} \sqsubseteq R_L \qquad \text{if } L_1 \cdot L_2 \subseteq L. \tag{25}$$

Claim 2. *For every $L_1, \dots, L_n, L \in S_{\mathcal{R}}$, $L_1 \cdots L_n \subseteq L$ iff $R_{L_1} \cdots R_{L_n} \sqsubseteq^*_{\mathcal{R}_1} R_L$.*

The "if" direction of the claim can easily be shown by induction on the length of the proof of $R_{L_1} \cdots R_{L_n} \sqsubseteq^*_{\mathcal{R}_1} R_L$.

We prove the "only if" direction of the claim by induction on n. For $n \le 2$ the claim follows directly from (23)–(25).

Now, if $L_1 \cdots L_n \cdot L_{n+1} \subseteq L$ then by Claim 1, there exists $L' \in S_{\mathcal{R}}$ such that $L_1 \cdots L_n \subseteq L'$ and $L' \cdot L_{n+1} \subseteq L$. By induction hypothesis, $R_{L_1} \cdots R_{L_n} \sqsubseteq^*_{\mathcal{R}_1} R_{L'}$. Since by (25) we have $R_{L'} \cdot R_{L_{n+1}} \sqsubseteq R_L \in \mathcal{R}_1$, we obtain that $R_{L_1} \cdots R_{L_{n+1}} \sqsubseteq^*_{\mathcal{R}_1} R_L$, which was required to show.

We are now in a position to define the conservative extension $\mathcal{R}'$ of $\mathcal{R}$ required by the theorem. Let $\mathcal{R}'$ be an extension of $\mathcal{R}$ with $\mathcal{R}_1$ and the following axioms for every role $R \in \Sigma$:

$$R \sqsubseteq R_L, \text{ and } R_L \sqsubseteq R, \text{ where } L = L_{\bar{\mathcal{R}}}(R) \in S_{\mathcal{R}}. \tag{26}$$

Clearly, $\mathcal{R}'$ is a conservative extension of R. Indeed, every model $\mathcal{I}$ of $\mathcal{R}$ can be extended to a model of $\mathcal{R}'$ by interpreting the fresh roles R_L for $L \in S_{\mathcal{R}}$ as $R_L^{\mathcal{I}} := \bigcup_{\rho \in L} \rho^{\mathcal{I}}$. It is readily checked that $\mathcal{I}$ satisfies all RIAs (23)–(26).

Before proving that $\bar{\mathcal{R}}'$ is stratified, first note that for every $R_1 \cdots R_n \sqsubseteq R \in \mathcal{R}$ we have $R_{L_1} \cdots R_{L_n} \sqsubseteq^*_{\mathcal{R}_1} R_L$ when $L_i = L_{\bar{\mathcal{R}}}(R_i)$, $(1 \leq i \leq n)$, and $L = L_{\bar{\mathcal{R}}}(R)$. This follows directly from Claim 2 since $L_1 \cdots L_n \subseteq L$. Thus, every RIA in $\mathcal{R}$ is provable using axioms (23)–(26), and so, all RIAs in $\mathcal{R}$ can be disregarded. Moreover, we can regard every role R_L for $L = L_{\bar{\mathcal{R}}}(R)$ as a syntactic variant for the role R because of the axioms (26). Thus, it is sufficient to show that $\bar{\mathcal{R}}_1$ is stratified w.r.t. every $\precsim$ admissible for $\bar{\mathcal{R}}_1$.

Since the RIAs in $\mathcal{R}_1$ do not contain inverses, it is easy to see that $\rho \sqsubseteq^*_{\bar{\mathcal{R}}_1} R$ iff either $\rho \sqsubseteq^*_{\mathcal{R}_1} R$ or $\mathsf{Inv}(\rho) \sqsubseteq^*_{\mathcal{R}_1} \mathsf{Inv}(R)$. So, it is sufficient to show that for every $L_1, \ldots, L_n, L \in S_{\mathcal{R}}$ such that $R_{L_1} \cdots R_{L_n} \sqsubseteq^*_{\mathcal{R}_1} R_L$ and every $k = 1 \ldots n$, there exist R_k^1 and R_k^2 such that:

$$R_{L_1} \cdots R_{L_k} \sqsubseteq^*_{\mathcal{R}_1} R_k^1, \qquad R_k^1 R_{L_{k+1}} \cdots R_{L_n} \sqsubseteq^*_{\mathcal{R}_1} R_L, \tag{27}$$

$$R_{L_k} \cdots R_{L_n} \sqsubseteq^*_{\mathcal{R}_1} R_k^2, \qquad R_{L_1} \cdots R_{L_{k-1}} R_k^2 \sqsubseteq^*_{\mathcal{R}_1} R_L. \tag{28}$$

Indeed, by Claim 2, $L_1 \cdots L_n \subseteq L$. By Claim 1, there exist $L_k^1, L_k^2 \in S_{\mathcal{R}}$ such that $L_1 \cdots L_k \subseteq L_k^1$, $L_k \cdots L_n \subseteq L_k^2$, $L_k^1 L_{k+1} \cdots L_n \subseteq L$, and $L_1 \cdots L_{k-1} L_k^2 \subseteq L$. By Claim 2 we obtain (27) and (28) for $R_k^1 := R_{L_k^1}$ and $R_k^2 := R_{L_k^2}$. □

6 Related Work and Outlook

Complex RIAs are closely related to *inclusion (interaction) axioms* in *grammar modal logics* $\Box_{i_1} \cdots \Box_{i_n} X \to \Box_{j_1} \cdots \Box_{j_n} X$ [6,5,8]. Such axioms often cause undecidability, however Baldoni [5] and Demri [8] found a decidable class called the *regular grammar modal logics*. Demri and de Nivelle [9] gave a decision procedure for this class by a translation into the two-variable guarded fragment. The decision procedure assumes that a regular automata are given as an input of the procedure. When applying these results to ontologies and complex RIAs, such a restriction poses a serious practical problem because the users are unlikely to provide such automata. One proposed solution to this problem, is to use a sufficient syntactic condition for regularity, such as $\prec$-regularity [12,11]. Another sufficient condition [20] requires *associativity* of RIAs: if $R_1 R_2 \sqsubseteq R_1'$ and $R_1' R_3 \sqsubseteq R'$ then there should be R_2' such that $R_2 R_3 \sqsubseteq R_2'$ and $R_1 R_2' \sqsubseteq R'$. A similar condition was required for completeness of the *ordered chaining calculus* for first-order logic with compositional theories [4]. It is easy to see that associativity is a partial case of our sufficient conditions, when $\precsim$ is a total relation on roles. Therefore, our syntactic condition can be seen as a generalization of both associativity and $\prec$-regularity. Note that Theorem 3 is, in fact, proved for total preorders, and therefore it holds for all preorders.

Theorem 1 can possibly be relevant to several results in language theory identifying regular fragments of context-free languages and semi-Thue systems such as *non-self-embedded languages*, *one-letter grammars*, and *finite languages* (see, e.g., [7,1,15]). However, neither the original regularity condition for $\mathcal{SROIQ}$ [11], nor our extended condition, nor the associativity condition seem to relate to the known cases of regular context-free grammars. The reason could probably be that the conditions that are natural for compositional properties of binary relations ($R_1R_2 \sqsubseteq R_3$) might be not be so natural in the context of formal language theory ($A_1 \to A_2A_3$) and vice versa.

Theorem 3 means that stratified sets of RIAs can express any regular compositional properties of roles. In other words, our syntactic restriction has already maximal expressive power w.r.t. such properties and no further extension is necessary. Note that the proof of Theorem 3 is not constructive: it does not provide an algorithm for building the extension $\mathcal{R}'$ *automatically* from $\mathcal{R}$—it is necessary to know the regular automata for $L_{\bar{\mathcal{R}}}(R)$. It is an interesting question whether there exists such a completion procedure that terminates if all $L_{\bar{\mathcal{R}}}(R)$ are regular. It seems to be not even clear if it is possible to effectively check regularity for $\mathcal{R}$. It was claimed [9] that this problem is undecidable since it is undecidable whether a linear grammar is regular. But the problem of regularity for context-free grammars seems to be harder since context-free grammars distinguish between terminal and non-terminal symbols. There is no such a distinction between types of roles in RIAs, which makes a reduction from context-free grammars to sets of RIAs problematic. In this respect, the sets of RIAs are more related to so-called *sentential forms of context-free grammars* [16] or *pure context-free grammars* [14] where the symbols are not distinguished. A sentential form of a context-free grammar is a pure grammar that generates the language consisting of terminal and non-terminal symbols. The resulted language can be non-regular even for a regular grammar. For example, the linear grammar $A \to a$, $A \to aAa$ generates a regular language $L(A) = a(aa)^*$, but its sentential form generates a non-regular language $L^s(A) = L(A) \cup \{a^iAa^i \mid i \geq 0\}$. Pure grammars have different algorithmic properties than context-free grammars. For example, unlike for context-free grammars, given a pure context-free grammar and a regular automaton, it is decidable if they generate the same language [14]. The problem of regularity for pure grammars, however, is still open, to the best of our knowledge.

In this work we introduced a notion of stratified set of role inclusion axioms which provides a syntactically-checkable sufficient condition for regularity of RIAs—a condition that ensures decidability of $\mathcal{SROIQ}$ [11]. We demonstrated that for every stratified $\mathcal{SROIQ}$ ontology, one can construct a regular automaton representing the RIAs, which is worst case exponential in the size of the ontology. This implies that the complexity of reasoning with extended $\mathcal{SROIQ}$ remains the same as the complexity of the original $\mathcal{SROIQ}$, namely N2ExpTime-complete [13]. Moreover, we demonstrated that our conditions for regularity are in a sense maximal—every ontology $\mathcal{O}$ with regular RIAs can be conservatively extended to an ontology with stratified RIAs.

Acknowledgement

The author would like to thank the anonymous referees for valuable comments and Frantisek Simancik for pointing out to the literature on sentential forms and pure grammars. This work was supported by the EPSRC, grant number EP/G02085X.

References

1. Andrei, S., Chin, W.N., Cavadini, S.V.: Self-embedded context-free grammars with regular counterparts. Acta Inf. 40(5), 349–365 (2004)
2. Baader, F., Brandt, S., Lutz, C.: Pushing the $\mathcal{EL}$ envelope. In: Proc. IJCAI 05, pp. 364–369 (2005)
3. Baader, F., Nipkow, T.: Term rewriting and all that. Cambridge University Press, New York (1998)
4. Bachmair, L., Ganzinger, H.: Ordered chaining calculi for first-order theories of transitive relations. J. ACM 45(6), 1007–1049 (1998)
5. Baldoni, M.: Normal Multimodal Logics: Automatic Deduction and Logic Programming Extension. Ph.D. thesis, Università degli Studi di Torino (1998)
6. Fariñas del Cerro, L., Penttonen, M.: Grammar logics. Logique et Analyse 121-122, 123–134 (1988)
7. Chomsky, N.: On certain formal properties of grammars. Information and Control 2(2), 137–167 (1959)
8. Demri, S.: The complexity of regularity in grammar logics and related modal logics. J. Log. Comput. 11(6), 933–960 (2001)
9. Demri, S., de Nivelle, H.: Deciding regular grammar logics with converse through first-order logic. Journal of Logic, Language and Information 14(3), 289–329 (2005)
10. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley, Reading (1979)
11. Horrocks, I., Kutz, O., Sattler, U.: The even more irresistible $\mathcal{SROIQ}$. In: KR. pp. 57–67 (2006)
12. Horrocks, I., Sattler, U.: Decidability of $\mathcal{SHIQ}$ with complex role inclusion axioms. Artif. Intell. 160(1-2), 79–104 (2004)
13. Kazakov, Y.: $\mathcal{RIQ}$ and $\mathcal{SROIQ}$ are harder than $\mathcal{SHOIQ}$. In: KR, pp. 274–284. AAAI Press, Menlo Park (2008)
14. Maurer, H., Salomaa, A., Wood, D.: Pure grammars. Information and Control 44(1), 47–72 (1980)
15. Nederhof, M.J.: Practical experiments with regular approximation of context-free languages. Computational Linguistics 26(1), 17–44 (2000)
16. Salomaa, A.: On sentential forms of context-free grammars. Acta Informatica 2(1), 40–49 (1973)
17. Schulz, S., Romacker, M., Hahn, U.: Part-whole reasoning in medical ontologies revisited: Introducing SEP triplets into classification-based description logics. In: Proc. of the 1998 AMIA Annual Fall Symposium, pp. 830–834. Hanley & Belfus (1998)
18. Sipser, M.: Introduction to the Theory of Computation, 2nd edn. Course Technology (February 2005)
19. Suntisrivaraporn, B., Baader, F., Schulz, S., Spackman, K.A.: Replacing SEP-triplets in SNOMED CT using tractable description logic operators. In: Bellazzi, R., Abu-Hanna, A., Hunter, J. (eds.) AIME 2007. LNCS (LNAI), vol. 4594, pp. 287–291. Springer, Heidelberg (2007)
20. Wessel, M.: Obstacles on the way to qualitative spatial reasoning with description logics: Some undecidability results. In: Description Logics. CEUR Workshop Proceedings., vol. 49. CEUR-WS.org (2001)

Decreasing Diagrams and Relative Termination*

Nao Hirokawa[1] and Aart Middeldorp[2]

[1] School of Information Science
Japan Advanced Institute of Science and Technology, Japan
[2] Institute of Computer Science
University of Innsbruck, Austria

Abstract. In this paper we use the decreasing diagrams technique to show that a left-linear term rewrite system $\mathcal{R}$ is confluent if all its critical pairs are joinable and the critical pair steps are relatively terminating with respect to $\mathcal{R}$. We further show how to encode the rule-labeling heuristic for decreasing diagrams as a satisfiability problem. Experimental data for both methods are presented.

1 Introduction

This paper is concerned with automatically proving confluence of term rewrite systems. Unlike termination, for which the interest in automation gave and continues to give rise to new methods and tools,[1] automating confluence has received little attention. Only very recently, the first confluence tool made its appearance: ACP [2] implements Knuth and Bendix' condition—joinability of critical pairs—for terminating rewrite systems [14], several critical pair criteria for left-linear rewrite systems [11,19,21], as well as divide and conquer techniques based on persistence [1], layer-preservation [16], and commutativity [17].

For abstract rewrite systems, the *decreasing diagrams* technique of van Oostrom [20] subsumes all sufficient conditions for confluence. To use this technique for term rewrite systems, a well-founded order on the rewrite steps has to be supplied such that rewrite peaks can be completed into so-called decreasing diagrams.

We present two results in this paper. We show how to encode the rule-labeling heuristic of van Oostrom [22] for linear rewrite systems as a satisfiability problem. In this heuristic rewrite steps are labeled by the applied rewrite rule. By limiting the number of steps that may be used to complete local diagrams, we obtain a finite search problem which is readily transformed into a satisfiability problem. Any satisfying assignment returned by a modern SAT or SMT solver is then translated back into a concrete rule-labeling.

Our second and main result employs the decreasing diagrams technique to obtain a new confluence result for left-linear but not necessarily right-linear rewrite

* The research described in this paper is the starting point of FWF (Austrian Science Fund) project P22467 and the Grant-in-Aid for Young Scientists (B) 22700009 of the Japan Society for the Promotion of Science.

[1] `http://termination-portal.org/wiki/Termination_Competition`

systems. It requires that the rewrite steps involved in the generation of critical pairs are *relatively terminating* with respect to the rewrite system. This result can be viewed as a generalization of the two standard approaches for proving confluence: orthogonality and joinability of critical pairs for terminating systems. In the non-trivial proof we use the self-labeling heuristic in which rewrite steps are labeled by their starting term.

Throughout the paper we assume familiarity with the basics of term rewriting ([18]). In the next section we recall the decreasing diagrams technique and present a small variation which better serves our purposes. Section 3 is devoted to our main result. We prove that a locally confluent left-linear term rewrite system is confluent if there are no infinite rewrite sequences that involve infinitely many steps that were used in the generation of critical pairs. In Section 4 we explain how this result is implemented. Moreover, we show how the rule-labeling heuristic for decreasing diagrams can be transformed into a satisfiability problem. Section 5 presents experimental data. In Section 6 we conclude with suggestions for future research.

2 Decreasing Diagrams

We start this preliminary section by recalling the decreasing diagrams technique for abstract rewrite systems (ARSs) from [20]. We write $\langle A, \{\rightarrow_\alpha\}_{\alpha \in I}\rangle$ to denote the ARS $\langle A, \rightarrow\rangle$ where $\rightarrow$ is the union of $\rightarrow_\alpha$ for all $\alpha \in I$. If $J \subseteq I$ then $\rightarrow_J$ denotes the union of $\rightarrow_\alpha$ for all $\alpha \in J$.

Let $\mathcal{A} = \langle A, \{\rightarrow_\alpha\}_{\alpha \in I}\rangle$ be an ARS and let $>$ be a well-founded order on I. For every $\alpha \in I$ we write $\xrightarrow{\vee}_\alpha$ for the union of $\rightarrow_\beta$ for all $\beta < \alpha$. If $\alpha, \beta \in I$ then $\xrightarrow{\vee}_{\alpha\beta}$ denotes the union of $\xrightarrow{\vee}_\alpha$ and $\xrightarrow{\vee}_\beta$. Moreover, we write $\xrightarrow{\vee}{}^*_{\alpha\beta}$ for $(\xrightarrow{\vee}_{\alpha\beta})^*$. We say that α and β are *locally decreasing* with respect to $>$ and we write $\mathrm{LD}_>(\alpha, \beta)$ if

$${}_\alpha\!\leftarrow \cdot \rightarrow_\beta \;\subseteq\; \xrightarrow{\vee}{}^*_\alpha \cdot \rightarrow^=_\beta \cdot \xrightarrow{\vee}{}^*_{\alpha\beta} \cdot {}^{*}_{\alpha\beta}\!\xleftarrow{\vee} \cdot {}^{=}_{\alpha}\!\leftarrow \cdot {}^{*}_{\beta}\!\xleftarrow{\vee}$$

Graphically (dashed arrows are implicitly existentially quantified and double-headed arrows denote reflexive and transitive closure):

$$\begin{tikzcd}
 & & & \cdot \arrow[dlll, "\alpha"'] \arrow[drrr, "\beta"] & & & \\
\cdot \arrow[dr, dashed, twoheadrightarrow, "\vee"', "\alpha"] & & & \mathrm{LD}_>(\alpha,\beta) & & & \cdot \arrow[dl, dashed, twoheadrightarrow, "\vee", "\beta"'] \\
 & \cdot \arrow[dr, dashed, "="', "\beta"] & & & & \cdot \arrow[dl, dashed, "=", "\alpha"'] & \\
 & & \cdot \arrow[dr, dashed, twoheadrightarrow, "\vee"', "\alpha\beta"] & & \cdot \arrow[dl, dashed, twoheadrightarrow, "\vee", "\alpha\beta"'] & & \\
 & & & \cdot & & &
\end{tikzcd}$$

The ARS $\mathcal{A} = \langle A, \{\rightarrow_\alpha\}_{\alpha \in I}\rangle$ is locally decreasing if there exists a well-founded order $>$ on I such that $\mathrm{LD}(\alpha, \beta)$ for all $\alpha, \beta \in I$.

Van Oostrom [20] obtained the following result.

Theorem 1. *Every locally decreasing ARS is confluent.* □

Variations of this fundamental confluence result are presented in [3,12,13,22]. We present a version of Theorem 1 which is more suitable for our purposes.

Let $\mathcal{A} = \langle A, \{\to_\alpha\}_{\alpha \in I}\rangle$ be an ARS. Let $(>, \geqslant)$ consist of a well-founded order $>$ on I together with a quasi-order $\geqslant$ such that $\geqslant \cdot > \cdot \geqslant \subseteq >$. For every $\alpha \in I$ we write $\xrightarrow{\Vee}_\alpha$ for the union of $\to_\beta$ for all $\beta \leqslant \alpha$ and all $\beta < \alpha$. (Note that $> \subseteq \geqslant$ need not hold.) We say that α and β are locally decreasing with respect to $(>, \geqslant)$ and we write $\mathrm{LD}_{(>,\geqslant)}(\alpha, \beta)$ if

$${}_{\alpha}\!\leftarrow \cdot \to_\beta \;\subseteq\; \xrightarrow{\vee}{}^{*}_{\alpha} \cdot \xrightarrow{\Vee}{}^{=}_{\beta} \cdot \xrightarrow{\vee}{}^{*}_{\alpha\beta} \cdot {}^{*}_{\alpha\beta}\!\xleftarrow{\vee} \cdot {}^{=}_{\alpha}\!\xleftarrow{\Vee} \cdot {}^{*}_{\beta}\!\xleftarrow{\vee}$$

The ARS $\mathcal{A} = \langle A, \{\to_\alpha\}_{\alpha \in I}\rangle$ is *extended* locally decreasing if there exists $(>, \geqslant)$ such that $\mathrm{LD}_{(>,\geqslant)}(\alpha, \beta)$ for all $\alpha, \beta \in I$. Despite the name, from the proof of the following theorem we infer that every extended locally decreasing ARS has a locally decreasing presentation.

Theorem 2. *Every extended locally decreasing ARS is confluent.*

Proof. Let $\mathcal{A} = \langle A, \{\to_\alpha\}_{\alpha \in I}\rangle$ be extended locally decreasing with respect to $(>, \geqslant)$. We write C_α as the set of all $\beta \in I$ with $\alpha \geqslant \beta$ and $\alpha \not> \beta$. The set of all such C_α is denoted by $\mathcal{C}$. For every $C \in \mathcal{C}$ we write $\to_C$ for the union of $\to_\alpha$ for all $\alpha \in C$. The well-founded order $>$ on I can be lifted to $\mathcal{C}$: $C_\alpha > C_\beta$ if $\alpha > \beta$. If $C_\alpha = C_{\alpha'}$, $C_\beta = C_{\beta'}$, and $\alpha > \beta$ then $\alpha' \geqslant \alpha > \beta \geqslant \beta'$ and thus $\alpha' > \beta'$ because of the requirement $\geqslant \cdot > \cdot \geqslant \subseteq >$. Hence $>$ is well-defined on $\mathcal{C}$. If $\beta \leqslant \alpha$ and $\beta < \alpha$ then $\to_\beta \subseteq \to_{C_\beta} \subseteq \xrightarrow{\vee}_{C_\alpha}$. If $\beta \leqslant \alpha$ and $\beta \not< \alpha$ then $\beta \in C_\alpha$ and thus $\to_\beta \subseteq \to_{C_\alpha}$. Hence $\xrightarrow{\Vee}_\alpha \subseteq \xrightarrow{\vee}_{C_\alpha} \cup \to_{C_\alpha}$. Now consider arbitrary sets $C, D \in \mathcal{C}$ and let $\alpha \in C$ and $\beta \in D$. From the assumption $\mathrm{LD}_{(>,\geqslant)}(\alpha, \beta)$ we obtain

$${}_{\alpha}\!\leftarrow \cdot \to_\beta \;\subseteq\; \xrightarrow{\vee}{}^{*}_{\alpha} \cdot \xrightarrow{\Vee}{}^{=}_{\beta} \cdot \xrightarrow{\vee}{}^{*}_{\alpha\beta} \cdot {}^{*}_{\alpha\beta}\!\xleftarrow{\vee} \cdot {}^{=}_{\alpha}\!\xleftarrow{\Vee} \cdot {}^{*}_{\beta}\!\xleftarrow{\vee}$$

By construction, the latter relation is contained in

$$\xrightarrow{\vee}{}^{*}_{C} \cdot \xrightarrow{\vee}{}^{=}_{D} \cdot \xrightarrow{\vee}{}^{*}_{CD} \cdot {}^{*}_{CD}\!\xleftarrow{\vee} \cdot {}^{=}_{C}\!\xleftarrow{\vee} \cdot {}^{*}_{D}\!\xleftarrow{\vee}$$

Since

$${}_{C}\!\leftarrow \cdot \to_D \;=\; \bigcup_{\alpha \in C,\, \beta \in D} {}_{\alpha}\!\leftarrow \cdot \to_\beta$$

we conclude $\mathrm{LD}_{>}(C, D)$. According to Theorem 1, the ARS $\langle A, \{\to_C\}_{C \in \mathcal{C}}\rangle$ is confluent. Since

$$\bigcup_{\alpha \in I} \to_\alpha = \bigcup_{C \in \mathcal{C}} \to_C$$

it follows that $\mathcal{A}$ is confluent. □

We are interested in the application of Theorems 1 and 2 for proving confluence of term rewrite systems (TRSs). Many sufficient conditions for confluence of TRSs are based on critical pairs. Critical pairs are generated from overlaps. An overlap $(l_1 \to r_1, p, l_2 \to r_2)_\mu$ of a TRS $\mathcal{R}$ consists of variants $l_1 \to r_1$ and

$l_2 \to r_2$ of rules of $\mathcal{R}$ without common variables, a position $p \in \mathcal{P}\mathsf{os}_{\mathcal{F}}(l_2)$, and a most general unifier μ of l_1 and $l_2|_p$. If $p = \epsilon$ then we require that $l_1 \to r_1$ and $l_2 \to r_2$ are not variants. The induced critical pair is $(l_2\mu[r_1\mu]_p, r_2\mu)$. Following Dershowitz [5], we write $s \leftarrow\!\rtimes\!\rightarrow t$ to indicate that (s,t) is a critical pair.

In [22] van Oostrom proposed the *rule-labeling heuristic* in which rewrite steps are partitioned according to the employed rewrite rules. If one can find an order on the rules of a *linear* TRS such that every critical pair is locally decreasing, confluence is guaranteed. A formalization of this heuristic is given below where ${}_{\alpha}\!\leftarrow\!\rtimes\!\rightarrow_{\beta}$ denotes the set of critical pairs obtained from overlaps $(\alpha, p, \beta)_\mu$.

Theorem 3. *A linear TRS $\mathcal{R}$ is confluent if there exists a well-founded order $>$ on the rules of $\mathcal{R}$ such that ${}_{\alpha}\!\leftarrow\!\rtimes\!\rightarrow_{\beta} \subseteq \xrightarrow{\vee}{}^{*}_{\alpha} \cdot \xrightarrow{\veebar}{}^{=}_{\beta} \cdot \xrightarrow{\vee}{}^{*}_{\alpha\beta} \cdot {}_{\alpha\beta}^{\;*}\!\xleftarrow{\vee} \cdot {}_{\alpha}^{=}\!\xleftarrow{\veebar} \cdot {}_{\beta}^{*}\!\xleftarrow{\vee}$ for all rewrite rules $\alpha, \beta \in \mathcal{R}$. Here $\geqslant$ is the reflexive closure of $>$.* $\square$

The heuristic readily applies to the following example from [10].

Example 4. Consider the linear TRS $\mathcal{R}$ consisting of the rewrite rules

$$\begin{array}{llll}
1\colon & \mathsf{nats} \to \mathsf{0} : \mathsf{inc}(\mathsf{nats}) & 4\colon & \mathsf{inc}(x : y) \to \mathsf{s}(x) : \mathsf{inc}(y) \\
2\colon & \mathsf{hd}(x : y) \to x & 5\colon & \mathsf{inc}(\mathsf{tl}(\mathsf{nats})) \to \mathsf{tl}(\mathsf{inc}(\mathsf{nats})) \\
3\colon & \mathsf{tl}(x : y) \to y & &
\end{array}$$

There is one critical pair: $\mathsf{inc}(\mathsf{tl}(\mathsf{0} : \mathsf{inc}(\mathsf{nats}))) \xleftarrow{1} \mathsf{inc}(\mathsf{tl}(\mathsf{nats})) \xrightarrow{5} \mathsf{tl}(\mathsf{inc}(\mathsf{nats}))$. We have

$$\mathsf{inc}(\mathsf{tl}(\mathsf{0} : \mathsf{inc}(\mathsf{nats}))) \xrightarrow{3} \mathsf{inc}(\mathsf{inc}(\mathsf{nats}))$$

$$\mathsf{tl}(\mathsf{inc}(\mathsf{nats})) \xrightarrow{1} \mathsf{tl}(\mathsf{inc}(\mathsf{0} : \mathsf{inc}(\mathsf{nats}))) \xrightarrow{4} \mathsf{tl}(\mathsf{s}(\mathsf{0}) : \mathsf{inc}(\mathsf{inc}(\mathsf{nats}))) \xrightarrow{3} \mathsf{inc}(\mathsf{inc}(\mathsf{nats}))$$

Hence the critical pair is locally decreasing with respect to the rule-labeling heuristic together with the order $5 > 3, 4$.

The following example (Vincent van Oostrom, personal communication) shows that linearity in Theorem 3 cannot be weakened to left-linearity.

Example 5. Consider the TRS $\mathcal{R}$ consisting of the rewrite rules

$$1\colon \mathsf{f}(\mathsf{a},\mathsf{a}) \to \mathsf{c} \qquad 2\colon \mathsf{f}(\mathsf{b},x) \to \mathsf{f}(x,x) \qquad 3\colon \mathsf{f}(x,\mathsf{b}) \to \mathsf{f}(x,x) \qquad 4\colon \mathsf{a} \to \mathsf{b}$$

There are three critical pairs: $\mathsf{f}(\mathsf{a},\mathsf{b}) \xleftarrow{4} \mathsf{f}(\mathsf{a},\mathsf{a}) \xrightarrow{1} \mathsf{c}$, $\mathsf{f}(\mathsf{b},\mathsf{a}) \xleftarrow{4} \mathsf{f}(\mathsf{a},\mathsf{a}) \xrightarrow{1} \mathsf{c}$, and $\mathsf{f}(\mathsf{b},\mathsf{b}) \xleftarrow{2} \mathsf{f}(\mathsf{b},\mathsf{b}) \xrightarrow{3} \mathsf{f}(\mathsf{b},\mathsf{b})$. Since $\mathsf{f}(\mathsf{a},\mathsf{b}) \xrightarrow{3} \mathsf{f}(\mathsf{a},\mathsf{a}) \xrightarrow{1} \mathsf{c}$ and $\mathsf{f}(\mathsf{b},\mathsf{a}) \xrightarrow{2} \mathsf{f}(\mathsf{a},\mathsf{a}) \xrightarrow{1} \mathsf{c}$, it follows that the critical pairs are locally decreasing by taking the order $4 > 2, 3$. Nevertheless, the conversion $\mathsf{f}(\mathsf{b},\mathsf{b}) \leftarrow \mathsf{f}(\mathsf{b},\mathsf{a}) \leftarrow \mathsf{f}(\mathsf{a},\mathsf{a}) \to \mathsf{c}$ reveals that $\mathcal{R}$ is not confluent.

In the next section we impose a relative termination condition to obtain a confluence criterion for possibly non-right-linear TRSs.

3 Confluence via Relative Termination

Let $\mathcal{R}$ be a TRS. We denote the set

$$\{l_2\mu \to l_2\mu[r_1\mu]_p, l_2\mu \to r_2\mu \mid (l_1 \to r_1, p, l_2 \to r_2)_\mu \text{ is an overlap of } \mathcal{R}\}$$

of rewrite steps that give rise to critical pairs of $\mathcal{R}$ by $\mathsf{CPS}(\mathcal{R})$. The rules in $\mathsf{CPS}(\mathcal{R})$ are called *critical pair steps*. We say that $\mathcal{R}$ is relatively terminating with respect to $\mathcal{S}$ or that $\mathcal{R}/\mathcal{S}$ is terminating if the relation $\to_{\mathcal{R}/\mathcal{S}} = \to_{\mathcal{S}}^* \cdot \to_{\mathcal{R}} \cdot \to_{\mathcal{S}}^*$ is well-founded. The main result of this section (Theorem 16 below) states that a left-linear locally confluent TRS $\mathcal{R}$ is confluent if $\mathsf{CPS}(\mathcal{R})$ is relatively terminating with respect to $\mathcal{R}$. Since $\mathsf{CPS}(\mathcal{R})$ is empty for every orthogonal TRS $\mathcal{R}$, this yields a generalization of orthogonality. In the proof we use decreasing diagrams with the *self-labeling heuristic* in which rewrite steps are labeled by their starting term. A key problem when trying to prove confluence in the absence of termination is the handling of duplicating rules. Parallel rewrite steps are typically used for this purpose [11,17]. To anticipate future developments (cf. Section 6) we use multi-steps instead. However, first we present a special case of our main result in which duplicating rules are taken care of by requiring them to be relatively terminating with respect to the non-duplicating ones.

Theorem 6. *Let $\mathcal{R}$ be a left-linear TRS. Let $\mathcal{R}_{\mathsf{d}}$ be the subset of duplicating rules and $\mathcal{R}_{\mathsf{nd}}$ the subset of non-duplicating rules in $\mathcal{R}$. The TRS $\mathcal{R}$ is confluent if $\leftarrow\!\rtimes\!\to\ \subseteq\ \downarrow$ and $\mathsf{CPS}(\mathcal{R}) \cup \mathcal{R}_{\mathsf{d}}$ is relatively terminating with respect to $\mathcal{R}_{\mathsf{nd}}$.*

Proof. We label rewrite steps by their starting term. Labels are compared with respect to the strict order $> \,=\, \to^+_{(\mathsf{CPS}(\mathcal{R})\cup\mathcal{R}_{\mathsf{d}})/\mathcal{R}_{\mathsf{nd}}}$ and the quasi-order $\geqslant \,=\, \to^*_{\mathcal{R}}$. Note that $>$ is well-founded by the assumption that $\mathsf{CPS}(\mathcal{R}) \cup \mathcal{R}_{\mathsf{d}}$ is relatively terminating with respect to $\mathcal{R}_{\mathsf{nd}}$. We show that all local peaks of $\mathcal{R}$ are extended locally decreasing. Let $s \to t_1$ and $s \to t_2$ by applying the rewrite rules $l_1 \to r_1$ and $l_2 \to r_2$ at the positions p_1 and p_2. We may assume that $l_1 \to r_1$ and $l_2 \to r_2$ do not share variables and thus there exists a substitution σ such that $s = s[l_1\sigma]_{p_1} = s[l_2\sigma]_{p_2}$, $t_1 = s[r_1\sigma]_{p_1}$, and $t_2 = s[r_2\sigma]_{p_2}$. We distinguish three cases.

1. If $p_1 \parallel p_2$ then $t_1 \to u \leftarrow t_2$ for the term $u = s[r_1\sigma, r_2\sigma]_{p_1,p_2}$. We have $s > t_1$ if $l_1 \to r_1$ is duplicating and $s \geqslant t_1$ if $l_1 \to r_1$ is non-duplicating. So in both cases we have $t_1 \xrightarrow{\mathbb{V}}_s u$. Similarly, $t_2 \xrightarrow{\mathbb{V}}_s u$ and thus we have local decreasingness.
2. Suppose the redexes $l_1\sigma$ at position p_1 and $l_2\sigma$ at position p_2 overlap. If $p_1 = p_2$ and $l_1 \to r_1$ and $l_2 \to r_2$ are variants then $t_1 = t_2$ and there is nothing to prove. Assume without loss of generality that $p_1 \leqslant p_2$. There exists a substitution τ such that $t_1 = s[v\tau]_{p_1}$ and $t_2 = s[u\tau]_{p_1}$ with $u \leftarrow\!\rtimes\!\to v$. By assumption $u \downarrow v$ and hence also $t_1 \downarrow t_2$. Every label a in the valley between t_1 and t_2 satisfies $t_1 \geqslant a$ or $t_2 \geqslant a$. Since $s \to_{\mathsf{CPS}(\mathcal{R})} t_1$ and $s \to_{\mathsf{CPS}(\mathcal{R})} t_2$, it follows that $s > t_1, t_2$. Hence $s > a$ for every label a in the valley between t_1 and t_2. Consequently, local decreasingness holds.

3. In the remaining case we have a variable overlap. Assume without loss of generality that $p_1 < p_2$. Let x be the variable in l_1 whose position is above $p_2 \setminus p_1$. Due to linearity of l_1 we have $t_1 \to^* u \leftarrow t_2$ for some term u. The number of steps in the sequence from t_1 to u equals the number of occurrences of the variable x in r_1. If this number is not more than one then local decreasingness is obtained as in the first case. If this number is more than one then $l_1 \to r_1$ is duplicating and hence $s > t_1$. Therefore $s > a$ for every term a in the sequence from t_1 to u. Moreover $s > t_2$ or $s \geqslant t_2$. Hence also in this case we have local decreasingness. $\square$

Example 7. Consider the TRS $\mathcal{R}$ from [9, p.28] consisting of the rewrite rules

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

The only critical pair $\mathsf{f}(\mathsf{g}(\mathsf{g}(\mathsf{a}))) \leftarrow\!\rtimes\!\rightarrow \mathsf{f}(\mathsf{h}(\mathsf{a},\mathsf{a}))$ is clearly joinable. The TRS $\mathsf{CPS}(\mathcal{R}) \cup \mathcal{R}_\mathsf{d}$ consists of the rewrite rules

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

and can be shown to be relatively terminating with respect to $\mathcal{R}_\mathsf{nd}$ using the method described at the beginning of Section 4. Hence the confluence of $\mathcal{R}$ is concluded by Theorem 6.

The following example[2] shows that left-linearity is essential in Theorem 6.

Example 8. Consider the non-left-linear TRS $\mathcal{R}$

$$\mathsf{f}(x,x) \to \mathsf{a} \qquad \mathsf{f}(x,\mathsf{g}(x)) \to \mathsf{b} \qquad \mathsf{c} \to \mathsf{g}(\mathsf{c})$$

from [11]. Since $\mathsf{CPS}(\mathcal{R}) \cup \mathcal{R}_\mathsf{d}$ is empty, termination of $(\mathsf{CPS}(\mathcal{R}) \cup \mathcal{R}_\mathsf{d})/\mathcal{R}_\mathsf{nd}$ is trivial. However, $\mathcal{R}$ is not confluent because the term $\mathsf{f}(\mathsf{c},\mathsf{c})$ has two distinct normal forms.

The termination of $(\mathsf{CPS}(\mathcal{R}) \cup \mathcal{R}_\mathsf{d})/\mathcal{R}_\mathsf{nd}$ can be weakened to the termination of $\mathsf{CPS}(\mathcal{R})/\mathcal{R}$. In the proof of Theorem 6 we showed the local decreasingness of $\to_\mathcal{R}$. The following example shows that this no longer holds under the weakened termination assumption.

Example 9. Consider the orthogonal TRS $\mathcal{R}$ consisting of the two rules $\mathsf{f}(x) \to \mathsf{g}(x,x)$ and $\mathsf{a} \to \mathsf{b}$. Consider the local peak $\mathsf{f}(\mathsf{b}) \leftarrow \mathsf{f}(\mathsf{a}) \to \mathsf{g}(\mathsf{a},\mathsf{a})$. There are two ways to complete the diagram:

$$\begin{tikzcd}
\mathsf{f}(\mathsf{a}) \arrow[r, "\mathsf{f}(\mathsf{a})"] \arrow[dd, "\mathsf{f}(\mathsf{a})"'] & \mathsf{g}(\mathsf{a},\mathsf{a}) \arrow[d, "\mathsf{g}(\mathsf{a},\mathsf{a})"] \\
 & \mathsf{g}(\mathsf{b},\mathsf{a}) \arrow[d, "\mathsf{g}(\mathsf{b},\mathsf{a})"] \\
\mathsf{f}(\mathsf{b}) \arrow[r, "\mathsf{f}(\mathsf{b})"'] & \mathsf{g}(\mathsf{b},\mathsf{b})
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathsf{f}(\mathsf{a}) \arrow[r, "\mathsf{f}(\mathsf{a})"] \arrow[dd, "\mathsf{f}(\mathsf{a})"'] & \mathsf{g}(\mathsf{a},\mathsf{a}) \arrow[d, "\mathsf{g}(\mathsf{a},\mathsf{a})"] \\
 & \mathsf{g}(\mathsf{a},\mathsf{b}) \arrow[d, "\mathsf{g}(\mathsf{a},\mathsf{b})"] \\
\mathsf{f}(\mathsf{b}) \arrow[r, "\mathsf{f}(\mathsf{b})"'] & \mathsf{g}(\mathsf{b},\mathsf{b})
\end{tikzcd}$$

[2] This example contradicts [12, Theorem 4].

Since $\mathsf{CPS}(\mathcal{R})$ is empty, neither of them is extended locally decreasing with respect to the order in the proof of Theorem 6.

To address the problem, we first recall *multi-steps* (cf. [18]).

Definition 10. *Let $\mathcal{R}$ be a TRS. The* multi-step *relation $\multimap_{\mathcal{R}}$ (or simply $\multimap$) is inductively defined as follows:*

(1) $x \multimap_{\mathcal{R}} x$ for all variables x,
(2) $f(s_1,\dots,s_n) \multimap_{\mathcal{R}} f(t_1,\dots,t_n)$ if for each i we have $s_i \multimap_{\mathcal{R}} t_i$, and
(3) $l\sigma \multimap_{\mathcal{R}} r\tau$ if $l \to r \in \mathcal{R}$ and $\sigma \multimap_{\mathcal{R}} \tau$.

where $\sigma \multimap_{\mathcal{R}} \tau$ if $x\sigma \multimap_{\mathcal{R}} x\tau$ for all variables x.

The following result is well-known ([18, Lemma 4.7.12]).

Lemma 11. *For every TRS $\mathcal{R}$ we have $\to_{\mathcal{R}} \subseteq \multimap_{\mathcal{R}} \subseteq \to^*_{\mathcal{R}}$.* □

The following lemma relates $\multimap_{\mathcal{R}}$ to $\to_{\mathsf{CPS}(\mathcal{R})/\mathcal{R}}$. It is the key to prove our main result.

Lemma 12. *Let $\mathcal{R}$ be a TRS and $l \to r$ a left-linear rule in $\mathcal{R}$. If $l\sigma \multimap_{\mathcal{R}} t$ then one of the following conditions holds:*

(a) $t \in \{l\tau, r\tau\}$ and $\sigma \multimap_{\mathcal{R}} \tau$ for some τ,
(b) $l\sigma \to_{\mathsf{CPS}(\mathcal{R})} \cdot \multimap_{\mathcal{R}} t$ and $l\sigma \to_{\mathsf{CPS}(\mathcal{R})} r\sigma$.

Proof. We may write $l\sigma = C[s_1,\dots,s_n] \multimap_{\mathcal{R}} C[t_1,\dots,t_n] = t$ where $s_i \multimap_{\mathcal{R}} t_i$ is obtained by case (3) in the definition of $\multimap_{\mathcal{R}}$ for all $1 \leqslant i \leqslant n$. If $n = 0$ then $l\sigma = t$ and hence we can take $\tau = \sigma$ to satisfy condition (a). So let $n > 0$. Let p_i be the position of s_i in $l\sigma$. We distinguish two cases.

- Suppose that $p_1,\dots,p_n \notin \mathcal{P}\mathsf{os}_{\mathcal{F}}(l)$. We define a substitution τ as follows. For $x \in \mathcal{V}\mathsf{ar}(l)$ let q be the (unique) position in $\mathcal{P}\mathsf{os}_{\mathcal{V}}(l)$ such that $l|_q = x$. Let $P = \{p_i \mid p_i \geqslant q\}$ be the set of positions in $l\sigma$ of those terms $s_1,\dots,s_n$ that occur in $\sigma(x)$. We define $\tau(x)$ as the term that is obtained from $\sigma(x)$ by replacing for all $p_i \in P$ the subterm s_i at position $p_i \backslash q$ with t_i. We have $t = l\tau$ and $\sigma \multimap_{\mathcal{R}} \tau$, so condition (a) is satisfied.
- In the remaining case at least one position among $p_1,\dots,p_n$ belongs to $\mathcal{P}\mathsf{os}_{\mathcal{F}}(l)$. Without loss of generality we assume that $p_1 \in \mathcal{P}\mathsf{os}_{\mathcal{F}}(l)$. Since $s_1 \multimap_{\mathcal{R}} t_1$ is obtained by case (3), $s_1 = l_1\mu$ and $t_1 = r_1\nu$ for some rewrite rule $l_1 \to r_1$ and substitutions μ and ν with $\mu \multimap_{\mathcal{R}} \nu$. We assume that $l_1 \to r_1$ and $l \to r$ share no variables. Hence we may assume that $\mu = \sigma$. We distinguish two further cases.
  - If $l_1 \to r_1$ and $l \to r$ are variants and $p_1 = \epsilon$ then $n = 1$, $C = \Box$, and $l\sigma = s_1 = l_1\sigma \multimap_{\mathcal{R}} r_1\nu = t$. Because $l_1 \to r_1$ and $l \to r$ are variants, there exists a substitution τ such that $r\tau = r_1\nu$ and $\sigma \multimap_{\mathcal{R}} \tau$. So in this case condition (a) is satisfied.

- If $l_1 \to r_1$ and $l \to r$ are not variants or $p_1 \neq \epsilon$ then there exists an overlap $(l_1 \to r_1, p_1, l \to r)_\theta$ such that $l\sigma = l\sigma[l_1\sigma]_{p_1}$ is an instance of $l\theta = l\theta[l_1\theta]_{p_1}$. The TRS $\mathsf{CPS}(\mathcal{R})$ contains the rules $l\theta \to l\theta[r_1\theta]_{p_1}$ and $l\theta \to r\theta$. The latter rule is used to obtain $l\sigma \to_{\mathsf{CPS}(\mathcal{R})} r\sigma$. An application of the former rule yields $l\sigma \to_{\mathsf{CPS}(\mathcal{R})} l\sigma[r_1\sigma]_{p_1}$. From $\sigma \mathrel{\circ\!\!\!\longrightarrow}_{\mathcal{R}} \nu$ we infer that $r_1\sigma \mathrel{\circ\!\!\!\longrightarrow}_{\mathcal{R}} r_1\nu = t_1$. Hence $l\sigma \to_{\mathsf{CPS}(\mathcal{R})} l\sigma[r_1\sigma]_{p_1} \mathrel{\circ\!\!\!\longrightarrow}_{\mathcal{R}} l\sigma[t_1]_{p_1} = C[t_1, s_2, \ldots, s_n] \mathrel{\circ\!\!\!\longrightarrow}_{\mathcal{R}}^* C[t_1, \ldots, t_n] = t$. The $\mathrel{\circ\!\!\!\longrightarrow}_{\mathcal{R}}$-steps can be combined into a single one and hence condition (b) is satisfied. □

The following example shows that both conditions in Lemma 12 can occur.

Example 13. Consider the TRS $\mathcal{R}$ consisting of the rules $\mathsf{f}(\mathsf{g}(x), y) \to \mathsf{h}(x, y)$, $\mathsf{g}(\mathsf{a}) \to \mathsf{b}$, and $\mathsf{a} \to \mathsf{c}$. Let $l \to r$ be the first rule, $t = \mathsf{f}(\mathsf{b}, \mathsf{c})$, and $\sigma = \{x, y \mapsto \mathsf{a}\}$. We have $l\sigma \mathrel{\circ\!\!\!\longrightarrow}_{\mathcal{R}} t$ with t satisfying condition (b) in Lemma 12: $l\sigma \to_{\mathsf{CPS}(\mathcal{R})} \mathsf{f}(\mathsf{b}, \mathsf{a}) \mathrel{\circ\!\!\!\longrightarrow}_{\mathcal{R}} t$ and $l\sigma \to_{\mathsf{CPS}(\mathcal{R})} r\sigma$. Note that condition (a) is not satisfied. If we take $t = \mathsf{f}(\mathsf{g}(\mathsf{a}), \mathsf{c})$ or $t = \mathsf{h}(\mathsf{g}(\mathsf{c}), \mathsf{c})$ then condition (a) is satisfied but condition (b) is not.

The following example shows the necessity of left-linearity in the preceding lemma.

Example 14. Consider the TRS $\mathcal{R}$ consisting of the rewrite rules $\mathsf{f}(x, x) \to \mathsf{b}$ and $\mathsf{a} \to \mathsf{b}$. Let $l \to r$ be the former rule, $t = \mathsf{f}(\mathsf{a}, \mathsf{b})$, and $\sigma = \{x \mapsto \mathsf{a}\}$. We have $l\sigma \mathrel{\circ\!\!\!\longrightarrow}_{\mathcal{R}} t$ but t satisfies neither condition in Lemma 12.

The final preliminary lemma states some obvious closure properties.

Lemma 15. *Let $>$ and $\geqslant$ be closed under contexts.*

1. *If $t \overset{\vee}{\mathrel{\circ\!\!\!\longrightarrow}}{}_s^* u$ then $C[t] \overset{\vee}{\mathrel{\circ\!\!\!\longrightarrow}}{}_{C[s]}^* C[u]$.*
2. *If $t \overset{\underline{\vee}}{\mathrel{\circ\!\!\!\longrightarrow}}{}_s^= u$ then $C[t] \overset{\underline{\vee}}{\mathrel{\circ\!\!\!\longrightarrow}}{}_{C[s]}^= C[u]$.*
3. *Let $\geqslant\, = \,\to_{\mathcal{R}}^*$ and $\geqslant \cdot > \cdot \geqslant\, \subseteq\, >$. If $s > t$ and $t \mathrel{\circ\!\!\!\longrightarrow}^* u$ then $t \overset{\vee}{\mathrel{\circ\!\!\!\longrightarrow}}{}_s^* u$.*

Proof. Straightforward. □

After these preliminaries we are ready for the main result. In order to anticipate future developments (see Section 6), we avoid the use of advanced results from the confluence literature in the proof.

Theorem 16. *A left-linear TRS $\mathcal{R}$ is confluent if $\leftarrow\!\rtimes\!\rightarrow\, \subseteq\, \downarrow$ and $\mathsf{CPS}(\mathcal{R})/\mathcal{R}$ is terminating.*

Proof. Because of Lemma 11, it is sufficient to prove confluence of $\mathrel{\circ\!\!\!\longrightarrow}_{\mathcal{R}}$. We show that the relation $\mathrel{\circ\!\!\!\longrightarrow}_{\mathcal{R}}$ is extended locally decreasing with respect to the source labeling. Labels are compared with respect to the strict order $> \,=\, \to^+_{\mathsf{CPS}(\mathcal{R})/\mathcal{R}}$ and the quasi-order $\geqslant\, =\, \to_{\mathcal{R}}^*$. We show that

$$_s{\longleftarrow\!\!\!\circ} \cdot \mathrel{\circ\!\!\!\longrightarrow}_s \;\subseteq\; \overset{\underline{\vee}}{\mathrel{\circ\!\!\!\longrightarrow}}{}_s^= \cdot \overset{\vee}{\mathrel{\circ\!\!\!\longrightarrow}}{}_s^* \cdot {}_s^*\overset{\vee}{\longleftarrow\!\!\!\circ} \cdot {}_s^=\overset{\underline{\vee}}{\longleftarrow\!\!\!\circ}$$

for all terms s by well-founded induction on the order $(> \cup \rhd)^+$. In the base case s is a variable and the inclusion trivially holds. Let $s = f(s_1, \ldots, s_n)$. Suppose $t \longleftarrow\!\!\!\circ\; s \mathrel{\circ\!\!\!\longrightarrow} u$. We distinguish the following cases, depending on the derivation of $s \mathrel{\circ\!\!\!\longrightarrow} t$ and $s \mathrel{\circ\!\!\!\longrightarrow} u$.

– Neither $s \mathrel{-\!\!\circ\!\!\rightarrow} t$ nor $s \mathrel{-\!\!\circ\!\!\rightarrow} u$ is obtained by (1), because s is not a variable. Suppose both $s \mathrel{-\!\!\circ\!\!\rightarrow} t$ and $s \mathrel{-\!\!\circ\!\!\rightarrow} u$ are obtained by (2). Then t and u can be written as $f(t_1, \ldots, t_n)$ and $f(u_1, \ldots, u_n)$. Fix $i \in \{1, \ldots, n\}$. We have $t_i \mathrel{\leftarrow\!\!\circ\!\!-} s_i \mathrel{-\!\!\circ\!\!\rightarrow} u_i$. By the induction hypothesis there exist t_i', u_i', and v_i such that

$$t_i \overset{\underline{\vee}}{\mathrel{-\!\!\circ\!\!\rightarrow}}{}^{=}_{s_i} t_i' \overset{\vee}{\mathrel{-\!\!\circ\!\!\rightarrow}}{}^{*}_{s_i} v_i \;{}_{s_i}^{*}\overset{\vee}{\mathrel{\leftarrow\!\!\circ\!\!-}} u_i' \;{}_{s_i}^{=}\overset{\underline{\vee}}{\mathrel{\leftarrow\!\!\circ\!\!-}} u_i$$

With repeated applications of Lemma 15(1,2) we obtain

$$t \overset{\underline{\vee}}{\mathrel{-\!\!\circ\!\!\rightarrow}}{}^{=}_{s} f(t_1', \ldots, t_n') \overset{\vee}{\mathrel{-\!\!\circ\!\!\rightarrow}}{}^{*}_{s} f(v_1, \ldots, v_n) \;{}_{s}^{*}\overset{\vee}{\mathrel{\leftarrow\!\!\circ\!\!-}} f(u_1', \ldots, u_n') \;{}_{s}^{=}\overset{\underline{\vee}}{\mathrel{\leftarrow\!\!\circ\!\!-}} u$$

– Suppose $s \mathrel{-\!\!\circ\!\!\rightarrow} t$ or $s \mathrel{-\!\!\circ\!\!\rightarrow} u$ is obtained by (3). Without loss of generality we assume that $s \mathrel{-\!\!\circ\!\!\rightarrow} t$ is obtained by (3), i.e., $s = l\sigma$, $t = r\tau$, and $\sigma \mathrel{-\!\!\circ\!\!\rightarrow} \tau$. Following Lemma 12, we distinguish the following two cases for $l\sigma \mathrel{-\!\!\circ\!\!\rightarrow} u$.

- Suppose $u \in \{l\mu, r\mu\}$ for some μ with $\sigma \mathrel{-\!\!\circ\!\!\rightarrow} \mu$. Fix $x \in \mathcal{V}\mathsf{ar}(l)$. We have $x\tau \mathrel{\leftarrow\!\!\circ\!\!-} x\sigma \mathrel{-\!\!\circ\!\!\rightarrow} x\mu$. By the induction hypothesis there exist terms t_x, u_x, and v_x such that

$$x\tau \overset{\underline{\vee}}{\mathrel{-\!\!\circ\!\!\rightarrow}}{}^{=}_{x\sigma} t_x \overset{\vee}{\mathrel{-\!\!\circ\!\!\rightarrow}}{}^{*}_{x\sigma} v_x \;{}_{x\sigma}^{*}\overset{\vee}{\mathrel{\leftarrow\!\!\circ\!\!-}} u_x \;{}_{x\sigma}^{=}\overset{\underline{\vee}}{\mathrel{\leftarrow\!\!\circ\!\!-}} x\mu$$

  Define substitutions τ', ν, and μ' as follows: $\tau'(x) = t_x$, $\nu(x) = v_x$, and $\mu'(x) = u_x$ for all $x \in \mathcal{V}\mathsf{ar}(l)$, and $\tau'(x) = \nu(x) = \mu'(x) = x$ for all $x \notin \mathcal{V}\mathsf{ar}(l)$. We obtain

$$t \overset{\underline{\vee}}{\mathrel{-\!\!\circ\!\!\rightarrow}}{}^{=}_{s} r\tau' \overset{\vee}{\mathrel{-\!\!\circ\!\!\rightarrow}}{}^{*}_{s} r\nu \;{}_{s}^{*}\overset{\vee}{\mathrel{\leftarrow\!\!\circ\!\!-}} r\mu' \;{}_{s}^{=}\overset{\underline{\vee}}{\mathrel{\leftarrow\!\!\circ\!\!-}} u$$

  by repeated applications of Lemma 15(1,2).
- In the remaining case we have $s \rightarrow_{\mathsf{CPS}(\mathcal{R})} u' \mathrel{-\!\!\circ\!\!\rightarrow} u$ for some term u' as well as $s \rightarrow_{\mathsf{CPS}(\mathcal{R})} r\sigma$. Clearly $r\sigma \mathrel{-\!\!\circ\!\!\rightarrow} r\tau = t$. Since $\mathcal{R}$ is locally confluent (due to $\leftarrow \rtimes \rightarrow \subseteq \downarrow$), there exists a term v' such that $r\sigma \rightarrow^* v' \;{}^*\!\!\leftarrow u'$ and thus also $r\sigma \mathrel{-\!\!\circ\!\!\rightarrow}^* v' \;{}^*\!\!\mathrel{\leftarrow\!\!\circ\!\!-} u'$. We have $s > r\sigma$ and $s > u'$. Lemma 15(3) ensures that $r\sigma \overset{\vee}{\mathrel{-\!\!\circ\!\!\rightarrow}}{}^{*}_{s} v' \;{}_{s}^{*}\overset{\vee}{\mathrel{\leftarrow\!\!\circ\!\!-}} u'$. For every term v with $s > v$ we have

$${}_{v}\!\mathrel{\leftarrow\!\!\circ\!\!-} \cdot \mathrel{-\!\!\circ\!\!\rightarrow}_{v} \;\subseteq\; \overset{\underline{\vee}}{\mathrel{-\!\!\circ\!\!\rightarrow}}{}^{=}_{v} \cdot \overset{\vee}{\mathrel{-\!\!\circ\!\!\rightarrow}}{}^{*}_{v} \cdot {}_{v}^{*}\overset{\vee}{\mathrel{\leftarrow\!\!\circ\!\!-}} \cdot {}_{v}^{=}\overset{\underline{\vee}}{\mathrel{\leftarrow\!\!\circ\!\!-}}$$

  by the induction hypothesis. Hence the ARS $\langle \mathcal{T}(\mathcal{F}, \mathcal{V}), \{\mathrel{-\!\!\circ\!\!\rightarrow}_v\}_{v<s}\rangle$ is locally decreasing and therefore the relation

$$\overset{\vee}{\mathrel{-\!\!\circ\!\!\rightarrow}}_{s} \;=\; \bigcup_{v<s} \mathrel{-\!\!\circ\!\!\rightarrow}_{v}$$

  is confluent. This is used to obtain the diagram

$$\begin{tikzcd}
s \arrow[r, "\mathsf{CPS}(\mathcal{R})"] \arrow[d, "\mathsf{CPS}(\mathcal{R})"'] & u' \arrow[r, "\overset{\vee}{\circ}", "s"'] \arrow[d, twoheadrightarrow, ">\circ\, s"] & u \arrow[dd, twoheadrightarrow, ">\circ\, s"] \\
r\sigma \arrow[r, twoheadrightarrow, "\overset{\vee}{\circ}", "s"'] \arrow[d, ">\circ\, s"'] \arrow[dr, phantom, "\mathrm{CR}(\xrightarrow{\overset{\vee}{\circ}}_s)"] & v' \arrow[d, twoheadrightarrow, ">\circ\, s"] \arrow[r, phantom, "\mathrm{CR}(\xrightarrow{\overset{\vee}{\circ}}_s)"] & \\
t \arrow[r, twoheadrightarrow, "\overset{\vee}{\circ}", "s"'] & \cdot \arrow[r, twoheadrightarrow, "\overset{\vee}{\circ}", "s"'] & \cdot
\end{tikzcd}$$

from which we conclude that $t \xrightarrow{\overset{\vee}{\circ}}{}^{*}_{s} \cdot {}^{*}_{s}\xleftarrow{\overset{\vee}{\circ}} u$. □

The above result can also be proved using Lemma 12 in connection with the conversion version of decreasing diagrams together with the predecessor labeling diagrams [22] in which steps $s \xrightarrow{\circ} t$ are labeled by terms u such that $u \to^{*} s$ (Vincent van Oostrom, personal communication).

Example 17. Suppose we extend the TRS $\mathcal{R}$ of Example 4 by the rewrite rule

$$\mathsf{d}(x : y) \to x : (x : \mathsf{d}(y))$$

The resulting TRS $\mathcal{R}'$ has the same critical pair as $\mathcal{R}$ and $\mathsf{CPS}(\mathcal{R}')$ consists of

$$\mathsf{inc}(\mathsf{tl}(\mathsf{nats})) \to \mathsf{tl}(\mathsf{inc}(\mathsf{nats})) \qquad \mathsf{inc}(\mathsf{tl}(\mathsf{nats})) \to \mathsf{inc}(\mathsf{tl}(\mathsf{0} : \mathsf{inc}(\mathsf{nats})))$$

By taking the matrix interpretation ([7])

$$\begin{array}{lll}
\mathsf{inc}_{\mathcal{M}}(\boldsymbol{x}) = \begin{pmatrix}1&0\\1&0\end{pmatrix}\boldsymbol{x} & \mathsf{hd}_{\mathcal{M}}(\boldsymbol{x}) = \boldsymbol{x} & \mathsf{0}_{\mathcal{M}} = \begin{pmatrix}0\\0\end{pmatrix} \\
\mathsf{nats}_{\mathcal{M}} = \begin{pmatrix}0\\1\end{pmatrix} & \mathsf{tl}_{\mathcal{M}}(\boldsymbol{x}) = \begin{pmatrix}1&1\\1&0\end{pmatrix}\boldsymbol{x} & \mathsf{s}_{\mathcal{M}}(\boldsymbol{x}) = \begin{pmatrix}1&1\\0&0\end{pmatrix}\boldsymbol{x} \\
\mathsf{d}_{\mathcal{M}}(\boldsymbol{x}) = \begin{pmatrix}1&1\\1&1\end{pmatrix}\boldsymbol{x} & :_{\mathcal{M}}(\boldsymbol{x},\boldsymbol{y}) = \begin{pmatrix}1&1\\1&1\end{pmatrix}\boldsymbol{x} + \boldsymbol{y} &
\end{array}$$

we obtain $\mathcal{R}' \subseteq \geqslant_{\mathcal{M}}$ and $\mathsf{CPS}(\mathcal{R}') \subseteq >_{\mathcal{M}}$:

$$[\mathsf{inc}(\mathsf{tl}(\mathsf{nats}))]_{\mathcal{M}} = \begin{pmatrix}1\\1\end{pmatrix} > \begin{pmatrix}0\\0\end{pmatrix} = [\mathsf{tl}(\mathsf{inc}(\mathsf{nats}))]_{\mathcal{M}} = [\mathsf{inc}(\mathsf{tl}(\mathsf{0} : \mathsf{inc}(\mathsf{nats})))]_{\mathcal{M}}$$

Hence $\mathsf{CPS}(\mathcal{R}')/\mathcal{R}'$ is terminating and Theorem 16 yields the confluence of $\mathcal{R}'$. Note that Theorem 6 does not apply because $\mathsf{CPS}(\mathcal{R}') \cup \mathcal{R}'_{\mathsf{d}}$ is not relatively terminating with respect to $\mathcal{R}'_{\mathsf{nd}}$; consider the term $\mathsf{d}(\mathsf{nats})$.

Replacing $\mathsf{CPS}(\mathcal{R})$ in Theorem 16 by $\mathsf{CPS}'(\mathcal{R}) = \{l_1\mu \to r_1\mu, l_2\mu \to r_2\mu \mid (l_1 \to r_1, p, l_2 \to r_2)_\mu$ is an overlap of $\mathcal{R}\}$ yields a correct but strictly weaker confluence criterion as termination of $\mathsf{CPS}'(\mathcal{R})/\mathcal{R}$ implies termination of $\mathsf{CPS}(\mathcal{R})/\mathcal{R}$ but not vice versa; $\mathsf{CPS}'(\mathcal{R}')/\mathcal{R}'$ in Example 17 is not terminating.

The next example explains why one cannot replace $\mathsf{CPS}(\mathcal{R})$ by one of its subsets $\mathsf{CPS}_o(\mathcal{R}) = \{l_2\mu \to r_2\mu \mid (l_1 \to r_1, p, l_2 \to r_2)_\mu$ is an overlap of $\mathcal{R}\}$ and $\mathsf{CPS}_i(\mathcal{R}) = \{l_2\mu \to l_2\mu[r_1\mu]_p \mid (l_1 \to r_1, p, l_2 \to r_2)_\mu$ is an overlap of $\mathcal{R}\}$.

Example 18. Consider the TRSs $\mathcal{R}_1 = \{\mathsf{f}(\mathsf{a}) \to \mathsf{c}, \mathsf{f}(\mathsf{b}) \to \mathsf{d}, \mathsf{a} \to \mathsf{b}, \mathsf{b} \to \mathsf{a}\}$ and $\mathcal{R}_2 = \{\mathsf{a} \to \mathsf{c}, \mathsf{b} \to \mathsf{d}, \mathsf{f}(\mathsf{a}) \to \mathsf{f}(\mathsf{b}), \mathsf{f}(\mathsf{b}) \to \mathsf{f}(\mathsf{a})\}$. Both TRSs are locally confluent but not confluent. We have $\mathsf{CPS}_o(\mathcal{R}_1) = \{\mathsf{f}(\mathsf{a}) \to \mathsf{c}, \mathsf{f}(\mathsf{b}) \to \mathsf{d}\}$ and $\mathsf{CPS}_i(\mathcal{R}_2) = \{\mathsf{f}(\mathsf{a}) \to \mathsf{f}(\mathsf{c}), \mathsf{f}(\mathsf{b}) \to \mathsf{f}(\mathsf{d})\}$. It is easy to see that $\mathsf{CPS}_o(\mathcal{R}_1)/\mathcal{R}_1$ and $\mathsf{CPS}_i(\mathcal{R}_2)/\mathcal{R}_2$ are terminating.

An easy extension of our main result is obtained by excluding critical pair steps from $\mathsf{CPS}(\mathcal{R})$ that give rise to trivial critical pair steps. The proof is based on the observation that Lemma 12 still holds for this modification of $\mathsf{CPS}(\mathcal{R})$.

4 Automation

Concerning the automation of Theorem 16, for checking relative termination we use the following criteria of Geser [8]:

Theorem 19. *For TRSs $\mathcal{R}$ and $\mathcal{S}$, $\mathcal{R}/\mathcal{S}$ is terminating if*

1. *$\mathcal{R} = \varnothing$,*
2. *$\mathcal{R} \cup \mathcal{S}$ is terminating, or*
3. *there exist a well-founded order $>$ and a quasi-order $\geqslant$ such that $>$ and $\geqslant$ are closed under contexts and substitutions, $\geqslant \cdot > \cdot \geqslant \subseteq >$, $\mathcal{R} \cup \mathcal{S} \subseteq \geqslant$, and $(\mathcal{R} \setminus >)/(\mathcal{S} \setminus >)$ is terminating.*

Based on this result, termination of $\mathsf{CPS}(\mathcal{R})/\mathcal{R}$ is shown by repeatedly using the last condition to simplify $\mathsf{CPS}(\mathcal{R})$ and $\mathcal{R}$. As soon as the first condition applies, termination is concluded. If the first condition does not apply and the third condition does not make progress, we try to establish termination of $\mathsf{CPS}(\mathcal{R}) \cup \mathcal{R}$ using the termination tool $\mathsf{T_TT_2}$ [15]. For checking the third condition we use matrix interpretations [7].

In the remainder of this section we show how to implement Theorem 3. We start by observing that the condition of Theorem 3 is undecidable even for locally confluent TRSs.

Lemma 20. *The following decision problem is undecidable:*

instance:	*a finite locally confluent linear TRS $\mathcal{R}$,*
question:	*are all critical pairs locally decreasing with respect to the rule-labeling heuristic?*

Proof. We provide a reduction from the problem whether two (arbitrary) ground terms in a linear non-overlapping TRS are joinable. The latter is undecidable as an easy consequence of the encoding of Turing machines as linear non-overlapping TRSs, see e.g. [18]. So let $\mathcal{S}$ be a (finite) linear non-overlapping TRS and let s and t be arbitrary ground terms. We extend $\mathcal{S}$ with fresh constants a, b and the rewrite rules $\{\mathsf{a} \to s, \mathsf{b} \to t, \mathsf{a} \to \mathsf{b}, \mathsf{b} \to \mathsf{a}\}$ to obtain the TRS $\mathcal{R}$. If s and t are joinable (in $\mathcal{S}$) then all its all critical pairs are locally decreasing by ordering all rules in $\mathcal{S}$ below the above four rules If s and t are not joinable, then no order on the rules will make the critical pairs locally decreasing with respect to the rule-labeling heuristic. So confluence of $\mathcal{R}$ can be established by the rule-labeling heuristic if and only if the terms s and t are joinable in $\mathcal{S}$. $\square$

By putting a bound on the number of steps to check joinability we obtain a decidable condition for (extended) local decreasingness:

$$l_2[r_1]_p\mu \; \overbrace{\xrightarrow{\vee}{}^{*}_{\alpha} \cdot \xrightarrow{\veebar}{}^{=}_{\beta} \cdot \xrightarrow{\vee}{}^{*}_{\alpha\beta}}^{\text{at most } k \text{ steps}} \cdot \overbrace{{}_{\alpha\beta}^{*}\xleftarrow{\vee} \cdot {}_{\alpha}^{=}\xleftarrow{\veebar} \cdot {}_{\beta}^{*}\xleftarrow{\vee}}^{\text{at most } k \text{ steps}} \; r_2\mu$$

for each overlap $(l_1 \to r_1, p, l_2 \to r_2)_\mu$ of $\mathcal{R}$ with $\alpha = l_1 \to r_1$ and $\beta = l_2 \to r_2$. Below we reduce this to *precedence constraints* of the form

$$\phi ::= \top \mid \bot \mid \phi \vee \phi \mid \phi \wedge \phi \mid \alpha > \alpha \mid \alpha \geqslant \alpha$$

where α stands for variables corresponding to the rules in $\mathcal{R}$. From the encodings of termination methods for term rewriting, we know that the satisfiability of such precedence constraints is easily determined by SAT or SMT solvers (cf. [4,23]).

Definition 21. *For terms s, t and $k \geqslant 0$, a pair $((\gamma_1, \ldots, \gamma_m), (\delta_1, \ldots, \delta_n))$ is called a k-*join instance* of (s, t) if $m, n \leqslant k$, $\gamma_1, \ldots, \gamma_m, \delta_1, \ldots, \delta_n \in \mathcal{R}$, and*

$$s \to_{\gamma_1} \cdot \cdots \cdot \to_{\gamma_m} \cdot {}_{\delta_n}\!\leftarrow \cdot \cdots \cdot {}_{\delta_1}\!\leftarrow t$$

The embedding order *$\sqsupseteq$ on sequences is defined as $(a_1, \ldots, a_n) \sqsupseteq (a_{i_1}, \ldots, a_{i_m})$ whenever $1 \leqslant i_1 < \cdots < i_m \leqslant n$. The set of all minimal (with respect to $\sqsupseteq \times \sqsupseteq$) k-join instances of (s, t) is denoted by $J_k(s, t)$. We define $\Phi^{\alpha}_{\beta}((\gamma_1, \ldots, \gamma_n))$ as*

$$\bigvee_{i \leqslant n} \Bigl(\bigwedge_{j < i} \alpha > \gamma_j \wedge \Psi_{i,n} \Bigr)$$

with $\Psi_{i,n}$ denoting $\top$ if $i = n$ and

$$\beta \geqslant \gamma_i \wedge \bigwedge_{i < j \leqslant n} (\alpha > \gamma_j \vee \beta > \gamma_j)$$

if $i < n$. Furthermore, $\mathsf{RL}_k(\mathcal{R})$ denotes the conjunction of

$$\bigvee \bigl\{ \Phi^{l_1 \to r_1}_{l_2 \to r_2}(\boldsymbol{\gamma}) \wedge \Phi^{l_2 \to r_2}_{l_1 \to r_1}(\boldsymbol{\delta}) \bigm| (\boldsymbol{\gamma}, \boldsymbol{\delta}) \in J_k(l_2[r_1]_p\mu, r_2\mu) \bigr\}$$

for all overlaps $(l_1 \to r_1, p, l_2 \to r_2)_\mu$ of $\mathcal{R}$.

Table 1. Summary of experimental results

	(a)	(b)	(c)	(d)	(e)	(f)	(g)
YES	20	81	67	49	100	107	135
timeout (60 s)	0	0	0	3	4	3	17

The only non-trivial part of the encoding is the minimality condition in $J_k(s,t)$. The next lemma explains why non-minimal pairs can be excluded from the set and Example 23 shows the benefit of doing so.

Lemma 22. *If $\Phi^{\alpha}_{\beta}(\boldsymbol{\delta})$ is satisfiable and $\boldsymbol{\delta} \sqsupseteq \boldsymbol{\gamma}$ then $\Phi^{\alpha}_{\beta}(\boldsymbol{\gamma})$ is satisfiable.*

Proof. Straightforward. □

We illustrate the encoding on a concrete example.

Example 23. Consider the TRS $\mathcal{R}$ of Example 4. We show how $\mathsf{RL}_4(\mathcal{R})$ is computed. There is a single overlap $(1, 11, 5)_\epsilon$ resulting in the critical pair $s \leftarrow\rtimes\rightarrow t$ with $s = \mathsf{inc}(\mathsf{tl}(0 : \mathsf{inc}(\mathsf{nats})))$ and $t = \mathsf{tl}(\mathsf{inc}(\mathsf{nats}))$. Its 4-join instances are

$$\begin{array}{llll} ((3),(1,4,3)) & ((3,1),(1,4,3,1)) & ((3,1),(1,4,1,3)) & ((3,1),(1,1,4,3)) \\ & ((1,3),(1,4,3,1)) & ((1,3),(1,4,1,3)) & ((1,3),(1,1,4,3)) \end{array}$$

Only the first one belongs to $J_4(s,t)$ and hence $\mathsf{RL}_4(\mathcal{R}) = \Phi^1_5((3)) \wedge \Phi^5_1((1,4,3))$ with $\Phi^1_5((3)) = 5 \geqslant 3 \vee 1 > 3$ and

$$\begin{aligned} \Phi^5_1((1,4,3)) = {} & (1 \geqslant 1 \wedge (1 > 4 \vee 5 > 4) \wedge (1 > 3 \vee 5 > 3)) \\ & \vee (5 > 1 \wedge 1 \geqslant 4 \wedge (1 > 3 \vee 5 > 3)) \vee (5 > 1 \wedge 5 > 4 \wedge 1 \geqslant 3) \\ & \vee (5 > 1 \wedge 5 > 4 \wedge 5 > 3) \end{aligned}$$

This formula is satisfied by taking (e.g.) the order $5 > 3, 4$. Hence, the confluence of $\mathcal{R}$ is concluded by local decreasingness with respect to the rule labeling heuristic using at most 3 steps to close critical pairs.

Theorem 24. *A linear TRS $\mathcal{R}$ is confluent if $\mathsf{RL}_k(\mathcal{R})$ is satisfiable for some $k \geqslant 0$.* □

5 Experimental Results

We tested our methods on a collection of 425 TRSs, consisting of the 103 TRSs in the ACP distribution,[3] the TRSs of Examples 5, 17, and 18, and those TRSs in version 5.0 of the Termination Problems Data Base[4] that are either non-terminating or not known to be terminating. (Systems that have

[3] http://www.nue.riec.tohoku.ac.jp/tools/acp/
[4] http://termination-portal.org/wiki/TPDB

extra variables in right-hand sides of rewrite rules are excluded.) The results are summarized in Table 1. The following techniques are used to produce the columns: (a) Knuth and Bendix' criterion [14]: termination and joinability of all critical pairs, (b) orthogonality, (c) Theorem 24 with $k = 4$, (d) Theorem 6, (e) Theorem 16, (f) the extension of Theorem 16 mentioned at the end of Section 3 in which critical pair steps that generate trivial critical pairs are excluded from $\mathsf{CPS}(\mathcal{R})$, and (g) ACP [2]. To obtain the data in columns (a)–(f) we slightly extended the open source termination tool $\mathsf{T_TT_2}$. For the data in column (c) the SAT solver $\mathsf{MiniSat}$ [6] is used. Since local confluence is undecidable (for non-terminating TRSs), in (c)–(f) it is approximated by $\leftarrow \rtimes \rightarrow \subseteq \bigcup\{\rightarrow^{i} \cdot {}^{j}\!\leftarrow \mid i, j \leqslant 4\}$.

ACP proves that 198 of the 424 TRSs are not confluent. Of the remaining 226 TRSs, local confluence can be shown using at most 4 rewrite steps from both terms in every critical pair for 187 TRSs. Moreover, of these 187 TRSs, 148 are left-linear and 76 are linear. As a final remark, the combination of (c) and (f) proves that 129 TRSs are confluent, (a)+(c)+(f) shows confluence for 134 TRSs, and (c)+(f)+(g) shows confluence for 145 TRSs. These numbers clearly show that both our results have a role to play.

6 Conclusion

In this paper we presented two results based on the decreasing diagrams technique for proving confluence of TRSs. For linear TRSs we showed how the rule-labeling heuristic can be implemented by means of an encoding as a satisfiability problem and we employed the self-labeling heuristic to obtain the result that an arbitrary left-linear locally confluent TRS is confluent if its critical pair steps are relatively terminating with respect to its rewrite rules. We expect that both results will increase the power of ACP [2].

As future work we plan to investigate whether the latter result can be strengthened by decreasing the set $\mathsf{CPS}(\mathcal{R})$ of critical pair steps that need to be relatively terminating with respect to $\mathcal{R}$. We anticipate that some of the many critical pair criteria for confluence that have been proposed in the literature (e.g. [11,21]) can be used for this purpose. The idea here is to exclude the critical pair steps that give rise to critical pairs whose joinability can be shown by the conditions of the considered criterion. Another direction for future work is to determine whether the conversion version of decreasing diagrams [22] can increase the power of automatic confluence tools. Last but not least, in order to certify the output of such tools, we plan to formalize the confluence results presented in this paper in the Isabelle proof assistant.

Acknowledgements. We thank Mizuhito Ogawa, Vincent van Oostrom, and Harald Zankl for commenting on an earlier version of this paper. The detailed comments of the anonymous referees helped to improve the paper.

References

1. Aoto, T., Toyama, Y.: Persistency of confluence. Journal of Universal Computer Science 3(11), 1134–1147 (1997)
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)
3. Bezem, M., Klop, J., van Oostrom, V.: Diagram techniques for confluence. I&C 141(2), 172–204 (1998)
4. Codish, M., Lagoon, V., Stuckey, P.: Solving partial order constraints for LPO termination. In: Pfenning, F. (ed.) RTA 2006. LNCS, vol. 4098, pp. 4–18. Springer, Heidelberg (2006)
5. Dershowitz, N.: Open Closed Open. In: Giesl, J. (ed.) RTA 2005. LNCS, vol. 3467, pp. 276–393. Springer, Heidelberg (2005)
6. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)
7. Endrullis, J., Waldmann, J., Zantema, H.: Matrix interpretations for proving termination of term rewriting. JAR 40(2-3), 195–220 (2008)
8. Geser, A.: Relative Termination. PhD thesis, Universität Passau, Available as technical report 91-03 (1990)
9. Gramlich, B.: Termination and Confluence Properties of Structured Rewrite Systems. PhD thesis, Universität Kaiserslautern (1996)
10. Gramlich, B., Lucas, S.: Generalizing Newman's lemma for left-linear rewrite systems. In: Pfenning, F. (ed.) RTA 2006. LNCS, vol. 4098, pp. 66–80. Springer, Heidelberg (2006)
11. Huet, G.: Confluent reductions: Abstract properties and applications to term rewriting systems. J. ACM 27(4), 797–821 (1980)
12. Jouannaud, J.P., van Oostrom, V.: Diagrammatic confluence and completion. In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikoletseas, S., Thomas, W. (eds.) ICALP 2009. LNCS, vol. 5556, pp. 212–222. Springer, Heidelberg (2009)
13. Klop, J., van Oostrom, V., de Vrijer, R.: A geometric proof of confluence by decreasing diagrams. Journal of Logic and Computation 10(3), 437–460 (2000)
14. Knuth, D., Bendix, P.: Simple word problems in universal algebras. In: Computational Problems in Abstract Algebra, pp. 263–297. Pergamon Press, Oxford (1970)
15. Korp, M., Sternagel, C., Zankl, H., Middeldorp, A.: Tyrolean termination tool 2. In: Treinen, R. (ed.) RTA 2009. LNCS, vol. 5595, pp. 295–304. Springer, Heidelberg (2009)
16. Ohlebusch, E.: Modular Properties of Composable Term Rewriting Systems. PhD thesis, Universität Bielefeld (1994)
17. Rosen, B.: Tree-manipulating systems and Church-Rosser theorems. J. ACM 20(1), 160–187 (1973)
18. Terese: Term Rewriting Systems. Cambridge Tracts in Theoretical Computer Science, vol. 55. Cambridge University Press, Cambridge (2003)
19. Toyama, Y.: Commutativity of term rewriting systems. In: Programming of Future Generation Computers II, pp. 393–407. North-Holland, Amsterdam (1988)
20. van Oostrom, V.: Confluence by decreasing diagrams. TCS 126(2), 259–280 (1994)
21. van Oostrom, V.: Developing developments. TCS 175(1), 159–181 (1997)
22. van Oostrom, V.: Confluence by decreasing diagrams converted. In: Voronkov, A. (ed.) RTA 2008. LNCS, vol. 5117, pp. 306–320. Springer, Heidelberg (2008)
23. Zankl, H., Hirokawa, N., Middeldorp, A.: KBO orientability. JAR 43(2), 173–201 (2009)

Monotonicity Criteria for Polynomial Interpretations over the Naturals*

Friedrich Neurauter, Aart Middeldorp, and Harald Zankl

Institute of Computer Science, University of Innsbruck, Austria

Abstract. Polynomial interpretations are a useful technique for proving termination of term rewrite systems. In an automated setting, termination tools are concerned with parametric polynomials whose coefficients (i.e., the parameters) are initially unknown and have to be instantiated suitably such that the resulting concrete polynomials satisfy certain conditions. We focus on monotonicity and well-definedness, the two main conditions that are independent of the respective term rewrite system considered, and provide constraints on the abstract coefficients for linear, quadratic and cubic parametric polynomials such that monotonicity and well-definedness of the resulting concrete polynomials are guaranteed whenever the constraints are satisfied. Our approach subsumes the absolute positiveness approach, which is currently used in many termination tools. In particular, it allows for negative numbers in certain coefficients. We also give an example of a term rewrite system whose termination proof relies on the use of negative coefficients, thus showing that our approach is more powerful.

1 Introduction

Polynomial interpretations are a simple yet useful technique for proving termination of term rewrite systems (TRSs). They come in various flavors. While originally conceived by Lankford [10] for establishing direct termination proofs, polynomial interpretations are nowadays often used in the context of the dependency pair (DP) framework [1,6,7]. Moreover, the classical approach of Lankford, who only considered polynomial algebras over the natural numbers, was extended by several authors to polynomial algebras over the real numbers [3,11].

This paper is concerned with automatically proving termination of term rewrite systems by means of polynomial interpretations over the natural numbers. In the classical approach, we associate with every n-ary function symbol f a polynomial P_f in n indeterminates with integer coefficients, which induces a mapping or *interpretation* from terms to integer numbers in the obvious way. In order to conclude termination of a given TRS, three conditions have to be satisfied. First, every polynomial must be *well-defined*, i.e., it must induce a well-defined polynomial function $f_{\mathbb{N}}\colon \mathbb{N}^n \to \mathbb{N}$ over the natural numbers. In addition, the interpretation functions $f_{\mathbb{N}}$ are required to be *strictly monotone* in all arguments. Finally,

* This research is supported by FWF (Austrian Science Fund) project P18763.

one has to show *compatibility* of the interpretation with the given TRS. More precisely, for every rewrite rule $l \to r$ the polynomial P_l associated with the left-hand side must be greater than P_r, the corresponding polynomial of the right-hand side, i.e., $P_l > P_r$, for all values (in $\mathbb{N}$) of the indeterminates. These three requirements essentially carry over to the case of using polynomial interpretations as reduction pairs in the DP framework, but in a weakened form. Most notably, the interpretation functions are merely required to be weakly monotone, and for some rules $P_l \geq P_r$ suffices.

In an automated setting, termination tools are concerned with parametric polynomials whose coefficients (i.e., the parameters) are initially unknown and have to be instantiated suitably such that the resulting concrete polynomials satisfy the above conditions. In this paper, we focus on monotonicity (strict and weak) and well-definedness of linear, quadratic and cubic parametric polynomials, two conditions that are independent of the respective TRS considered. The aim is to provide exact constraints in terms of the abstract coefficients of a parametric polynomial such that monotonicity and well-definedness of the resulting concrete polynomial are guaranteed for every instantiation of the coefficients that satisfies the constraints. For example, given the parametric polynomial $ax^2 + bx + c$, we identify constraints on the parameters a, b and c such that the associated polynomial function is both well-defined and (strictly) monotone. Our approach subsumes the absolute positiveness approach [9], which is currently used in many termination tools. In contrast to the latter, negative numbers in certain coefficients can be handled without further ado. Previous work allowing negative coefficients ensures well-definedness and (weak) monotonicity by extending polynomials with “max” [8,5]. However, all our interpretation functions are polynomials and our results do also apply to strict monotonicity. Hence in the sequel we do not consider these approaches.

The remainder of this paper is organized as follows. In Section 2, we introduce some preliminary definitions and terminology concerning polynomials and polynomial interpretations. The follow-up section is the main section of this paper where we present our results concerning monotonicity of linear, quadratic and cubic parametric polynomials. In Section 4, we give a constructive proof showing that our approach is more powerful than the absolute positiveness approach. Finally, Section 5 presents some experimental results.

2 Preliminaries

For any ring R, we denote the associated polynomial ring in n indeterminates $x_1, \ldots, x_n$ by $R[x_1, \ldots, x_n]$. For example, the polynomial $2x^2 - x + 1$ is an element of $\mathbb{Z}[x]$, the ring of all univariate polynomials with coefficients in $\mathbb{Z}$. Let $P := \sum_{k=0}^{n} a_k x^k$ be an element of the polynomial ring $R[x]$. For the largest k where $a_k \neq 0$, we call $a_k x^k$ the *leading term* of P, a_k its *leading coefficient* and k its *degree*. Moreover, we call a_0 the *constant coefficient* or *constant term* of P.

A *quadratic equation* is an equation of the form $ax^2 + bx + c = 0$, where x is an indeterminate, and a, b and c represent constants, with $a \neq 0$. The solutions of a quadratic equation, called *roots*, are given by the *quadratic formula*:

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

In this formula, the expression $d := b^2 - 4ac$ underneath the square root sign is of central importance because it determines the nature of the roots; it is also called the *discriminant* of a quadratic equation. If all coefficients are real numbers, one of the following three cases applies:

1. If d is positive, there are two distinct roots, both of which are real numbers.
2. If d is zero, there is exactly one real root, called a *double root*.
3. If d is negative, there are no real roots. Both roots are complex numbers.

The key concept for using polynomial interpretations to establish (direct) termination of term rewrite systems is the notion of well-founded monotone algebras (WFMAs) since they induce reduction orders on terms. Let $\mathcal{F}$ be a signature. A *well-founded monotone $\mathcal{F}$-algebra* $(\mathcal{A}, >)$ is a non-empty algebra $\mathcal{A} = (A, \{f_A\}_{f \in \mathcal{F}})$ together with a well-founded order $>$ on the carrier A of $\mathcal{A}$ such that every algebra operation f_A is strictly monotone in all arguments, i.e., if $f \in \mathcal{F}$ has arity $n \geq 1$ then $f_A(a_1, \ldots, a_i, \ldots, a_n) > f_A(a_1, \ldots, b, \ldots, a_n)$ for all $a_1, \ldots, a_n, b \in A$ and $i \in \{1, \ldots, n\}$ with $a_i > b$.

Concerning the use of polynomial interpretations in the context of the DP framework, the notion of a *well-founded weakly monotone algebra* (WFWMA) is sufficient to obtain a reduction pair. A WFWMA is just like a WFMA, with the exception that weak rather than strict monotonicity is required; i.e., $f_A(a_1, \ldots, a_i, \ldots, a_n) \geq f_A(a_1, \ldots, b, \ldots, a_n)$ whenever $a_i \geq b$. Here $\geq$ is the reflexive closure of $>$.

Given a monotone algebra $(\mathcal{A}, >)$, we define the relations $\geq_{\mathcal{A}}$ and $>_{\mathcal{A}}$ on terms as follows: $s \geq_{\mathcal{A}} t$ if $[\alpha]_{\mathcal{A}}(s) \geq [\alpha]_{\mathcal{A}}(t)$ and $s >_{\mathcal{A}} t$ if $[\alpha]_{\mathcal{A}}(s) > [\alpha]_{\mathcal{A}}(t)$, for all assignments α of elements of A to the variables in s and t ($[\alpha]_{\mathcal{A}}(\cdot)$ denotes the usual evaluation function associated with $\mathcal{A}$). Now if $(\mathcal{A}, >)$ is a WFMA, then $>_{\mathcal{A}}$ is a reduction order that can be used to prove termination of term rewrite systems by showing that $>_{\mathcal{A}}$ orients the rewrite rules from left to right. If, on the other hand, $(\mathcal{A}, >)$ is a WFWMA, then $(\geq_{\mathcal{A}}, >_{\mathcal{A}})$ is a reduction pair that can be used to establish termination in the context of the DP framework.

3 Parametric Polynomials

Polynomial interpretations over the natural numbers are based on the well-founded algebra $(\mathcal{N}, >)$, where $>$ is the standard order on the natural numbers $\mathbb{N}$ and $\mathcal{N} = (\mathbb{N}, \{f_{\mathbb{N}}\}_{f \in \mathcal{F}})$ such that every algebra operation $f_{\mathbb{N}}$ is a polynomial with integer coefficients. Depending on whether all algebra operations are strictly or weakly monotone, $(\mathcal{N}, >)$ is either a WFMA or a WFWMA. To be precise, every n-ary function symbol $f \in \mathcal{F}$ is associated with a polynomial with integer coefficients such that the corresponding algebra operation $f_{\mathbb{N}} \colon \mathbb{N}^n \to \mathbb{N}$ is a well-defined polynomial function which is strictly or weakly monotone in all arguments. Note, however, that this does not imply that all coefficients of the polynomials must be natural numbers.

Example 1. The univariate integer polynomial $2x^2 - x + 1 \in \mathbb{Z}[x]$ gives rise to the polynomial function $f_\mathbb{N} \colon \mathbb{N} \to \mathbb{N}$, $x \mapsto 2x^2 - x + 1$, which is obviously well-defined over $\mathbb{N}$. Moreover, it is also strictly monotone with respect to $\mathbb{N}$. Note, however, that monotonicity does not hold if we view $2x^2 - x + 1$ as a function over the (non-negative) real numbers.

Summing up, an n-ary polynomial function $f_\mathbb{N}$ used in a polynomial interpretation is an element of the polynomial ring $\mathbb{Z}[x_1, \dots, x_n]$ and must satisfy:

1. *well-definedness*: $f_\mathbb{N}(x_1, \dots, x_n) \geq 0$ for all $x_1, \dots, x_n \in \mathbb{N}$
2. *strict (weak) monotonicity*: $f_\mathbb{N}(x_1, \dots, x_i, \dots, x_n) \underset{(=)}{>} f_\mathbb{N}(x_1, \dots, y, \dots, x_n)$ for all $i \in \{1, \dots, n\}$ and $x_1, \dots, x_n, y \in \mathbb{N}$ with $x_i > y$.

Alas, both of these properties are instances of the undecidable problem of checking *positiveness of polynomials*[1] in the polynomial ring $\mathbb{Z}[x_1, \dots, x_n]$ (undecidable by reduction from Hilbert's 10-th problem).

Termination tools face the following problem. They deal with parametric polynomials, i.e., polynomials whose coefficients are unknowns (e.g., $ax^2 + bx + c$), and the task is to find suitable integer numbers for the unknown coefficients such that the resulting polynomials induce algebra operations that satisfy both of the above properties. The solution that is used in practice is to restrict the search space for the unknown coefficients to the non-negative integers (absolute positiveness approach [9,2]) because then well-definedness and weak monotonicity are obtained for free. To obtain strict monotonicity in the i-th argument of a polynomial function $f_\mathbb{N}(\dots, x_i, \dots)$, at least one of the terms $(c_k x_i^k)_{k>0}$ must have a positive coefficient $c_k > 0$.

Obviously, this approach is easy to implement and works quite well in practice. However, it is not optimal in the sense that it excludes certain polynomials, like $2x^2 - x + 1$, which might be useful to prove termination of certain TRSs. So how can we do better? To this end, let us observe that in general termination tools only use restricted forms of polynomials to interpret function symbols. There are restrictions concerning the degree of the polynomials (linear, quadratic, etc.) and sometimes also restrictions that disallow certain kinds of monomials. Now the idea is as follows. Despite the fact that well-definedness and monotonicity are undecidable in general, it might be the case that they are decidable for the restricted forms of polynomials used in practice. And indeed, that is the case, as we shall see shortly.

Remark 2. Checking compatibility of a rewrite rule $l \to r$ with a polynomial interpretation means showing that the rule gives rise to a (weak) decrease; i.e., $P_l - P_r > 0$ ($P_l - P_r \geq 0$). In $\mathbb{N}$, both cases reduce to checking non-negativity of polynomials because $x > y$ if and only if $x \geq y + 1$. Since well-definedness of a polynomial as defined above is equivalent to non-negativity of a polynomial in $\mathbb{N}$, any method that ensures non-negativity of parametric polynomials can also be used for checking compatibility. However, we remark that the method presented in this paper is not ideally suited for this purpose as it also enforces strict monotonicity, which is irrelevant for compatibility.

[1] Given $P \in \mathbb{Z}[x_1, \dots, x_n]$, decide $P(x_1, \dots, x_n) > 0$ for all $x_1, \dots, x_n \in \mathbb{N}$.

In the sequel, we analyze parametric polynomials whose only restriction is a bound on the degree. We will first treat linear parametric polynomials. While this does not yield new results or insights, it is instructive to demonstrate our approach in a simple setting. This is followed by an analysis of quadratic and finally also cubic parametric polynomials, both of which yield new results. The following lemmas will be helpful in this analysis. The first one gives a more succinct characterization of monotonicity, whereas the second one relates monotonicity and well-definedness.

Lemma 3. *A (not necessarily polynomial) function $f_{\mathbb{N}}\colon \mathbb{N}^n \to \mathbb{Z}$ is strictly (weakly) monotone in all arguments if and only if*

$$f_{\mathbb{N}}(x_1, \ldots, x_i + 1, \ldots, x_n) \underset{(=)}{>} f_{\mathbb{N}}(x_1, \ldots, x_i, \ldots, x_n)$$

for all $x_1, \ldots, x_n \in \mathbb{N}$ and all $i \in \{1, \ldots, n\}$.

Lemma 4. *Let $f\colon \mathbb{Z}^n \to \mathbb{Z}$ be the polynomial function associated with a polynomial in $\mathbb{Z}[x_1, \ldots, x_n]$, and let $f_{\mathbb{N}}\colon \mathbb{N}^n \to \mathbb{Z}$ denote its restriction to $\mathbb{N}$. Then $f_{\mathbb{N}}$ is strictly (weakly) monotone and well-defined if and only if it is strictly (weakly) monotone and $f_{\mathbb{N}}(0, \ldots, 0) \geq 0$.*

In these lemmata, as well as in the remainder of the paper, monotonicity and well-definedness refer to the two properties mentioned at the beginning of this section. In particular, monotonicity is meant with respect to all arguments.

3.1 Linear Parametric Polynomials

In this section we consider the generic linear parametric polynomial function $f_{\mathbb{N}}(x_1, \ldots, x_n) = a_n x_n + a_{n-1} x_{n-1} + \cdots + a_1 x_1 + a_0$, and derive constraints on the coefficients a_i that guarantee monotonicity and well-definedness.

Theorem 5. *The function $f_{\mathbb{N}}(x_1, \ldots, x_n) = a_n x_n + \ldots + a_1 x_1 + a_0$ ($a_i \in \mathbb{Z}$, $0 \leq i \leq n$) is strictly (weakly) monotone and well-defined if and only if $a_0 \geq 0$ and $a_i > 0$ ($a_i \geq 0$) for all $i \in \{1, \ldots, n\}$.*

Proof. Easy consequence of Lemmata 4 and 3. □

Remark 6. Note that all coefficients must be non-negative and that the constraints on the coefficients are exactly the ones one would obtain by the absolute positiveness approach. Furthermore, these constraints are optimal in the sense that they are both necessary and sufficient for monotonicity and well-definedness.

3.2 Quadratic Parametric Polynomials

Next we apply the approach illustrated by Theorem 5 to the generic quadratic parametric polynomial function

$$f_{\mathbb{N}}(x_1, \ldots, x_n) = a_0 + \sum_{j=1}^{n} a_j x_j + \sum_{1 \leq j \leq k \leq n} a_{jk} x_j x_k \ \in \mathbb{Z}[x_1, \ldots, x_n] \qquad (1)$$

Theorem 7. *The function $f_\mathbb{N}$ is strictly (weakly) monotone and well-defined if and only if $a_0 \geq 0$, $a_{jk} \geq 0$ and $a_j > -a_{jj}$ ($a_j \geq -a_{jj}$) for all $1 \leq j \leq k \leq n$.*

Proof. By Lemmata 3 and 4, this theorem holds if and only if $f_\mathbb{N}(0,\ldots,0) \geq 0$, and $f_\mathbb{N}(x_1,\ldots,x_i+1,\ldots,x_n) \underset{(\geq)}{>} f_\mathbb{N}(x_1,\ldots,x_i,\ldots,x_n)$ for all $x_1,\ldots,x_n \in \mathbb{N}$ and all $i \in \{1,\ldots,n\}$. Clearly, $f_\mathbb{N}(0,\ldots,0) \geq 0$ holds if and only if $a_0 \geq 0$, and the monotonicity condition $f_\mathbb{N}(x_1,\ldots,x_i+1,\ldots,x_n) > f_\mathbb{N}(x_1,\ldots,x_i,\ldots,x_n)$ yields

$$a_i(x_i+1) + a_{ii}(x_i+1)^2 + \sum_{i<k\leq n} a_{ik}(x_i+1)x_k + \sum_{1\leq j<i} a_{ji}x_j(x_i+1)$$
$$> a_ix_i + a_{ii}x_i^2 + \sum_{i<k\leq n} a_{ik}x_ix_k + \sum_{1\leq j<i} a_{ji}x_jx_i$$

which simplifies to

$$a_i + a_{ii} + 2a_{ii}x_i + \sum_{i<k\leq n} a_{ik}x_k + \sum_{1\leq j<i} a_{ji}x_j > 0$$

This is a linear inequality that holds for all $x_1,\ldots,x_n \in \mathbb{N}$ if and only if $a_i + a_{ii} > 0$ and all other coefficients are non-negative. Taking the quantification over i into account, this proves the claim for strict monotonicity; the result for weak monotonicity follows by replacing $>$ with $\geq$ in the above calculation. □

Corollary 8. *The function $f_\mathbb{N}(x) = ax^2 + bx + c$ is strictly (weakly) monotone and well-defined if and only if $a \geq 0$, $c \geq 0$ and $b > -a$ ($b \geq -a$).*

Hence, in a quadratic parametric polynomial all coefficients must be non-negative except the coefficients of the linear monomials. They can be negative; for example, the polynomial $2x^2 - x + 1$ satisfies the constraints of Corollary 8; hence, it is both well-defined and strictly monotone.

Remark 9. Not only does our approach improve upon absolute positiveness for quadratic parametric polynomials, but the constraints derived from it are even optimal, i.e., necessary and sufficient for monotonicity and well-definedness.

Example 10. The polynomial function $f_\mathbb{N}(x_1,x_2) = 2x_1^2+3x_2^2+x_1x_2-x_1-2x_2+1$ is both well-defined and strictly monotone according to Theorem 7. Yet we can also infer this result in a more modular and probably more intuitive way by using Corollary 8. To this end, let $f_\mathbb{N}(x_1,x_2) = g_1(x_1) + g_2(x_2) + x_1x_2 + 1$, where $g_1(x_1) = 2x_1^2 - x_1$ and $g_2(x_2) = 3x_2^2 - 2x_2$. Clearly, by Corollary 8, $g_1(x_1)$ and $g_2(x_2)$ are both well-defined and strictly monotone. The same also holds for their sum, $g_1(x_1) + g_2(x_2)$, because $g_1(x_1)$ and $g_2(x_2)$ do not share variables. Finally, we may conclude that $f_\mathbb{N}$ is then also well-defined and strictly monotone by observing that the addition of monomials with non-negative coefficients (in this case: x_1x_2 and 1) is not harmful.

Another thing that is noteworthy about the previous theorem is that it subsumes the result of Theorem 5. That is to say, if we set the coefficients a_{jk} of all quadratic monomials in (1) to zero, thereby obtaining the linear parametric polynomial function $f'_{\mathbb{N}}(x_1, \ldots, x_n) = a_0 + \sum_{j=1}^{n} a_j x_j$, then the constraints generated by Theorem 7 are in fact the ones Theorem 5 would produce when applied to $f'_{\mathbb{N}}$. In theory, this means that if we want to prove termination of some TRS, then we do not have to specify a priori whether to interpret a function symbol by a linear or a quadratic parametric polynomial function; we can always go for quadratic interpretations, and it is solely determined by the constraint solving process (i.e., the process that assigns suitable integers to the abstract coefficients such that all constraints are satisfied) whether the resulting concrete polynomial function is linear or quadratic. In practice, however, this approach has an important drawback; that is, it increases both the number of abstract coefficients and the number of constraints involving these coefficients, which is detrimental to the performance of the constraint solving process.

3.3 Cubic Parametric Polynomials

Next we apply our approach to cubic parametric polynomials. First, we consider the univariate polynomial function $f_{\mathbb{N}}(x) = ax^3 + bx^2 + cx + d \in \mathbb{Z}[x]$, for which the monotonicity condition $f_{\mathbb{N}}(x+1) \underset{(-)}{>} f_{\mathbb{N}}(x)$ for all $x \in \mathbb{N}$ simplifies to

$$\forall x \in \mathbb{N} \quad 3ax^2 + (3a + 2b)x + (a + b + c) \underset{(-)}{>} 0 \tag{2}$$

In the interesting case, where $a \neq 0$, the polynomial $P := 3ax^2 + (3a + 2b)x + (a + b + c)$ is a quadratic polynomial in x whose geometric representation is a parabola in two-dimensional space, which has a global minimum at $x_{min} := -(3a + 2b)/(6a)$. Since a is involved in the leading coefficient of P, a must necessarily be positive in order for (2) to hold. Next we focus on strict monotonicity, that is, the solution of the inequality

$$\forall x \in \mathbb{N} \quad 3ax^2 + (3a + 2b)x + (a + b + c) > 0 \tag{3}$$

Now this inequality holds if and only if either $x_{min} < 0$ and $P(0) > 0$ or $x_{min} \geq 0$ and both $P(\lfloor x_{min} \rfloor) > 0$ and $P(\lceil x_{min} \rceil) > 0$. However, these constraints use the floor and ceiling functions, but we would rather have a set of polynomial constraints in a, b and c (which can easily be encoded in SAT or SMT). It is possible, however, to eliminate the floor and ceiling functions from the above constraints, but only at the expense of introducing new variables; e.g., $\lfloor x_{min} \rfloor = n$ for some $n \in \mathbb{Z}$ if and only if $n \leq x_{min} < n + 1$. Thus one obtains a set of polynomial constraints in a, b, c and the additional variables. But one can also avoid the introduction of new variables with the following approach. To this end, we examine the roots of P and distinguish two possible cases:

$\boxed{\textbf{Case 1}}$ P has no roots in $\mathbb{R}$ (both roots are complex numbers),

$\boxed{\textbf{Case 2}}$ both roots of P are real numbers.

In the first case, (3) trivially holds. Moreover, this case is completely characterized by the discriminant of P being negative, i.e., $4b^2 - 3a^2 - 12ac < 0$. In the other case, when both roots r_1 and r_2 are real numbers, the discriminant is non-negative and (3) holds if and only if the closed interval $[r_1, r_2]$ does not contain a natural number, i.e., $[r_1, r_2] \cap \mathbb{N} = \varnothing$. While this condition can be fully characterized with the help of the floor and/or ceiling functions, we can also obtain a polynomial characterization as follows. We require the larger of the two roots, that is, r_2, to be negative because then (3) is guaranteed to hold. This observation leads to the constraints

$$4b^2 - 3a^2 - 12ac \geq 0 \quad \text{and} \quad r_2 = \frac{-(3a+2b) + \sqrt{4b^2 - 3a^2 - 12ac}}{6a} < 0$$

which can be simplified to

$$4b^2 - 3a^2 - 12ac \geq 0 \tag{4}$$

$$\sqrt{4b^2 - 3a^2 - 12ac} < 3a + 2b \tag{5}$$

Due to (4), (5) holds if and only if $4b^2 - 3a^2 - 12ac < (3a+2b)^2$ and $3a + 2b \geq 0$, which simplifies to $a + b + c > 0$ and $3a + 2b \geq 0$. Putting everything together, we obtain the following theorem.

Theorem 11. *The function $f_{\mathbb{N}}(x) = ax^3 + bx^2 + cx + d$ is strictly monotone and well-defined if $a \geq 0$, $d \geq 0$ and either $4b^2 - 3a^2 - 12ac < 0$ or $4b^2 - 3a^2 - 12ac \geq 0$, $a + b + c > 0$ and $3a + 2b \geq 0$.*

Note that these constraints are only sufficient for monotonicity and well-definedness, they are not necessary. However, they are very close to necessary constraints, as will be explained below.

Remark 12. Weak monotonicity of $ax^3 + bx^2 + cx + d$ is obtained by similar reasoning. The only difference is that in case 2 we differentiate between distinct real roots $r_1 \neq r_2$ and a double root $r_1 = r_2$. In the latter case, which is characterized algebraically by the discriminant of P being zero, (2) holds unconditionally, whereas in the former case, where the discriminant of P is positive, it suffices to require the larger of the two roots to be negative or zero.

Theorem 13. *The function $f_{\mathbb{N}}(x) = ax^3 + bx^2 + cx + d$ is weakly monotone and well-defined if $a \geq 0$, $d \geq 0$ and either $4b^2 - 3a^2 - 12ac \leq 0$ or $4b^2 - 3a^2 - 12ac > 0$, $a + b + c \geq 0$ and $3a + 2b \geq 0$.*

In case $a = 0$, i.e., $f_{\mathbb{N}}(x) = bx^2 + cx + d$, Theorem 11 yields exactly the same constraints as Corollary 8, that is, necessary and sufficient constraints. One possible interpretation of this fact is that the simplification we made on our way to Theorem 11 did not cast away anything essential. Indeed, that is the case. To this end, we observe that the only case where (3) holds that is not covered by the constraints of Theorem 11 is when both roots r_1 and r_2 are positive and $[r_1, r_2] \cap \mathbb{N} = \varnothing$; e.g., the polynomial $2x^3 - 6x^2 + 5x$ is both strictly monotone and well-defined, but does not satisfy the constraints of Theorem 11. However, it turns out that this case is very rare; for example, empirical investigations reveal that in the set of polynomials $\{3ax^2 + (3a + 2b)x + (a + b + c) \mid 1 \leq a \leq 7, -15 \leq b, c \leq 15\ (a, b, c \in \mathbb{Z})\}$

3937 out of a total of 6727 polynomials satisfy (3), but only 25 of them are of this special kind. In other words, the constraints of Theorem 11 comprise 3912 out of 3937, hence almost all, polynomials; and this is way more than the 1792 ($= 7 \times 16 \times 16$) polynomials that the absolute positiveness approach, where a, b and c are restricted to the non-negative integers, can handle. The following table summarizes all our experiments with varying ranges for a, b and c:

a	b, c	Theorem 11
$[1,7]$	$[-15,15]$	3912 of 3937
$[1,7]$	$[-31,31]$	14055 of 14133
$[1,15]$	$[-31,31]$	34718 of 34980

By design, our approach covers two out of the three possible scenarios mentioned above. But which of these scenarios can the absolute positiveness approach deal with? Just like our method, it fails on all instances of the scenario where the polynomial $P := 3ax^2 + (3a+2b)x + (a+b+c)$ has two positive roots r_1 and r_2, which gives rise to the factorization $P = k(x-r_1)(x-r_2)$, $k > 0$. This expression is equivalent to $kx^2 - k(r_1+r_2)x + kr_1r_2$, the linear coefficient $-k(r_1+r_2)$ of which should be equal to $3a + 2b$. Now this gives rise to a contradiction because a and b are restricted to the non-negative integers whereas $-k(r_1+r_2)$ is a negative number. Concerning the two remaining scenarios, the absolute positiveness approach can handle only some instances of the respective scenarios while failing at others. We present one failing example for either scenario:

– If $a = 1$, $b = -1$ and $c = 1$, then $P = 3x^2 + x + 1$, which has no real roots. Clearly, P is positive for all $x \in \mathbb{N}$; in fact this is even true for all $x \in \mathbb{R}$. However, the absolute positiveness approach fails because b is negative.
– If $a = 3$, $b = -1$ and $c = -1$, then $P = 9x^2 + 7x + 1$, both roots of which are negative real numbers. Clearly, P is positive for all $x \in \mathbb{N}$, but the absolute positiveness approach fails because b and c are negative.

Generalization to Multivariate Cubic Parametric Polynomials

In this subsection, we elaborate on the question how to generalize the result of Theorem 11 to the multivariate case. In general, this is always possible by a very simple approach that we already introduced in Example 10. To this end, let $f_{\mathbb{N}}(x_1, \dots, x_n)$ denote the n-variate generic cubic parametric polynomial function, and let us note that we can write it as

$$f_{\mathbb{N}}(x_1, \dots, x_n) = \sum_{j=1}^{n} g_j(x_j) + r(x_1, \dots, x_n) \tag{6}$$

where $g_j(x_j)$ denotes the univariate generic cubic parametric polynomial function in x_j without constant term and $r(x_1, \dots, x_n)$ contains all the remaining monomials. Now, let us assume that all the $g_j(x_j)$ are both strictly monotone and well-defined. Then the same also holds for their sum, $\sum_{j=1}^{n} g_j(x_j)$, because they do not share variables. But when is this also true of $f_{\mathbb{N}}$? By Lemma 3, $f_{\mathbb{N}}$ is strictly monotone in its i-th argument if and only if $f_{\mathbb{N}}(x_1, \ldots, x_i + 1, \ldots, x_n) - f_{\mathbb{N}}(x_1, \ldots, x_n) > 0$ for all natural numbers $x_1, \ldots, x_n$. With the help of (6), this simplifies to: $\forall x_1, \ldots, x_n \in \mathbb{N}$

$$g_i(x_i + 1) - g_i(x_i) + r(x_1, \ldots, x_i + 1, \ldots, x_n) - r(x_1, \ldots, x_i, \ldots, x_n) > 0 \quad (7)$$

By assumption, $g_i(x_i + 1) - g_i(x_i) > 0$ for all $x_i \in \mathbb{N}$, such that (7) is guaranteed to hold if the second summand, $r(x_1, \ldots, x_i+1, \ldots, x_n) - r(x_1, \ldots, x_i, \ldots, x_n)$, is non-negative for all $x_1, \ldots, x_n \in \mathbb{N}$, that is, if $r(x_1, \ldots, x_n)$ is weakly monotone in all arguments. In other words, strict monotonicity of the functions $(g_j(x_j))_{1 \leq j \leq n}$ implies strict monotonicity of $f_{\mathbb{N}}$, provided that $r(x_1, \ldots, x_n)$ is weakly monotone in all arguments. Moreover, if all the functions $(g_j(x_j))_{1 \leq j \leq n}$ are strictly monotone and well-defined, and if $r(x_1, \ldots, x_n)$ is weakly monotone and well-defined, then $f_{\mathbb{N}}$ is strictly monotone and well-defined; and note that we can easily make $r(x_1, \ldots, x_n)$ weakly monotone and well-defined by restricting all its coefficients to be non-negative. Hence, the n-variate generic cubic parametric polynomial function $f_{\mathbb{N}}(x_1, \ldots, x_n)$ is strictly monotone and well-defined if

– all the $g_j(x_j)$ satisfy the constraints of Theorem 11, and
– all coefficients of $r(x_1, \ldots, x_n)$ are non-negative.

Example 14. Consider the bivariate generic cubic parametric polynomial function $f_{\mathbb{N}}(x_1, x_2) = ax_1^3 + bx_1^2x_2 + cx_1x_2^2 + dx_2^3 + ex_1^2 + fx_1x_2 + gx_2^2 + hx_1 + ix_2 + j = g_1(x_1) + g_2(x_2) + r(x_1, x_2)$, where $g_1(x_1) = ax_1^3 + ex_1^2 + hx_1$, $g_2(x_2) = dx_2^3 + gx_2^2 + ix_2$ and $r(x_1, x_2) = bx_1^2x_2 + cx_1x_2^2 + fx_1x_2 + j$. This function is both strictly monotone and well-defined if $ax_1^3 + ex_1^2 + hx_1$ and $dx_2^3 + gx_2^2 + ix_2$ satisfy the constraints of Theorem 11, and the coefficients of $r(x_1, \ldots, x_n)$ are non-negative, i.e., $b, c, f, j \geq 0$.

4 Negative Coefficients in Polynomial Interpretations

In the previous section, we have seen that in principle we may use polynomial interpretations with (some) negative coefficients for proving termination of TRSs. Now the obvious question is the following: Does there exist a TRS that can be proved terminating by a polynomial interpretation with negative coefficients according to Theorems 7 and 11, but cannot be proved terminating by a polynomial interpretation where the coefficients of all polynomials are non-negative?

To elaborate on this question, let us consider the following scenario. Assume we have a TRS whose signature contains (amongst others) the *successor* symbol s, the constant $\mathsf{0}$ and another unary symbol f, and assume that the interpretations associated with the former two are the natural interpretations $\mathsf{s}_{\mathbb{N}}(x) = x+1$ and $\mathsf{0}_{\mathbb{N}} = 0$, whereas f is supposed to be interpreted by $\mathsf{f}_{\mathbb{N}}(x) = ax^2 + bx + c$. Now the idea is to add rules to the TRS which enforce $\mathsf{f}_{\mathbb{N}}(x) = 2x^2 - x + 1$. This can be achieved as follows.

First, note that by polynomial interpolation the coefficients a, b and c of the polynomial function $\mathsf{f}_{\mathbb{N}}(x) = ax^2 + bx + c$ are uniquely determined by the image of $\mathsf{f}_{\mathbb{N}}$ at three pairwise different locations; for example, the constraints $\mathsf{f}_{\mathbb{N}}(0) = 1$,

$\mathsf{f}_{\mathbb{N}}(1) = 2$ and $\mathsf{f}_{\mathbb{N}}(2) = 7$ enforce $\mathsf{f}_{\mathbb{N}}(x) = 2x^2 - x + 1$, as desired. Next we encode these three constraints in terms of the TRS $\mathcal{R}_1$:

$$\begin{array}{ccc} \mathsf{s}^2(\mathsf{0}) \to \mathsf{f}(\mathsf{0}) & \mathsf{s}^3(\mathsf{0}) \to \mathsf{f}(\mathsf{s}(\mathsf{0})) & \mathsf{s}^8(\mathsf{0}) \to \mathsf{f}(\mathsf{s}^2(\mathsf{0})) \\ \mathsf{f}(\mathsf{0}) \to \mathsf{0} & \mathsf{f}(\mathsf{s}(\mathsf{0})) \to \mathsf{s}(\mathsf{0}) & \mathsf{f}(\mathsf{s}^2(\mathsf{0})) \to \mathsf{s}^6(\mathsf{0}) \end{array}$$

Every constraint gives rise to two rewrite rules; e.g., the constraint $\mathsf{f}_{\mathbb{N}}(0) = 1$ is expressed by $\mathsf{f}(\mathsf{0}) \to \mathsf{0}$ and $\mathsf{s}^2(\mathsf{0}) \to \mathsf{f}(\mathsf{0})$. The former encodes $\mathsf{f}_{\mathbb{N}}(0) > 0$, whereas the latter encodes $\mathsf{f}_{\mathbb{N}}(0) < 2$. So these rewrite rules are polynomially terminating by construction, with $\mathsf{f}_{\mathbb{N}}(x) = 2x^2 - x + 1$.

Moreover, we can use $\mathcal{R}_1$ to prove a more general statement that does away with one of the above assumptions. That is to say that any feasible interpretation $\mathsf{f}_{\mathbb{N}}$ must necessarily contain at least one monomial with a negative coefficient. To this end, let us observe that no linear interpretation for f is feasible because the set of points $\{(i, \mathsf{f}_{\mathbb{N}}(i))\}_{i \in \{0,1,2\}}$ is not collinear. The case when $\mathsf{f}_{\mathbb{N}}$ is quadratic was dealt with above. So let us consider interpretations of degree at least three. Then the leading term of $\mathsf{f}_{\mathbb{N}}$ has the shape ax^k, where $a \geq 1$ and $k \geq 3$. Since $\mathsf{f}_{\mathbb{N}}(2) = 7$ must be satisfied, the claim follows immediately because for $x = 2$ the leading term alone contributes a value of at least 8.

Finally, a thorough inspection of the constraints imposed by $\mathcal{R}_1$ reveals that we can also relax the restrictions concerning the interpretations of s and $\mathsf{0}$.

Lemma 15. *In any polynomial interpretation compatible with* $\mathcal{R}_1$ *that satisfies* $\mathsf{s}_{\mathbb{N}}(x) = x + d$ *for some* $d \in \mathbb{N}$*,* $\mathsf{f}_{\mathbb{N}}$ *must contain at least one monomial with a negative coefficient. In particular,* $\mathsf{f}_{\mathbb{N}}$ *is not linear.*

Proof. Without loss of generality, let f be interpreted by $\mathsf{f}_{\mathbb{N}}(x) = \sum_{i=0}^{n} a_i x^i$ ($a_n \geq 1$) and $\mathsf{0}$ by some natural number z. Then the compatibility requirement with respect to $\mathcal{R}_1$ gives rise to the following constraints:

$$\begin{aligned} z &< \mathsf{f}_{\mathbb{N}}(z) < z + 2d \\ z + d &< \mathsf{f}_{\mathbb{N}}(z + d) < z + 3d \\ z + 6d &< \mathsf{f}_{\mathbb{N}}(z + 2d) < z + 8d \end{aligned}$$

Hence, d must be a positive integer, i.e., $d \geq 1$. Moreover, no linear interpretation $\mathsf{f}_{\mathbb{N}}(x) = a_1 x + a_0$ satisfies these constraints. To this end, observe that by the first four constraints $a_1 = \frac{\mathsf{f}_{\mathbb{N}}(z+d) - \mathsf{f}_{\mathbb{N}}(z)}{d} < 3$, whereas by the last four constraints $a_1 = \frac{\mathsf{f}_{\mathbb{N}}(z+2d) - \mathsf{f}_{\mathbb{N}}(z+d)}{d} > 3$, which contradicts the former. In other words, the set of points $\{(z + id, \mathsf{f}_{\mathbb{N}}(z + id))\}_{i \in \{0,1,2\}}$ is not collinear. Next we focus on $\mathsf{f}_{\mathbb{N}}(z + 2d) < z + 8d$. Clearly, if the value of the leading term $a_n x^n$ at $x = z + 2d$ is greater than or equal to $z + 8d$, then $\mathsf{f}_{\mathbb{N}}$ must contain at least one monomial with a negative coefficient in order to satisfy $\mathsf{f}_{\mathbb{N}}(z + 2d) < z + 8d$. So, when is $a_n(z + 2d)^n \geq z + 8d$? Considering the worst case, i.e. $a_n = 1$, let us investigate for which integers $n \geq 2$, $z \geq 0$ and $d \geq 1$ the inequality $(z + 2d)^n \geq z + 8d$ holds. If $n \geq 3$, then it holds for all $z \geq 0$ and $d \geq 1$ by the following reasoning $(z + 2d)^n \geq z^n + (2d)^n \geq z + 8d$. For $n = 2$, $(z + 2d)^2 \geq z + 8d$ is equivalent to $z^2 + (4d - 1)z + 4d(d - 2) \geq 0$ which holds for all $z \geq 0$ and $d \geq 1$ except $z = 0$

and $d = 1$. The latter case corresponds to using the natural interpretations for the symbols s and $\mathsf{0}$, namely, $\mathsf{s}_{\mathbb{N}}(x) = x + 1$ and $\mathsf{0}_{\mathbb{N}} = 0$. But then the six rewrite rules require the constraints $\mathsf{f}_{\mathbb{N}}(0) = 1, \mathsf{f}_{\mathbb{N}}(1) = 2$ and $\mathsf{f}_{\mathbb{N}}(2) = 7$, which uniquely determine the coefficients of $\mathsf{f}_{\mathbb{N}}(x) = a_2x^2 + a_1x + a_0$ as $a_2 = 2$, $a_1 = -1$ and $a_0 = 1$ by polynomial interpolation. Hence, $\mathsf{f}_{\mathbb{N}}$ has a negative coefficient. $\square$

The result of Lemma 15 relies on the assumption that the function symbol s is interpreted by a linear polynomial $\mathsf{s}_{\mathbb{N}}(x) = x + d$. Our next goal is to do away with this assumption by adding rules that enforce such an interpretation for s.

Lemma 16. *In any polynomial interpretation that is compatible with the rewrite rules* $\mathsf{g}(\mathsf{s}(x)) \to \mathsf{s}(\mathsf{s}(\mathsf{g}(x)))$ *and* $\mathsf{f}(\mathsf{g}(x)) \to \mathsf{g}(\mathsf{g}(\mathsf{f}(x)))$*,* $\mathsf{s}_{\mathbb{N}}$ *and* $\mathsf{g}_{\mathbb{N}}$ *must be linear polynomials. Moreover,* $\mathsf{s}_{\mathbb{N}}(x) = x + d$*, for some* $d > 0$*, and* $\mathsf{f}_{\mathbb{N}}$ *is not linear.*

Proof. Without loss of generality, let us assume that the leading terms of $\mathsf{s}_{\mathbb{N}}(x)$ and $\mathsf{g}_{\mathbb{N}}(x)$ are kx^i and mx^j, respectively, with $k, i, m, j \geq 1$. Then the leading term of the polynomial $P_{\text{lhs}} := \mathsf{g}_{\mathbb{N}}(\mathsf{s}_{\mathbb{N}}(x))$ associated with the left-hand side of the first rule is $m(kx^i)^j = mk^jx^{ij}$. Likewise, the leading term of the corresponding polynomial $P_{\text{rhs}} := \mathsf{s}_{\mathbb{N}}(\mathsf{s}_{\mathbb{N}}(\mathsf{g}_{\mathbb{N}}(x)))$ is $k(k(mx^j)^i)^i = k^{i+1}m^{i^2}x^{i^2j}$. Compatibility demands that the degree of the former must be greater than or equal to the degree of the latter, i.e., $ij \geq i^2j$. This condition holds if and only if $i = 1$. Repeating this reasoning for the second rule yields $j = 1$. Substituting these values into the leading terms of P_{lhs} and P_{rhs}, we get mkx and k^2mx, respectively. Hence, P_{lhs} and P_{rhs} have the same degree, such that, in order to ensure compatibility, the leading coefficient of the former must be greater than or equal to the leading coefficient of the latter, i.e., $mk \geq k^2m$. Since $m > 0$ and $k > 0$, this condition is equivalent to $k \leq 1$ and hence $k = 1$. Therefore $\mathsf{s}_{\mathbb{N}}(x) = x + d$. Clearly, $d \neq 0$. Finally, let us assume that f is interpreted by a linear polynomial $\mathsf{f}_{\mathbb{N}}$. Repeating the above reasoning for the second rule yields $\mathsf{g}_{\mathbb{N}}(x) = x + d'$. However, such an interpretation is not compatible with the first rule. Hence, $\mathsf{f}_{\mathbb{N}}$ cannot be linear. $\square$

Having all the relevant ingredients at hand, we are now ready to state the main theorem of this section, which also gives an affirmative answer to the question posed at the beginning of the section; that is, there are TRSs that can be proved terminating by a polynomial interpretation with negative coefficients, but cannot be proved terminating by a polynomial interpretation where the coefficients of all polynomials are non-negative.

Theorem 17. *Consider the TRS* $\mathcal{R}_1$ *extended with the rewrite rules* $\mathsf{g}(\mathsf{s}(x)) \to \mathsf{s}(\mathsf{s}(\mathsf{g}(x)))$ *and* $\mathsf{f}(\mathsf{g}(x)) \to \mathsf{g}(\mathsf{g}(\mathsf{f}(x)))$*. In any compatible polynomial interpretation,* $\mathsf{f}_{\mathbb{N}}$ *must contain at least one monomial with a negative coefficient.*

Proof. By Lemmata 15 and 16. $\square$

Specifying Interpretations by Interpolation

Now let us revisit the motivating scenario presented at the beginning of this section, in which we leveraged polynomial interpolation to create the TRS $\mathcal{R}_1$

in such a way that it enforces the function symbol f to be interpreted by $\mathsf{f}_{\mathbb{N}}(x) = 2x^2 - x + 1$, a polynomial of our choice. The construction presented there was based on three assumptions:

1. the successor symbol s had to be interpreted by $\mathsf{s}_{\mathbb{N}}(x) = x + 1$,
2. the constant $\mathsf{0}$ had to be interpreted by $\mathsf{0}_{\mathbb{N}} = 0$,
3. the function symbol f had to be interpreted by a quadratic polynomial.

Next we show how one can enforce all these assumptions by adding suitable rewrite rules to $\mathcal{R}_1$. This results in a TRS that is polynomially terminating, but only if the symbols s, f and $\mathsf{0}$ are interpreted accordingly (cf. Theorem 23). However, much to our surprise, most of the current termination tools with all their advanced termination techniques fail to prove this TRS terminating (cf. Section 5) in their automatic mode.

Concerning the first two of the above assumptions, it turns out that the constraints imposed by $\mathcal{R}_1$ alone suffice to do away with them, provided that the successor symbol is interpreted by a linear polynomial of the form $x + d$ (which poses no problem according to Lemma 16). This is the result of the next lemma.

Lemma 18. *In any polynomial interpretation compatible with $\mathcal{R}_1$ such that $\mathsf{s}_{\mathbb{N}}(x) = x + d$ and the degree of $\mathsf{f}_{\mathbb{N}}$ is at most two, the constant $\mathsf{0}$ must be interpreted by 0. Moreover, $d = 1$ and $\mathsf{f}_{\mathbb{N}}$ is not linear.*

Proof. Without loss of generality, $\mathsf{f}_{\mathbb{N}}(x) = ax^2 + bx + c$ subject to the constraints $a, c \geq 0$ and $a + b > 0$ (cf. Corollary 8). By Lemma 15, $\mathsf{f}_{\mathbb{N}}$ is not linear; hence $a \geq 1$. Writing z for $\mathsf{0}_{\mathbb{N}}$, the compatibility requirement yields

$$\begin{aligned} z <{}& \mathsf{f}_{\mathbb{N}}(z) < z + 2d \\ z + d <{}& \mathsf{f}_{\mathbb{N}}(z + d) < z + 3d \\ z + 6d <{}& \mathsf{f}_{\mathbb{N}}(z + 2d) < z + 8d \end{aligned}$$

Hence, d must be a positive integer, i.e., $d \geq 1$. Next we focus on the constraint $\mathsf{f}_{\mathbb{N}}(z + d) < z + 3d$ and try to derive a contradiction assuming $z \geq 1$. We reason as follows: $\mathsf{f}_{\mathbb{N}}(z+d) - \mathsf{f}_{\mathbb{N}}(z) = d(2az + ad + b) \geq d(2az + a + b) \geq d(2az + 1) \geq 3d$. Hence, $\mathsf{f}_{\mathbb{N}}(z + d) \geq \mathsf{f}_{\mathbb{N}}(z) + 3d$, which contradicts $\mathsf{f}_{\mathbb{N}}(z + d) < z + 3d$ together with the first of the above constraints $\mathsf{f}_{\mathbb{N}}(z) > z$. As a consequence, $z = \mathsf{0}_{\mathbb{N}} = 0$. Finally, it remains to show that d must be 1. We already know that d must be at least 1. So let us assume that $d \geq 2$ and derive a contradiction with respect to the constraint $\mathsf{f}_{\mathbb{N}}(z + 2d) < z + 8d$. This can be achieved as follows: $\mathsf{f}_{\mathbb{N}}(z + 2d) = \mathsf{f}_{\mathbb{N}}(2d) = 4ad^2 + 2bd + c \geq 4ad^2 + 2bd = d(4ad + 2b) = d((4d - 2)a + 2(a + b)) \geq d(6a + 2(a + b)) \geq d(6a + 2) \geq 8d = z + 8d$. □

Next we will elaborate on how to get rid of the assumption that the function symbol f has to be interpreted by a polynomial $\mathsf{f}_{\mathbb{N}}$ of degree at most two. Again, the idea is to enforce this condition by some additional rewrite rules based on the following observation. If $\mathsf{f}_{\mathbb{N}}$ is at most quadratic, then the function $\mathsf{f}_{\mathbb{N}}(x+d) - \mathsf{f}_{\mathbb{N}}(x)$ is at most linear; i.e., there is a linear function $\mathsf{r}_{\mathbb{N}}(x)$ such that $\mathsf{r}_{\mathbb{N}}(x) > \mathsf{f}_{\mathbb{N}}(x + d) - \mathsf{f}_{\mathbb{N}}(x)$, or equivalently, $\mathsf{f}_{\mathbb{N}}(x) + \mathsf{r}_{\mathbb{N}}(x) > \mathsf{f}_{\mathbb{N}}(x + d)$, for all $x \in \mathbb{N}$. This can be encoded in terms of the rewrite rule $\mathsf{h}(\mathsf{f}(x), \mathsf{r}(x)) \to \mathsf{f}(\mathsf{s}(x))$, as soon as the interpretation of h corresponds to the addition of two natural numbers. Yet this does not pose a major problem, as will be shown shortly.

Remark 19. Note that the construction motivated above is actually more general than it seems at first sight. That is, it can be used to set arbitrary upper bounds on the degree of an interpretation (cf. proof of Lemma 20). Moreover, it can easily be adapted to establish lower bounds.

Lemma 20. *Consider the rewrite rule* $\mathsf{h}(\mathsf{f}(x), \mathsf{r}(x)) \to \mathsf{f}(\mathsf{s}(x))$. *In any compatible polynomial interpretation where* $\mathsf{s}_{\mathbb{N}}(x) = x + d$ $(d \geq 1)$, $\mathsf{r}_{\mathbb{N}}$ *is some linear polynomial, and* $\mathsf{h}_{\mathbb{N}}(x, y) = x + y + p$ $(p \in \mathbb{N})$, *the degree of* $\mathsf{f}_{\mathbb{N}}$ *is at most two.*

Proof. Without loss of generality, let $\mathsf{f}_{\mathbb{N}}(x) = \sum_{i=0}^{n} a_i x^i$ $(a_n > 0)$. By compatibility with the single rewrite rule, the inequality

$$\mathsf{f}_{\mathbb{N}}(x) + \mathsf{r}_{\mathbb{N}}(x) + p > \mathsf{f}_{\mathbb{N}}(x + d) \tag{8}$$

must be satisfied for all $x \in \mathbb{N}$. Using Taylor's theorem,

$$\mathsf{f}_{\mathbb{N}}(x + d) = \sum_{k=0}^{n} \frac{d^k}{k!} \mathsf{f}_{\mathbb{N}}^{(k)}(x) = \mathsf{f}_{\mathbb{N}}(x) + d\,\mathsf{f}'_{\mathbb{N}}(x) + \frac{d^2}{2} \mathsf{f}''_{\mathbb{N}}(x) + \ldots + \frac{d^n}{n!} \mathsf{f}_{\mathbb{N}}^{(n)}(x)$$

we can simplify (8) to

$$\mathsf{r}_{\mathbb{N}}(x) + p > d\,\mathsf{f}'_{\mathbb{N}}(x) + \sum_{k=2}^{n} \frac{d^k}{k!} \mathsf{f}_{\mathbb{N}}^{(k)}(x) \tag{9}$$

As $d \geq 1$, the right-hand side of this inequality is a polynomial of degree $n - 1$ whose leading coefficient dna_n is positive, whereas the degree of the left-hand side is one. But by compatibility, the former must be greater than or equal to $n - 1$; i.e., $n \leq 2$. $\square$

It remains to show how the interpretation of h can be fixed to addition.

Lemma 21. *Consider the TRS* $\mathcal{R}_2$ *consisting of the rules*

$$\mathsf{g}(x) \to \mathsf{h}(x, x) \qquad \mathsf{s}(x) \to \mathsf{h}(x, \mathsf{0}) \qquad \mathsf{s}(x) \to \mathsf{h}(\mathsf{0}, x)$$

Any compatible polynomial interpretation that interprets s *by* $\mathsf{s}_{\mathbb{N}}(x) = x + d$ *and* g *by a linear polynomial satisfies* $\mathsf{h}_{\mathbb{N}}(x, y) = x + y + p$, $p \in \mathbb{N}$. *Moreover, if* $d = 1$, *then* $p = 0$ *and* $\mathsf{0}_{\mathbb{N}} = 0$.

Proof. Without loss of generality, let $\mathsf{0}_{\mathbb{N}} = z$ for some $z \in \mathbb{N}$. Because $\mathsf{g}_{\mathbb{N}}$ is linear, compatibility with the first rule constrains the function $h' \colon \mathbb{N} \to \mathbb{N}, x \mapsto \mathsf{h}_{\mathbb{N}}(x, x)$ to be at most linear. This can only be the case if $\mathsf{h}_{\mathbb{N}}$ contains no monomials of degree two or higher. In other words, $\mathsf{h}_{\mathbb{N}}(x, y) = p_x \cdot x + p_y \cdot y + p$, where $p \in \mathbb{N}$ (because of well-definedness), $p_x \geq 1$ and $p_y \geq 1$ (because of strict monotonicity).

Then compatibility with the second rule translates to $x + d > p_x \cdot x + p_y \cdot z + p$ for all $x \in \mathbb{N}$, which holds if and only if $p_x \leq 1$ and $d > p_y \cdot z + p$. Hence, $p_x = 1$, and by analogous reasoning with respect to the third rule, $p_y = 1$. Finally, if $d = 1$ then the condition $d > p_y \cdot z + p$ simplifies to $1 > z + p$, which holds if and only if $z = 0$ and $p = 0$ (because both z and p are non-negative). $\square$

Corollary 22. *Consider the TRS $\mathcal{R}_3$ consisting of the rules*

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

Any polynomial interpretation compatible with $\mathcal{R}_2 \cup \mathcal{R}_3$ requires degree at most two for $\mathsf{f}_\mathbb{N}$.

Proof. By Lemma 16, $\mathsf{s}_\mathbb{N}(x) = x+d$ $(d \geq 1)$ and $\mathsf{g}_\mathbb{N}(x)$ is linear. Hence, $\mathsf{h}_\mathbb{N}(x, y) = x+y+p$ $(p \in \mathbb{N})$ according to Lemma 21. Finally, Lemma 20 applied to the rule $\mathsf{h}(\mathsf{f}(x), \mathsf{g}(x)) \to \mathsf{f}(\mathsf{s}(x))$ proves the claim. $\square$

Now, combining Lemma 18 and Corollary 22 yields natural semantics for the symbols $\mathsf{0}$ and s and fixes $\mathsf{f}_\mathbb{N}(x) = 2x^2 - x + 1$, as originally desired.

Theorem 23. *Any polynomial interpretation compatible with $\mathcal{R}_1 \cup \mathcal{R}_2 \cup \mathcal{R}_3$ interprets $\mathsf{0}$ by 0 and s by $\mathsf{s}_\mathbb{N}(x) = x + 1$. Moreover, $\mathsf{f}_\mathbb{N}(x) = 2x^2 - x + 1$.*

Proof. From Lemmata 18 and 16 and Corollary 22, we infer that $\mathsf{0}_\mathbb{N} = 0$, $\mathsf{s}_\mathbb{N}(x) = x + 1$ and $\mathsf{f}_\mathbb{N}$ is a quadratic polynomial. Without loss of generality, $\mathsf{f}_\mathbb{N}(x) = ax^2 + bx + c$. Next we observe that $\mathcal{R}_1$ gives rise to the constraints $\mathsf{f}_\mathbb{N}(0) = 1$, $\mathsf{f}_\mathbb{N}(1) = 2$ and $\mathsf{f}_\mathbb{N}(2) = 7$, which uniquely determine the coefficients $a = 2$, $b = -1$ and $c = 1$ of $\mathsf{f}_\mathbb{N}$ by polynomial interpolation. $\square$

In order for Theorem 23 to be relevant, it remains to show that there actually exists a compatible polynomial interpretation. This is achieved, e.g., by defining $\mathsf{0}_\mathbb{N} = 0$, $\mathsf{s}_\mathbb{N}(x) = x + 1$, $\mathsf{f}_\mathbb{N}(x) = 2x^2 - x + 1$, $\mathsf{h}_\mathbb{N}(x, y) = x + y$ and $\mathsf{g}_\mathbb{N}(x) = 4x + 5$.

Remark 24. One can show that any polynomial interpretation compatible with the TRS $\mathcal{S} := \mathcal{R}_2 \cup \mathcal{R}_3 \cup \{\, \mathsf{s}(\mathsf{s}(\mathsf{0})) \to \mathsf{f}(\mathsf{s}(\mathsf{0}))\,\}$ must interpret $\mathsf{0}$ by 0 and s by $\mathsf{s}_\mathbb{N}(x) = x + 1$. Thus we can take our favourite univariate polynomial P, which must of course be both strictly monotone and well-defined, and design a TRS such that the interpretation of some unary function symbol k is fixed to it. To this end, we extend $\mathcal{S}$ by suitable rewrite rules encoding interpolation constraints for the symbol k and additional rules that set an upper bound on the degree of the interpretation of k, which corresponds to the degree of P.

5 Experimental Results

We implemented the criterion from Theorem 7 in the termination prover $\mathsf{T_TT_2}$.[2] The problem of finding suitable coefficients for the polynomials is formulated as a set of diophantine constraints (as in [4]) which are solved by a transformation to SAT. Simple heuristics are applied to decide which symbols should be interpreted

[2] See http://termination-portal.org/wiki/Category:Tools

by non-linear polynomials (e.g., defined function symbols, symbols that appear at most once on every left and right-hand side, symbols that do not appear nested). Using coefficients in $\{-8, \ldots, 7\}$ and either of the latter two heuristics, $\mathsf{T_TT_2}$ finds a compatible interpretation (i.e., the one mentioned at the end of Section 4) for the TRS in Theorem 23 fully automatically within five seconds. We remark that implementing Theorems 7 and 11 is about as expensive as the absolute positiveness approach, since the size of the search space is mainly determined by the degree of the polynomials.

Despite the tremendous progress in automatic termination proving during the last decade, it is remarkable that the other powerful termination tools AProVE[2] and JAMBOX[2] cannot prove this system terminating within ten minutes. The same holds for $\mathsf{T_TT_2}$ without the criterion from Theorem 7. Surprisingly, the 2006 version of TPA[2] finds a lengthy termination proof based on semantic labeling. However, it is straightforward to generate a variant of the TRS from Theorem 23 that is orientable if $\mathsf{f}_{\mathbb{N}}(0) = 0$, $\mathsf{f}_{\mathbb{N}}(1) = 1$, and $\mathsf{f}_{\mathbb{N}}(2) = 8$, suggesting $\mathsf{f}_{\mathbb{N}}(x) = 3x^2 - 2x$. While $\mathsf{T_TT_2}$ can still prove this system terminating, TPA now also fails. Moreover, due to Remark 24 one can generate myriads of TRSs that can easily be shown to be polynomially terminating but where an automated termination proof is out of reach for current termination analyzers.

Acknowledgements. We thank the anonymous referees for their helpful comments.

References

1. Arts, T., Giesl, J.: Termination of term rewriting using dependency pairs. TCS 236(1-2), 133–178 (2000)
2. Contejean, E., Marché, C., Tomás, A.P., Urbain, X.: Mechanically proving termination using polynomial interpretations. JAR 34(4), 325–363 (2005)
3. Dershowitz, N.: A note on simplification orderings. IPL 9(5), 212–215 (1979)
4. Fuhs, C., Giesl, J., Middeldorp, A., Schneider-Kamp, P., Thiemann, R., Zankl, H.: SAT solving for termination analysis with polynomial interpretations. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 340–354. Springer, Heidelberg (2007)
5. Fuhs, C., Giesl, J., Middeldorp, A., Schneider-Kamp, P., Thiemann, R., Zankl, H.: Maximal termination. In: Voronkov, A. (ed.) RTA 2008. LNCS, vol. 5117, pp. 110–125. Springer, Heidelberg (2008)
6. Giesl, J., Thiemann, R., Schneider-Kamp, P.: The dependency pair framework: Combining techniques for automated termination proofs. In: Baader, F., Voronkov, A. (eds.) LPAR 2004. LNCS (LNAI), vol. 3452, pp. 301–331. Springer, Heidelberg (2005)
7. Hirokawa, N., Middeldorp, A.: Automating the dependency pair method. I&C 199(1-2), 172–199 (2005)
8. Hirokawa, N., Middeldorp, A.: Tyrolean Termination Tool: Techniques and features. I&C 205(4), 474–511 (2007)
9. Hong, H., Jakuš, D.: Testing positiveness of polynomials. JAR 21(1), 23–38 (1998)
10. Lankford, D.: On proving term rewrite systems are noetherian. Tech. Rep. MTP-3, Louisiana Technical University, Ruston (1979)
11. Lucas, S.: Polynomials over the reals in proofs of termination: From theory to practice. TIA 39(3), 547–586 (2005)

Termination Tools in Ordered Completion*

Sarah Winkler and Aart Middeldorp

Institute of Computer Science, University of Innsbruck, Austria

Abstract. Ordered completion is one of the most frequently used calculi in equational theorem proving. The performance of an ordered completion run strongly depends on the reduction order supplied as input. This paper describes how termination tools can replace fixed reduction orders in ordered completion procedures, thus allowing for a novel degree of automation. Our method can be combined with the multi-completion approach proposed by Kondo and Kurihara. We present experimental results obtained with our ordered completion tool $\mathsf{omkb}_{\mathsf{TT}}$ for both ordered completion and equational theorem proving.

1 Introduction

Unfailing completion introduced by Bachmair, Dershowitz and Plaisted [2] aims to transform a set of equations into a ground-confluent and terminating system. Underlying many completion-based theorem proving systems, it has become a well-known calculus in automated reasoning. In contrast to standard completion [7], *ordered completion*, as it is called nowadays, always succeeds (in theory). The reduction order supplied as input is nevertheless a critical parameter when it comes to performance issues.

With *multi-completion*, Kondo and Kurihara [9] proposed a completion calculus that employs multiple reduction orders in parallel. It is applicable to both standard and ordered completion, and more efficient than a parallel execution of the respective processes. Wehrman, Stump and Westbrook [16] introduced a variant of standard completion that utilizes a termination prover instead of a fixed reduction order. The tool Slothrop demonstrates the potential of this approach by completing systems that cannot be handled by traditional completion procedures. In [11] it was shown how multi-completion and the use of termination tools can be combined. When implemented in the tool $\mathsf{mkb}_{\mathsf{TT}}$, this approach could cope with input systems that were not completed by Slothrop.

The current paper describes how termination tools can replace reduction orders in ordered completion procedures. In contrast to standard completion using termination provers, two challenges have to be faced. First of all, ordered completion procedures require the termination order to be totalizable for the theory. When using termination tools, the order which is synthesized during the termination proving process need not have this property. Second, the standard notion

* This research is supported by FWF (Austrian Science Fund) project P18763. The first author is supported by a DOC-fFORTE fellowship of the Austrian Academy of Sciences.

of fairness, which determines which (extended) critical pairs need to be computed to ensure correctness, depends on the (final) reduction order, which is not known in advance. We explain how to overcome these challenges, also in a multi-completion setting. We further show how ordered multi-completion with termination tools can be used for equational theorem proving.

The remainder of the paper is organized as follows. Section 2 summarizes definitions, inference systems and main results in the context of (ordered) completion which will be needed in the sequel. Section 3 describes the calculus oKBtt for ordered completion with termination tools. The results obtained in Section 3 are extended to oMKBtt, a calculus for ordered multi-completion with termination tools, in Section 4. More application-specific, we outline in Section 5 how oMKBtt can be used for refutational theorem proving. In Section 6 we briefly describe our tool $\mathsf{omkb}_{\mathsf{TT}}$ that implements the calculus oMKBtt. Experimental results are given in Section 7 before we add some concluding remarks in Section 8. For reasons of space, several proofs are missing. They can be found in the report version of the paper which can be obtained from the accompanying website.[1]

2 Preliminaries

We consider terms $\mathcal{T}(\mathcal{F},\mathcal{V})$ over a finite signature $\mathcal{F}$ and a set of variables $\mathcal{V}$. Terms without variables are *ground*. Sets of equations between terms will be denoted by $\mathcal{E}$ and are assumed to be symmetric. The associated *equational theory* is denoted by $\approx_{\mathcal{E}}$. As usual a set of directed equations $l \to r$ is called a rewrite system and denoted by $\mathcal{R}$, and $\to_{\mathcal{R}}$ is the associated *rewrite relation*. We write $s \xrightarrow{l\to r}_p t$ to express that $s \to_{\mathcal{R}} t$ was achieved by applying the rule $l \to r \in \mathcal{R}$ at position p. The relations $\to^+_{\mathcal{R}}$, $\to^*_{\mathcal{R}}$ and $\leftrightarrow_{\mathcal{R}}$ denote the transitive, transitive-reflexive and symmetric closure of $\to_{\mathcal{R}}$. The smallest equivalence relation containing $\to_{\mathcal{R}}$, which coincides with the equational theory $\approx_{\mathcal{R}}$ if $\mathcal{R}$ is considered as a set of equations, is denoted by $\leftrightarrow^*_{\mathcal{R}}$. Subscripts are omitted if the rewrite system or the set of equations is clear from the context.

A rewrite system $\mathcal{R}$ is *terminating* if it does not admit infinite rewrite sequences. It is *confluent* if for every peak $t \mathrel{{}^*{\leftarrow}} s \to^* u$ there exists a term v such that $t \to^* v \mathrel{{}^*{\leftarrow}} u$. $\mathcal{R}$ is *ground-confluent* if this property holds for all ground terms s. A rewrite system $\mathcal{R}$ with the property that for every rewrite rule $l \to r$ the right-hand side r is in normal form and the left-hand side l is in normal form with respect to $\mathcal{R} \setminus \{l \to r\}$ is called *reduced*. A rewrite system which is both terminating and (ground-)confluent is called (ground-)*complete*. We call $\mathcal{R}$ *complete for* a set of equations $\mathcal{E}$ if $\mathcal{R}$ is complete and $\leftrightarrow^*_{\mathcal{R}}$ coincides with $\approx_{\mathcal{E}}$.

A proper order $\succ$ on terms is a *rewrite order* if it is closed under contexts and substitutions. A well-founded rewrite order is called a *reduction order*. The relation $\to^+_{\mathcal{R}}$ is a reduction order for every terminating rewrite system $\mathcal{R}$. A reduction order $\succ$ is *complete for* a set of equations $\mathcal{E}$ if $s \succ t$ or $t \succ s$ holds

[1] See http://cl-informatik.uibk.ac.at/software/omkbtt

deduce$_2$	$\dfrac{\mathcal{E}, \mathcal{R}}{\mathcal{E} \cup \{s \approx t\}, \mathcal{R}}$	if $s \xleftarrow{r_1 \leftarrow l_1} u \xrightarrow{l_2 \to r_2} t$ where $l_1 \approx r_1, l_2 \approx r_2 \in \mathcal{R} \cup \mathcal{E}$ and $r_i \not\succ l_i$
simplify$_2$	$\dfrac{\mathcal{E} \cup \{s \approx t\}, \mathcal{R}}{\mathcal{E} \cup \{s \approx u\}, \mathcal{R}}$	if $t \xrightarrow{l\sigma \to r\sigma} u$ using $l \approx r \in \mathcal{E}$ where $t \rhd l$[a] and $l\sigma \succ r\sigma$
compose$_2$	$\dfrac{\mathcal{E}, \mathcal{R} \cup \{s \to t\}}{\mathcal{E}, \mathcal{R} \cup \{s \to u\}}$	if $t \xrightarrow{l\sigma \to r\sigma} u$ using $l \approx r \in \mathcal{E}$ and $l\sigma \succ r\sigma$
collapse$_2$	$\dfrac{\mathcal{E}, \mathcal{R} \cup \{t \to s\}}{\mathcal{E} \cup \{u \approx s\}, \mathcal{R}}$	if $t \xrightarrow{l\sigma \to r\sigma} u$ using $l \approx r \in \mathcal{E}$ where $t \rhd l$ and $l\sigma \succ r\sigma$

[a] $\rhd$ denotes the strict encompassment relation

Fig. 1. Additional inference rules for ordered completion (oKB)

for all ground terms s and t that satisfy $s \approx_{\mathcal{E}} t$. In the sequel we will consider lexicographic path orders (LPO [6]), Knuth-Bendix orders (KBO [7]), multiset path orders (MPO [4]) and orders induced by polynomial interpretations [10]. The first two are total on ground terms if the associated precedence is total. Orders induced by MPOs and polynomial interpretations can always be extended to an order with that property. Reduction orders that are total on ground terms are of course complete for any theory.

2.1 Ordered Completion

We assume that the reader is familiar with standard completion, originally proposed by Knuth and Bendix [7] and later on formulated as an inference system [1]. This inference system will in the sequel be referred to as KB. For ordered completion (oKB) [2] the inference system of standard completion is extended with the rules depicted in Fig. 1, where $\succ$ denotes the reduction order used.

An inference sequence $(\mathcal{E}_0, \mathcal{R}_0) \vdash (\mathcal{E}_1, \mathcal{R}_1) \vdash (\mathcal{E}_2, \mathcal{R}_2) \dots$ is called a *deduction* with *persistent* equalities $\mathcal{E}_\omega = \bigcup_i \bigcap_{j>i} \mathcal{E}_j$ and rules $\mathcal{R}_\omega = \bigcup_i \bigcap_{j>i} \mathcal{R}_j$.

Definition 1. *An equation $s \approx t$ is an* extended critical pair *with respect to a set of equations $\mathcal{E}$ and a reduction order $\succ$ if there are a term u and rewrite steps $u \xrightarrow{l_1\sigma \to r_1\sigma}_\epsilon s$ and $u \xrightarrow{l_2\sigma \to r_2\sigma}_p t$ such that $l_1 \approx r_1, l_2 \approx r_2 \in \mathcal{E}$ and $r_i\sigma \not\succ l_i\sigma$. The set of extended critical pairs among equations in $\mathcal{E}$ is denoted by $CP_\succ(\mathcal{E})$.*

An oKB deduction is *fair* if $\mathrm{CP}_\succ(\mathcal{E}_\omega \cup \mathcal{R}_\omega) \subseteq \bigcup_i \mathcal{E}_i$. The following theorems from [2] state the correctness and completeness of oKB.

Theorem 2. *Let $\mathcal{E}$ be a set of equations and $\succ$ a reduction order that can be extended to a reduction order $>$ which is complete for $\mathcal{E}$. Any fair oKB run will on inputs $(\mathcal{E}, \varnothing)$ and $\succ$ generate a system $\mathcal{E}_\omega \cup \mathcal{R}_\omega$ that is ground-complete with respect to $>$.*

$$\boxed{\mathsf{orient}\quad \frac{\mathcal{E}\cup\{s\approx t\},\mathcal{R},\mathcal{C}}{\mathcal{E},\mathcal{R}\cup\{s\to t\},\mathcal{C}\cup\{s\to t\}}\quad \text{if } \mathcal{C}\cup\{s\to t\} \text{ terminates}}$$

Fig. 2. The orient inference rule in KBtt

An oKB completion procedure is *simplifying* if for all inputs $\mathcal{E}_0$ and $\succ$ the rewrite system $\mathcal{R}_\omega$ is reduced and all equations $u \approx v$ in $\mathcal{E}_\omega$ are both unorientable with respect to $\succ$ and irreducible in $\mathcal{R}_\omega$.

Theorem 3. *Assume $\mathcal{R}$ is a reduced and complete rewrite system for $\mathcal{E}$ that is contained in a reduction order $\succ$ which can be extended to a complete reduction order for $\mathcal{E}$. Any fair and simplifying oKB run that starts from $(\mathcal{E},\varnothing)$ using $\succ$ yields $\mathcal{E}_\omega = \varnothing$ and $\mathcal{R}_\omega = \mathcal{R}$.*

In the requirement for a reduction order that is totalizable for the theory, ordered completion differs from standard completion. The more recent approach of Bofill *et al.* [3] lacks this restriction, but the obtained completion procedure is only of theoretical interest as it relies on enumerating all ground equational consequences of the theory $\mathcal{E}$.

2.2 Completion with Termination Tools

The inference system KBtt [16] for standard completion with termination tools operates on tuples $(\mathcal{E},\mathcal{R},\mathcal{C})$ consisting of a set of equations $\mathcal{E}$, and rewrite systems $\mathcal{R}$ and $\mathcal{C}$. The latter is called the *constraint system.* KBtt consists of the orient rule depicted in Fig. 2 together with the remaining KB rules where the constraint component is not modified.

Correctness and completeness of KBtt follow from the fact that any run of standard completion can be simulated by KB and vice versa [16].

2.3 Completion with Multiple Reduction Orders

Multi-completion (MKB), introduced by Kurihara and Kondo [9] considers a set of reduction orders $\mathcal{O} = \{\succ_1,\ldots,\succ_n\}$. To share inferences for different orders, a special data structure is used.

Definition 4. *A* node *is a tuple $\langle s : t, R_0, R_1, E\rangle$ where the* data *s, t are terms and the* labels *R_0, R_1, E are subsets of $\mathcal{O}$ such that R_0, R_1 and E are mutually disjoint, $s \succ_i t$ for all $\succ_i \in R_0$, and $t \succ_i s$ for all $\succ_i \in R_1$.*

The intuition is that given a node $\langle s : t, R_0, R_1, E\rangle$, all orders in the *equation label* E consider the data as an equation $s \approx t$ while orders in the *rewrite labels* R_0 and R_1 regard it as rewrite rules $s \to t$ and $t \to s$, respectively. Hence $\langle s : t, R_0, R_1, E\rangle$ is identified with $\langle t : s, R_1, R_0, E\rangle$.

MKB is described by an inference system consisting of five rules. Fig. 3 shows the orient inference rule. As shown in [9], slight modifications to the rewrite inference rules allow to perform ordered multi-completion (oMKB).

$$\text{orient} \quad \frac{\mathcal{N} \cup \{\langle s : t, R_0, R_1, E \uplus R\rangle\}}{\mathcal{N} \cup \{\langle s : t, R_0 \cup R, R_1, E\rangle\}} \quad \text{if } R \neq \varnothing \text{ and } s \succ_i t \text{ for all } \succ_i \in R$$

Fig. 3. orient in MKB

3 Ordered Completion with Termination Tools

This section describes how the ideas of KBtt can be incorporated into ordered completion procedures. The derived method will in the sequel be referred to as oKBtt. It is described by an inference system consisting of the rules depicted in Fig. 4 together with orient, delete, simplify, compose and collapse from KBtt.

deduce$_2$	$\dfrac{\mathcal{E}, \mathcal{R}, \mathcal{C}}{\mathcal{E} \cup \{s \approx t\}, \mathcal{R}, \mathcal{C}}$	if $s \leftarrow_{\mathcal{E}\cup\mathcal{R}} u \rightarrow_{\mathcal{E}\cup\mathcal{R}} t$
simplify$_2$	$\dfrac{\mathcal{E} \cup \{s \approx t\}, \mathcal{R}, \mathcal{C}}{\mathcal{E} \cup \{s \approx u\}, \mathcal{R}, \mathcal{C} \cup \{l\sigma \to r\sigma\}}$	if $t \xrightarrow{l\sigma \to r\sigma} u$ using $l \approx r \in \mathcal{E}$ where $t \rhd l$ and $\mathcal{C} \cup \{l\sigma \to r\sigma\}$ terminates
compose$_2$	$\dfrac{\mathcal{E}, \mathcal{R} \cup \{s \to t\}, \mathcal{C}}{\mathcal{E}, \mathcal{R} \cup \{s \to u\}, \mathcal{C} \cup \{l\sigma \to r\sigma\}}$	if $t \xrightarrow{l\sigma \to r\sigma} u$ using $l \approx r \in \mathcal{E}$ and $\mathcal{C} \cup \{l\sigma \to r\sigma\}$ terminates
collapse$_2$	$\dfrac{\mathcal{E}, \mathcal{R} \cup \{t \to s\}, \mathcal{C}}{\mathcal{E} \cup \{u \approx s\}, \mathcal{R}, \mathcal{C} \cup \{l\sigma \to r\sigma\}}$	if $t \xrightarrow{l\sigma \to r\sigma} u$ using $l \approx r \in \mathcal{E}$ where $t \rhd l$ and $\mathcal{C} \cup \{l\sigma \to r\sigma\}$ terminates

Fig. 4. Ordered completion with termination tools (oKBtt)

An inference sequence $(\mathcal{E}_0, \mathcal{R}_0, \mathcal{C}_0) \vdash (\mathcal{E}_1, \mathcal{R}_1, \mathcal{C}_1) \vdash (\mathcal{E}_2, \mathcal{R}_2, \mathcal{C}_2) \vdash \cdots$ with respect to oKBtt is called an oKBtt run and denoted by γ. Persistent equations $\mathcal{E}_\omega$ and rules $\mathcal{R}_\omega$ are defined as for oKB. The set $\mathcal{C}_\omega = \bigcup_i \mathcal{C}_i$ collects persistent constraint rules. We write $(\mathcal{E}_0, \mathcal{R}_0) \vdash^* (\mathcal{E}_\alpha, \mathcal{R}_\alpha)$ to express that the run has length α, where $\alpha = \omega$ if it is not finite.

Example 5. If oKBtt is run on the input equations $\mathsf{g}(\mathsf{f}(x, \mathsf{b})) \approx \mathsf{a}$ and $\mathsf{f}(\mathsf{g}(x), y) \approx \mathsf{f}(x, \mathsf{g}(y))$ and all termination checks are performed with respect to the polynomial interpretation $[\mathsf{f}](x, y) = x + 2y + 1$, $[\mathsf{g}](x) = x + 1$ and $[\mathsf{a}] = [\mathsf{b}] = [\mathsf{c}] = 0$, the following system is derived:

$$\mathcal{E} = \left\{ \begin{array}{l} \mathsf{f}(\mathsf{f}(x,\mathsf{b}),\mathsf{a}) \approx \mathsf{f}(\mathsf{c},\mathsf{f}(y,\mathsf{b})) \\ \mathsf{f}(\mathsf{f}(x,\mathsf{b}),\mathsf{a}) \approx \mathsf{f}(\mathsf{f}(y,\mathsf{b}),\mathsf{a}) \\ \mathsf{f}(\mathsf{c},\mathsf{f}(x,\mathsf{b})) \approx \mathsf{f}(\mathsf{c},\mathsf{f}(y,\mathsf{b})) \end{array} \right\} \quad \mathcal{R} = \left\{ \begin{array}{c} \mathsf{g}(\mathsf{f}(x,\mathsf{b})) \to \mathsf{a} \\ \mathsf{f}(x,\mathsf{g}(y)) \to \mathsf{f}(\mathsf{g}(x),y) \\ \mathsf{f}(\mathsf{g}(x),\mathsf{f}(y,\mathsf{b})) \to \mathsf{f}(x,\mathsf{c}) \end{array} \right\}$$

However, if the second equation would be oriented from left to right, the oKBtt run diverges. Since $\mathsf{f}(x, \mathsf{g}(y)) \to \mathsf{f}(\mathsf{g}(x), y)$ cannot be oriented by any KBO or LPO which compares lists of subterms only from left to right, ordered completion tools that do not support other termination methods (e.g. Waldmeister) cannot derive a ground-complete system.

Before showing that oKBtt runs can be simulated by ordered completion runs, and vice versa, we note that oKBtt is sound in that it does not change the equational theory.

Lemma 6. *For every oKBtt step* $(\mathcal{E}, \mathcal{R}, \mathcal{C}) \vdash (\mathcal{E}', \mathcal{R}', \mathcal{C}')$ *the relations* $\leftrightarrow^*_{\mathcal{E} \cup \mathcal{R}}$ *and* $\leftrightarrow^*_{\mathcal{E}' \cup \mathcal{R}'}$ *coincide.*

Lemma 7. *For every finite oKBtt run* $(\mathcal{E}_0, \mathcal{R}_0, \mathcal{C}_0) \vdash^* (\mathcal{E}_n, \mathcal{R}_n, \mathcal{C}_n)$ *such that* $\mathcal{R}_0 \subseteq \to^+_{\mathcal{C}_0}$*, there is a corresponding oKB run* $(\mathcal{E}_0, \mathcal{R}_0) \vdash^* (\mathcal{E}_n, \mathcal{R}_n)$ *using the reduction order* $\to^+_{\mathcal{C}_n}$*.*

Proof. Let $\succ_n$ denote $\to^+_{\mathcal{C}_n}$. We use induction on n. The claim is trivially true for $n = 0$. For a run of the form $(\mathcal{E}_0, \mathcal{R}_0, \mathcal{C}_0) \vdash^* (\mathcal{E}_n, \mathcal{R}_n, \mathcal{C}_n) \vdash (\mathcal{E}_{n+1}, \mathcal{R}_{n+1}, \mathcal{C}_{n+1})$, the induction hypothesis yields a corresponding oKB run $(\mathcal{E}_0, \mathcal{R}_0) \vdash^* (\mathcal{E}_n, \mathcal{R}_n)$ using the reduction order $\succ_n$. Since constraint rules are never removed we have $\mathcal{C}_k \subseteq \mathcal{C}_{n+1}$ for all $k \leq n$, so the same run can be obtained with $\succ_{n+1}$. Case distinction on the applied oKBtt rule shows that a step $(\mathcal{E}_n, \mathcal{R}_n) \vdash (\mathcal{E}_{n+1}, \mathcal{R}_{n+1})$ using $\succ_{n+1}$ is possible.

If orient added the rule $s \to t$ then $s \succ_{n+1} t$ holds by definition, so oKB can apply orient as well. In case simplify$_2$, compose$_2$ or collapse$_2$ was applied using an instance $l\sigma \to r\sigma$ of an equation in $\mathcal{E}_n$, we have $l\sigma \succ_{n+1} r\sigma$ by definition of the inference rules, hence the respective oKB step can be applied. Clearly, in the remaining cases the inference step can be simulated by the corresponding oKB rule since no conditions on the order are involved. □

Lemma 7 does not generalize to infinite runs; as also remarked in [16], $\to^+_{\mathcal{C}_\omega}$ is not necessarily a reduction order since an infinite union of terminating rewrite systems need not be terminating.

Simulating oKB by oKBtt is also complete as stated below. The straightforward proof can be found in the report version. It uses the fact that the reduction order supplied to oKB can be used for termination checks.

Lemma 8. *For every oKB run* $(\mathcal{E}_0, \mathcal{R}_0) \vdash^* (\mathcal{E}_\alpha, \mathcal{R}_\alpha)$ *of length* $\alpha \leq \omega$ *using a reduction order* $\succ$*, there exists an oKBtt run* $(\mathcal{E}_0, \mathcal{R}_0, \mathcal{C}_0) \vdash^* (\mathcal{E}_\alpha, \mathcal{R}_\alpha, \mathcal{C}_\alpha)$ *such that* $\mathcal{C}_\alpha \subseteq \succ$ *holds.*

Totalizability

Lemma 7 shows that an oKBtt run resulting in the final constraint system $\mathcal{C}$ can be simulated by ordered completion using the reduction order $\to^+_{\mathcal{C}}$. If this order should play the role of $\succ$ in Theorem 2 then it has to be contained in a reduction order $>$ which is complete for the theory. Unfortunately, such an order does not always exist. In the proof of the extended critical pair lemma [2], totalizability of the reduction order is needed to guarantee joinability of variable overlaps. Thus, if an oKBtt procedure outputs $\mathcal{E}$, $\mathcal{R}$ and $\mathcal{C}$ such that $\to^+_{\mathcal{C}}$ cannot be extended to a complete order for the theory, ground-confluence of $(\mathcal{E}, \mathcal{R})$ is not guaranteed.

Example 9. A fair oKBtt run starting from

$$\mathcal{E}_0 = \left\{ \begin{matrix} \mathsf{f}(\mathsf{a}+\mathsf{c}) \approx \mathsf{f}(\mathsf{c}+\mathsf{a}) & \mathsf{a} \approx \mathsf{b} \\ \mathsf{g}(\mathsf{c}+\mathsf{b}) \approx \mathsf{g}(\mathsf{b}+\mathsf{c}) & x+y \approx y+x \end{matrix} \right\}$$

might produce the following result:

$$\mathcal{E} = \{x+y \approx y+x\} \qquad \mathcal{R} = \left\{ \begin{matrix} \mathsf{a} \to \mathsf{b} \\ \mathsf{f}(\mathsf{b}+\mathsf{c}) \to \mathsf{f}(\mathsf{c}+\mathsf{b}) \\ \mathsf{g}(\mathsf{c}+\mathsf{b}) \to \mathsf{g}(\mathsf{b}+\mathsf{c}) \end{matrix} \right\}$$

with $\mathcal{C} = \mathcal{R} \cup \{\mathsf{f}(\mathsf{a}+\mathsf{c}) \to \mathsf{f}(\mathsf{c}+\mathsf{a})\}$. No reduction order $>$ extending $\to_\mathcal{C}^+$ can orient the ground instance $\mathsf{c}+\mathsf{a} \approx \mathsf{a}+\mathsf{c}$ from left to right. So $\mathsf{a}+\mathsf{c} > \mathsf{c}+\mathsf{a}$ must hold. This gives rise to the variable overlap $\mathsf{b}+\mathsf{c} \leftarrow \mathsf{a}+\mathsf{c} \to \mathsf{c}+\mathsf{a} \to \mathsf{c}+\mathsf{b}$. As $\mathsf{b}+\mathsf{c}$ and $\mathsf{c}+\mathsf{b}$ have to be incomparable in $>$ the overlap is not joinable.

To solve this problem we restrict the termination checks in oKBtt inferences.

Definition 10. *An* oKBtt$_\mathcal{P}$ procedure *refers to any program which implements the inference rules of oKBtt and employs the termination strategy $\mathcal{P}$ for termination checks in orient, simplify$_2$, compose$_2$ and collapse$_2$ inferences. An oKBtt$_{\mathsf{total}}$ procedure is an oKBtt$_\mathcal{P}$ procedure where $\mathcal{P}$ ensures* total *termination [15, Section 6.3.2] of the checked system.*

Examples of such termination strategies are LPO, KBO and MPO with total precedences as well as polynomial interpretations over $\mathbb{N}$.

Thus, for any constraint system $\mathcal{C}_n$ derived by an oKBtt$_{\mathsf{total}}$ procedure in finitely many steps, there is a reduction order $>$ extending $\to_{\mathcal{C}_n}^+$ which is total on ground terms.

Fairness

Theorem 2 requires a run to be *fair*, meaning that all extended critical pairs among persistent equations and rules are considered. In the context of oKBtt, the set of extended critical pairs cannot be computed during a run since the final reduction order $\to_\mathcal{C}^+$ is not known in advance.

We solve this problem by observing that any reduction order $>$ which is total on ground terms contains the embedding relation $\rhd_{\mathsf{emb}}$ [18, Proposition 2]. Since $CP_>(\mathcal{E}) \subseteq CP_\succ(\mathcal{E})$ whenever $\succ \subseteq >$, the idea is now to over-approximate $CP_>(\mathcal{E}_\omega \cup \mathcal{R}_\omega)$ by $CP_{\rhd_{\mathsf{emb}}}(\mathcal{E}_\omega \cup \mathcal{R}_\omega)$. This motivates the following definition.

Definition 11. *A run γ is* sufficiently fair *if $CP_{\rhd_{\mathsf{emb}}}(\mathcal{E}_\omega \cup \mathcal{R}_\omega) \subseteq \bigcup_i \mathcal{E}_i$.*

It follows that a sufficiently fair run of oKBtt$_{\mathsf{total}}$ is fair with respect to (any total extension of) the final reduction order $\to_\mathcal{C}^+$.

3.1 Correctness and Completeness

With the above considerations, we can carry over the correctness result of ordered completion to the $\mathsf{oKBtt}_{\mathsf{total}}$ setting.

Theorem 12. *If $(\mathcal{E}, \mathcal{R}, \mathcal{C}) \vdash^* (\mathcal{E}_n, \mathcal{R}_n, \mathcal{C}_n)$ is a sufficiently fair, finite $\mathsf{oKBtt}_{\mathsf{total}}$ run with $\mathcal{R} \subseteq \to^+_{\mathcal{C}}$ then $\mathcal{E}_n \cup \mathcal{R}_n$ is ground-complete for $\mathcal{E}$ with respect to any reduction order $>$ total on ground terms that extends $\to^+_{\mathcal{C}_n}$.*

Proof. By Lemma 7, there exists a corresponding oKB run γ' using the reduction order $\to^+_{\mathcal{C}_n}$. Any reduction order $>$ which is total on ground terms contains the embedding relation. Hence $CP_>(\mathcal{E}_n \cup \mathcal{R}_n) \subseteq CP_{\rhd_{\mathsf{emb}}}(\mathcal{E}_n \cup \mathcal{R}_n)$ and as a consequence the sufficiently fair run γ' is also fair with respect to $>$. By correctness of ordered completion, $\mathcal{E}_n \cup \mathcal{R}_n$ is ground-complete for $\mathcal{E}$ with respect to $>$. $\square$

Lemma 8 states that oKBtt is complete in that any oKB run γ can be simulated by an oKBtt run γ'. If γ is fair then also γ' is fair, although it need not be sufficiently fair. Nevertheless, sufficiently fair $\mathsf{oKBtt}_{\mathsf{total}}$ procedures are complete for deriving complete systems if additional equations are considered.

Theorem 13. *Assume $\mathcal{R}$ is a complete system for $\mathcal{E}$ and $\succ$ is a reduction order containing $\mathcal{R}$ which can be extended to a reduction order that is total on ground terms. There exists a sufficiently fair $\mathsf{oKBtt}_{\mathsf{total}}$ run starting from $(\mathcal{E}, \varnothing, \varnothing)$ which produces the result $\mathcal{R}_\omega = \mathcal{R}$ and $\mathcal{E}_\omega = \varnothing$.*

Proof. According to Theorem 3, there exists an oKB run γ producing $\mathcal{R}_\omega = \mathcal{R}$ and $\mathcal{E}_\omega = \varnothing$. By Lemma 8 there is a corresponding oKBtt run $(\mathcal{E}, \varnothing, \varnothing) \vdash^* (\varnothing, \mathcal{R}, \mathcal{C})$. This run can be extended to $(\mathcal{E}, \varnothing, \varnothing) \vdash^* (\varnothing, \mathcal{R}, \mathcal{C}) \vdash^* (\mathcal{E}', \mathcal{R}, \mathcal{C})$ by deducing the remaining equations in $\mathcal{E}' = CP_{\rhd_{\mathsf{emb}}}(\mathcal{E}_\omega \cup \mathcal{R}_\omega) \setminus CP_{\succ}(\mathcal{E}_\omega \cup \mathcal{R}_\omega)$ in order to make it sufficiently fair. Since $\mathcal{R}$ is complete for $\mathcal{E}$, all equations in $\mathcal{E}'$ can be simplified to trivial ones which allows to derive the result $(\varnothing, \mathcal{R}, \mathcal{C})$. $\square$

4 Ordered Multi-Completion with Termination Tools

Ordered multi-completion with termination tools (oMKBtt) simulates multiple oKBtt processes. Similar as in MKBtt, inference steps among these processes are shared. For this purpose, a *process* p is modeled as a bit string in $\mathcal{L}((0+1)^*)$. A set of processes P is called *well-encoded* if there are no processes $p, p' \in P$ such that p is a proper prefix of p'.

Definition 14. *An oMKBtt* node *$\langle s : t, R_0, R_1, E, C_0, C_1\rangle$ consists of a pair of terms $s : t$ (the* data*) and well-encoded sets of processes R_0, R_1, E, C_0, C_1 (the* labels*) such that $R_0 \cup C_0$, $R_1 \cup C_1$ and E are mutually disjoint.*

The set of processes occurring in a node n and a node set $\mathcal{N}$ are denoted by $\mathcal{P}(n)$ and $\mathcal{P}(\mathcal{N})$. The *projection* of a node set $\mathcal{N}$ to a process p is defined below.

$$\mathsf{orewrite}_1 \quad \frac{\mathcal{N} \cup \{\langle s : t, R_0, R_1, E, C_0, C_1\rangle\}}{\begin{array}{l}\mathcal{N} \cup \{\, \langle s : t, R_0 \setminus (R \cup S), R_1, E \setminus R, C_0, C_1\rangle \\ \qquad \langle s : u, R_0 \cap (R \cup S), \varnothing, E \cap R, \varnothing, \varnothing\rangle \\ \qquad \langle l\sigma : r\sigma, \varnothing, \varnothing, \varnothing, S, \varnothing\rangle \,\}\end{array}}$$

if
- $\langle l : r, R, \dots, E'', \dots\rangle \in \mathcal{N}$ and $t \xrightarrow{l\sigma \to r\sigma} u$ where t and l are variants
- $S \subseteq E'' \cap R_0$ such that $C_p(\mathcal{N}) \cup \{l\sigma \to r\sigma\}$ terminates for all $p \in S$
- $((R_0 \cup E) \cap R) \cup S \neq \varnothing$

$$\mathsf{orewrite}_2 \quad \frac{\mathcal{N} \cup \{\langle s : t, R_0, R_1, E, C_0, C_1\rangle\}}{\begin{array}{l}\mathcal{N} \cup \{\, \langle s : t, R_0 \setminus (R \cup S), R_1 \setminus (R \cup S), E \setminus (R \cup S), C_0, C_1\rangle \\ \qquad \langle s : u, R_0 \cap (R \cup S), \varnothing, (E \cup R_1) \cap (R \cup S), \varnothing, \varnothing\rangle \\ \qquad \langle l\sigma : r\sigma, \varnothing, \varnothing, \varnothing, S, \varnothing\rangle \,\}\end{array}}$$

if
- $\langle l : r, R, \dots, E'', \dots\rangle \in \mathcal{N}$ and $t \xrightarrow{l\sigma \to r\sigma} u$ where $t \rhd l$
- $S \subseteq E'' \cap (R_0 \cup R_1 \cup E)$ such that $C_p(\mathcal{N}) \cup \{l\sigma \to r\sigma\}$ terminates for all $p \in S$
- $(R_0 \cup R_1 \cup E) \cap (R \cup S) \neq \varnothing$

$$\mathsf{odeduce} \quad \frac{\mathcal{N}}{\mathcal{N} \cup \{\, \langle s : t, \varnothing, \varnothing, (R \cup E) \cap (R' \cup E'), \varnothing, \varnothing\rangle \,\}}$$

if
- $\langle l : r, R, \dots, E, \dots\rangle, \langle l' : r', R', \dots, E', \dots\rangle \in \mathcal{N}$
- $s \xleftarrow{l \to r} u \xrightarrow{l' \to r'} t$ and $(R \cup E) \cap (R' \cup E') \neq \varnothing$

Fig. 5. The orewrite and odeduce inference rules in oMKBtt

Definition 15. *Given a node $n = \langle s : t, R_0, R_1, E, C_0, C_1\rangle$ and a process p, let P_p denote the set of prefixes of p, and set*

$$E_p(n) = \begin{cases} \{s \approx t\} & \text{if } P_p \cap E \neq \varnothing \\ \varnothing & \text{otherwise} \end{cases} \qquad R_p(n) = \begin{cases} \{s \to t\} & \text{if } P_p \cap R_0 \neq \varnothing \\ \{t \to s\} & \text{if } P_p \cap R_1 \neq \varnothing \\ \varnothing & \text{otherwise} \end{cases}$$

The set $C_p(n)$ is defined analogous to $R_p(n)$. Furthermore, we define $E_p(\mathcal{N}) = \bigcup_{n \in \mathcal{N}} E_p(n)$, $R_p(\mathcal{N}) = \bigcup_{n \in \mathcal{N}} R_p(n)$ and $C_p(\mathcal{N}) = \bigcup_{n \in \mathcal{N}} C_p(n)$.

Note that the above projections are well-defined if all process sets in $\mathcal{N}$ are well-encoded. The inference system oMKBtt works on sets of nodes $\mathcal{N}$ and consists of the rules given in Fig. 5 together with orient, delete and (optionally) subsume and gc as defined for MKBtt [12]. Note that all inference rules preserve well-encodedness and the disjointness condition on labels. Given an oMKBtt run $\mathcal{N}_0 \vdash \mathcal{N}_1 \vdash \mathcal{N}_2 \vdash \dots$, the set $\mathcal{N}_\omega = \bigcup_i \bigcap_{j>i} \mathcal{N}_j$ collects *persisting nodes*. For a set of equations $\mathcal{E}$, the *initial* node set $\mathcal{N}_\mathcal{E}$ consists of all nodes $\langle s : t, \varnothing, \varnothing, \{\epsilon\}, \varnothing, \varnothing\rangle$ such that $s \approx t$ belongs to $\mathcal{E}$.

Example 16. We illustrate oMKBtt on the equations of Example 5. We start with the initial node set

$$\mathcal{N}_0 = \left\{ \begin{array}{ll} \langle \mathsf{g}(\mathsf{f}(x,\mathsf{b})) : \mathsf{a}, \varnothing, \varnothing, \{\epsilon\}, \varnothing, \varnothing \rangle & (1) \\ \langle \mathsf{f}(\mathsf{g}(x),y) : \mathsf{f}(x,\mathsf{g}(y)), \varnothing, \varnothing, \{\epsilon\}, \varnothing, \varnothing \rangle & (2) \end{array} \right\}$$

In the first step one may orient node (1), where only the direction from left to right is possible. Concerning the second node, both constraint systems

$$\begin{array}{l} C_0 = \{\ \mathsf{g}(\mathsf{f}(x,\mathsf{b})) \to \mathsf{a},\ \mathsf{f}(\mathsf{g}(x),y) \to \mathsf{f}(x,\mathsf{g}(y))\ \} \\ C_1 = \{\ \mathsf{g}(\mathsf{f}(x,\mathsf{b})) \to \mathsf{a},\ \mathsf{f}(x,\mathsf{g}(y)) \to \mathsf{f}(\mathsf{g}(x),y)\ \} \end{array}$$

terminate, the first using LPO with precedence $\mathsf{f} > \mathsf{g} > \mathsf{a}$ and the second with the polynomial interpretation from Example 5. Hence the process ϵ is split:

$$\begin{array}{ll} \mathsf{g}(\mathsf{f}(x,\mathsf{b})) : \mathsf{a}, \{0,1\}, \varnothing, \varnothing, \{0,1\}, \varnothing \rangle & (1) \\ \langle \mathsf{f}(\mathsf{g}(x),y) : \mathsf{f}(x,\mathsf{g}(y)), \{0\}, \{1\}, \varnothing, \{0\}, \{1\} \rangle & (2) \end{array}$$

Starting from the overlap $\mathsf{g}(\mathsf{f}(x,\mathsf{g}(\mathsf{b}))) \leftarrow \mathsf{g}(\mathsf{f}(\mathsf{g}(x),\mathsf{b})) \to \mathsf{a}$ between nodes (1) and (2), if process 0 is advanced further then infinitely many nodes of the form $\langle \mathsf{g}(\mathsf{f}(x,\mathsf{g}^n(\mathsf{b}))) : \mathsf{a}, \{0\}, \varnothing, \varnothing, \{0\}, \varnothing \rangle$ are generated. On the other hand, similarly as in Example 5, one can deduce $\mathsf{f}(\mathsf{g}(x),\mathsf{f}(y,\mathsf{b})) \approx \mathsf{f}(x,\mathsf{c})$, orient the corresponding new node (3) and add the critical pair (4) between nodes (2) and (3). It remains to consider the overlaps between node (4) and itself to obtain

$$\begin{array}{ll} \langle \mathsf{f}(\mathsf{g}(x),\mathsf{f}(y,\mathsf{b})) : \mathsf{f}(x,\mathsf{c}), \{1\}, \varnothing, \varnothing, \{1\}, \varnothing \rangle & (3) \\ \langle \mathsf{f}(\mathsf{f}(x,\mathsf{b}),\mathsf{a}) : \mathsf{f}(\mathsf{a},\mathsf{f}(y,\mathsf{b})), \varnothing, \varnothing, \{1\}, \varnothing, \varnothing \rangle & (4) \\ \langle \mathsf{f}(\mathsf{f}(x,\mathsf{b}),\mathsf{a}) : \mathsf{f}(\mathsf{f}(y,\mathsf{b}),\mathsf{a}), \varnothing, \varnothing, \{1\}, \varnothing, \varnothing \rangle & (5) \\ \langle \mathsf{f}(\mathsf{a},\mathsf{f}(x,\mathsf{b})) : \mathsf{f}(\mathsf{a},\mathsf{f}(y,\mathsf{b})), \varnothing, \varnothing, \{1\}, \varnothing, \varnothing \rangle & (6) \end{array}$$

at which point process 1 is saturated. Applying the projections $E_1(\mathcal{N})$ and $R_1(\mathcal{N})$ to the current node set $\mathcal{N} = \{(1), \dots, (6), \dots\}$ yields the ground-complete system $(\mathcal{E}, \mathcal{R})$ derived in Example 5.

Intuitively, $\mathsf{orewrite}_1$ simulates the oKBtt inferences compose, simplify and $\mathsf{compose}_2$ whenever t and l are variants while $\mathsf{orewrite}_2$ models these inference steps together with collapse, $\mathsf{simplify}_2$ and $\mathsf{collapse}_2$ if $t \rhd l$. To express this relationship formally in Lemmata 18 and 19 below, we need notation to refer to process splitting.

Definition 17. *If an oMKBtt inference step $\mathcal{N} \vdash \mathcal{N}'$ applies orient, then the set of processes S which were divided into two child processes is called the step's* split set. *In the other cases, the split set is empty. For a step with split set S and $p' \in \mathcal{P}(\mathcal{N}')$, the predecessor of p' is defined as*

$$\mathrm{pred}_S(p') = \begin{cases} p & \textit{if } p' = p0 \textit{ or } p' = p1 \textit{ for some } p \in S \\ p' & \textit{otherwise} \end{cases}$$

The longish but straightforward proofs of the following lemmata can be found in the report version. In Lemma 18, $\vdash^{=}$ denotes the reflexive closure of the oKBtt inference relation $\vdash$.

Lemma 18. *If $\mathcal{N} \vdash \mathcal{N}'$ is an oMKBtt step with split set S then*

$$(E_p(\mathcal{N}), R_p(\mathcal{N}), C_p(\mathcal{N})) \vdash^{=} (E_{p'}(\mathcal{N}'), R_{p'}(\mathcal{N}'), C_{p'}(\mathcal{N}'))$$

is a valid oKBtt inference for all $p' \in \mathcal{P}(\mathcal{N}')$, where $p = pred_S(p')$. Moreover, the strict part $\vdash$ holds for at least one $p' \in \mathcal{P}(\mathcal{N}')$.

Lemma 19. *Consider an oKBtt inference step $(\mathcal{E}, \mathcal{R}, \mathcal{C}) \vdash (\mathcal{E}', \mathcal{R}', \mathcal{C}')$. Assume there exist a node set $\mathcal{N}$ and a process p such that $\mathcal{E} = E_p(\mathcal{N})$, $\mathcal{R} = R_p(\mathcal{N})$ and $\mathcal{C} = C_p(\mathcal{N})$. Then there are a node set $\mathcal{N}'$, an inference step $\mathcal{N} \vdash \mathcal{N}'$ with split set S, and a process $p' \in \mathcal{P}(\mathcal{N}')$ such that $p = pred_S(p')$, $\mathcal{E}' = E_{p'}(\mathcal{N}')$, $\mathcal{R}' = R_{p'}(\mathcal{N}')$ and $\mathcal{C}' = C_{p'}(\mathcal{N}')$.*

Projecting an oMKBtt run γ of length α to a process $p \in \mathcal{P}(\mathcal{N}_\alpha)$ thus yields a valid oKBtt run, which is denoted by γ_p in the sequel. Before correctness can be addressed, we adapt the definition of (sufficient) fairness and note that oMKBtt is sound.

Definition 20. *A run γ of length α is* sufficiently fair *if either $\alpha < \omega$ and γ_p is sufficiently fair for at least one process $p \in \mathcal{P}(\mathcal{N}_\alpha)$, or $\alpha = \omega$ and γ_p is sufficiently fair for all $p \in \mathcal{P}(\mathcal{N}_\alpha)$.*

Lemma 21. *Consider an oMKBtt step $\mathcal{N} \vdash \mathcal{N}'$ with split set S and a process $q \in \mathcal{P}(\mathcal{N}')$ with predecessor $p = \mathrm{pred}_S(q)$. For $\mathcal{E} = E_p(\mathcal{N})$, $\mathcal{R} = R_p(\mathcal{N})$ and $\mathcal{E}' = E_q(\mathcal{N}')$, $\mathcal{R}' = R_q(\mathcal{N}')$ the relations $\leftrightarrow^*_{\mathcal{E} \cup \mathcal{R}}$ and $\leftrightarrow^*_{\mathcal{E}' \cup \mathcal{R}'}$ coincide.*

Similar to the oKBtt case, an *oMKBtt$_{\mathcal{P}}$ procedure* refers to a program that takes a set of equations $\mathcal{E}$ as input and uses the inference rules of oMKBtt to generate a derivation starting from $\mathcal{N}_{\mathcal{E}}$, where termination checks are performed with respect to a termination strategy $\mathcal{P}$. An oMKBtt$_{\mathsf{total}}$ procedure is any oMKBtt$_{\mathcal{P}}$ procedure where $\mathcal{P}$ guarantees total termination of the checked systems.

Using the simulation properties expressed in Lemmata 18 and 19, correctness and completeness easily follow from the corresponding results for oKBtt.

Theorem 22. *Let $\mathcal{N}_0 = \mathcal{N}_{\mathcal{E}}$ be the initial node set for $\mathcal{E}$ and let $\mathcal{N}_0 \vdash^* \mathcal{N}_n$ be a finite oMKBtt$_{\mathsf{total}}$ run. If $\mathcal{N}_0 \vdash^* \mathcal{N}_n$ is sufficiently fair for $p \in \mathcal{P}(\mathcal{N}_n)$ then $E_p(\mathcal{N}_n) \cup R_p(\mathcal{N}_n)$ is ground-complete for a reduction order $>$ that is total on ground terms and extends $\rightarrow^+_{\mathcal{C}}$, where $\mathcal{C} = C_p(\mathcal{N}_n)$.*

Theorem 23. *Assume $\mathcal{R}$ is a complete rewrite system for $\mathcal{E}$ and $\succ$ is a reduction order containing $\mathcal{R}$ which can be extended to a total reduction order. Then there exists a sufficiently fair and simplifying oMKBtt$_{\mathsf{total}}$ run $\mathcal{N}_{\mathcal{E}} \vdash^* \mathcal{N}_\alpha$ such that some process $p \in \mathcal{P}(\mathcal{N}_\alpha)$ satisfies $R_p(\mathcal{N}_\alpha) = \mathcal{R}$ and $E_p(\mathcal{N}_\alpha) = \varnothing$.*

5 Theorem Proving with oMKBtt

The use of ordered completion for refutational theorem proving proposed in [2] can easily be adapted to the oMKBtt setting. For a term s, we write $\hat{s}$ to denote the term where each variable is replaced by its corresponding Skolem constant. In the sequel, given equations $\mathcal{E}$ and a goal $s \approx t$, let $\mathcal{N}_{\mathcal{E}}^{s\approx t}$ denote the set

$$\mathcal{N}_{\mathcal{E}} \cup \{\, \langle \mathsf{equal}(x,x) : \mathsf{true}, \varnothing, \varnothing, \{\epsilon\}, \varnothing, \varnothing \rangle,\ \langle \mathsf{equal}(\hat{s},\hat{t}) : \mathsf{false}, \varnothing, \varnothing, \{\epsilon\}, \varnothing, \varnothing \rangle \,\}$$

As the following results show, theorem proving with oMKBtt is sound, independent of the applied termination techniques. To obtain completeness we restrict to $\mathsf{oMKBtt}_{\mathsf{total}}$ procedures.

Lemma 24. *If an oMKBtt run starting from $\mathcal{N}_0 = \mathcal{N}_{\mathcal{E}}^{s\approx t}$ generates a node $\langle \mathsf{true} : \mathsf{false}, \ldots, E, \ldots \rangle$ in some set $\mathcal{N}_i$ and $E \neq \varnothing$ then $s \approx t$ is valid in $\mathcal{E}$.*

Lemma 25. *If $s \approx t$ is valid in $\mathcal{E}$ then any sufficiently fair $\mathsf{oMKBtt}_{\mathsf{total}}$ run $\mathcal{N}_0 \vdash \mathcal{N}_1 \vdash \cdots \vdash \mathcal{N}_\alpha$ starting from $\mathcal{N}_0 = \mathcal{N}_{\mathcal{E}}^{s\approx t}$ generates a node $\langle \mathsf{true} : \mathsf{false}, \ldots, E, \ldots \rangle$ in some set $\mathcal{N}_i$ such that $E \neq \varnothing$.*

Proof. Since the run is sufficiently fair it is sufficiently fair for some process p. By Lemma 18 there is a sufficiently fair oKBtt run

$$(E_p(\mathcal{N}_0), R_p(\mathcal{N}_0), C_p(\mathcal{N}_0)) \vdash^* (E_p(\mathcal{N}_\alpha), R_p(\mathcal{N}_\alpha), C_p(\mathcal{N}_\alpha))$$

According to Lemma 7, there is a corresponding fair oKB run using the reduction order $\to_{\mathcal{C}}^{+}$, where $\mathcal{C} = C_p(\mathcal{N}_\alpha)$. Moreover, $\to_{\mathcal{C}}^{+}$ can be extended to a reduction order $>$ that is total on ground terms. By [2, Theorem 3], such a fair ordered completion run starting from $\mathcal{E}_0 = \mathcal{E} \cup \{\mathsf{equal}(x,x) \approx \mathsf{true},\ \mathsf{equal}(\hat{s},\hat{t}) \approx \mathsf{false}\}$ will have the contradictory statement $\mathsf{true} \approx \mathsf{false}$ in some set $\mathcal{E}_i \cup \mathcal{R}_i$, so there is a node $\langle \mathsf{true} : \mathsf{false}, \ldots \rangle$ in some $\mathcal{N}_i$. □

6 Implementation

This section briefly outlines our tool $\mathsf{omkb}_{\mathsf{TT}}$. Extending the existing $\mathsf{mkb}_{\mathsf{TT}}$ implementation [11,17], it is implemented in OCaml in about 10.000 lines of code. To check constraint systems for termination, $\mathsf{omkb}_{\mathsf{TT}}$ either uses an external tool which is compatible with a minimal interface or interfaces $\mathsf{T_TT_2}$ [8] internally.

Our tool $\mathsf{omkb}_{\mathsf{TT}}$ is equipped with a simple command-line interface. The input system is expected in the TPTP-3 [14] format. Among other options, users can fix the global time limit and the time limit for a termination call, specify either an external executable for termination checks or configure how $\mathsf{T_TT_2}$ should be used internally, and control which indexing technique, node selection strategy or goal representation to use. For further details we refer to the website and [17].

In the original presentation of completion-based theorem proving [2], given a goal $s \approx t$ the equations $\mathsf{equal}(x,x) \approx \mathsf{true}$ and $\mathsf{equal}(\hat{s},\hat{t}) \approx \mathsf{false}$ are added.

Table 1. Completing theories associated with TPTP UEQ systems

	ttt2total		kbo		lpo		poly		mpo		E	
	(1)	(2)	(1)	(2)	(1)	(2)	(1)	(2)	(1)	(2)	(1)	(2)
et	37	22.8	38	23.5	23	22.1	35	34.1	37	29.0	10	0.04
dt	45	24.5	55	17.4	24	156.5	44	11.5	45	10.5	35	0.06

Waldmeister uses a different representation of the goal [5]. The reducts of $\hat{s}$ and $\hat{t}$ are kept in two sets R_s and R_t. Whenever a term in R_s or R_t can be reduced, the new reducts are added to R_s or R_t, respectively. The goal is proven as soon as $R_s \cap R_t$ is non-empty. This approach is supported in $\mathsf{omkb_{TT}}$ as well. Sets R_s and R_t of pairs (u, P) where u is a term and P the set of processes for which this reduct was derived are maintained. The goal is proven if there exists a term u such that $(u, P) \in R_s$, $(u, P') \in R_t$ and $P \cap P'$ is non-empty.

7 Experimental Results

This section summarizes experimental results obtained with $\mathsf{omkb_{TT}}$. All tests were run on a single core of a server equipped with eight dual-core AMD Opteron® processors 885 running at a clock rate of 2.6GHz and 64GB of main memory.

In all of the following tests $\mathsf{omkb_{TT}}$ internally interfaces $\mathsf{T_TT_2}$ for termination checks. To compare the applicability of different termination techniques, different $\mathsf{T_TT_2}$ strategies were used: kbo, lpo and mpo denote the well-known reduction orders and poly refers to linear polynomial interpretations with coefficients in $\{0, \ldots, 7\}$. The strategy where all these techniques performed in parallel are applied iteratively is denoted by ttt2total. The strategy ttt2fast involves dependency pairs so total termination is not ensured. It is therefore only used for theorem proving, which is sound according to Lemma 24, although incomplete because completeness of refutational theorem proving holds only for totalizable reduction orders [2].

Examples stem from the unit equality division of TPTP 3.6.0 [14]. The test set e consists of 215 problems rated *easy*, d contains 565 problems classified as *difficult*. The sets et and dt consist of the 204 and 563 different theories associated with these problems. Table 1 shows ordered completion results obtained with $\mathsf{omkb_{TT}}$. The columns list (1) the number of successes, (2) the average time for a successful run in seconds (given a timeout of 600 seconds), and (3) the percentage of time spent on termination checks. In order to compare with other ordered completion tools, we ran E [13] on the same test set in *auto* mode, such that it heuristically determines the reduction order to use.[2] As an example, $\mathsf{omkb_{TT}}$ using ttt2total completes the theory underlying problem GRP447-1 from TPTP within 3 seconds, while neither E nor $\mathsf{mkb_{TT}}$ produce a solution within 1 hour.

[2] We did not use Waldmeister here since, according to personal communication with the developers, its *auto* mode should not be used for ordered completion.

Table 2. Performance of oMKBtt on TPTP UEQ problems

	ttt2total			kbo			lpo			poly			ttt2fast		
	(1)	(2)	(3)	(1)	(2)	(3)	(1)	(2)	(3)	(1)	(2)	(3)	(1)	(2)	(3)
e	149	43.9	82	163	16.6	8	164	24.3	14	143	59.1	90	138	49.9	80
d	116	66.0	64	148	64.8	4	152	50.6	6	109	95.7	79	121	55.0	17

The over-approximation of extended critical pairs with the embedding relation, i.e., the use of $CP_{\rhd_{\mathsf{emb}}}$ instead of $CP_{\varnothing}$ allows for a performance gain of about 28%.

Table 2 shows theorem proving results obtained with omkb$_{\mathsf{TT}}$. Both Waldmeister and E solve about 200 problems in e and more than 400 of the d set.[3] Although the considered termination strategies are incomparable in power, kbo handles the most problems, both for ordered completion and theorem proving. The reason for that is that little time is spent on termination checks, as can be seen from Table 2. Although the combination of multiple techniques in ttt2total is theoretically more powerful than each technique separately, the larger number of processes (25% more than kbo and twice as much as in lpo or poly) decreases performance and causes more timeouts. The evaluation of different combinations of termination strategies, such as the incremental use of polynomial interpretations, is subject to future work.

We compared the simple approach where the goal is represented as two nodes with the Waldmeister-like approach described in Section 6. According to our results, the latter is faster and therefore able to prove about 3% more examples. However, in some cases the simple approach succeeds whereas the Waldmeister-like approach fails due to a "combinatorial explosion".

8 Conclusion

We outlined how termination tools can replace a fixed term order in ordered completion and completion-based theorem proving. This approach can also be combined with multi-completion. Besides the advantage that no reduction order has to be provided as input, this novel approach allows to derive ground-complete systems for problems that are not compatible with standard orders such as LPO and KBO. Hence our tool omkb$_{\mathsf{TT}}$ can deal with input systems that cannot be solved with other tools, to the best of our knowledge.

In contrast to standard completion, in the case of ordered completion the reduction order implicitly developed in the inference sequence needs to be extensible to a reduction order which is complete for the theory. Hence omkb$_{\mathsf{TT}}$ restricts to termination techniques which entail *total termination*. It is subject to further research whether the existence of a suitable order $>$ can be guaranteed by other means such that applicable termination techniques are less restricted.

[3] It should be noted that omkb$_{\mathsf{TT}}$ cannot (yet) cope with existentially quantified goals. There are 16 such problems in e and 61 in d.

Acknowledgements. The comments of the anonymous referees helped to improve the paper.

References

1. Bachmair, L., Dershowitz, N.: Equational inference, canonical proofs, and proof orderings. Journal of the ACM 41(2), 236–276 (1994)
2. Bachmair, L., Dershowitz, N., Plaisted, D.A.: Completion without failure. In: Aït-Kaci, H., Nivat, M. (eds.) Resolution of Equations in Algebraic Structures. Rewriting Techniques of Progress in Theoretical Computer Science, vol. 2, pp. 1–30. Academic Press, London (1989)
3. Bofill, M., Godoy, G., Nieuwenhuis, R., Rubio, A.: Paramodulation and Knuth–Bendix completion with nontotal and nonmonotonic orderings. Journal of Automated Reasoning 30(1), 99–120 (2003)
4. Dershowitz, N.: Orderings for term rewriting systems. Theoretical Computer Science 17(3), 279–301 (1982)
5. Hillenbrand, T., Löchner, B.: The next Waldmeister loop. In: Voronkov, A. (ed.) CADE 2002. LNCS (LNAI), vol. 2392, pp. 486–500. Springer, Heidelberg (2002)
6. Kamin, S., Lévy, J.J.: Two generalizations of the recursive path ordering. University of Illinois (1980) (unpublished manuscript)
7. Knuth, D.E., Bendix, P.: Simple word problems in universal algebras. In: Leech, J. (ed.) Computational Problems in Abstract Algebra, pp. 263–297. Pergamon Press, Oxford (1970)
8. Korp, M., Sternagel, C., Zankl, H., Middeldorp, A.: Tyrolean termination tool 2. In: Treinen, R. (ed.) RTA 2009. LNCS, vol. 5595, pp. 295–304. Springer, Heidelberg (2009)
9. Kurihara, M., Kondo, H.: Completion for multiple reduction orderings. Journal of Automated Reasoning 23(1), 25–42 (1999)
10. Lankford, D.: On proving term rewrite systems are noetherian. Technical Report MTP-3, Louisiana Technical University (1979)
11. Sato, H., Winkler, S., Kurihara, M., Middeldorp, A.: Multi-completion with termination tools (System description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 306–312. Springer, Heidelberg (2008)
12. Sato, H., Winkler, S., Kurihara, M., Middeldorp, A.: Constraint-based multi-completion procedures for term rewriting systems. IEICE Transactions on Electronics, Information and Communication Engineers E92-D(2), 220–234 (2009)
13. Schulz, S.: The E Equational Theorem Prover (2009), `http://www.eprover.org`
14. Sutcliffe, G.: The TPTP problem library and associated infrastructure. Journal of Automated Reasoning 43(4), 337–362 (2009)
15. Terese: Term Rewriting Systems. Cambridge Tracts in Theoretical Computer Science, vol. 55. Cambridge University Press, Cambridge (2003)
16. Wehrman, I., Stump, A., Westbrook, E.M.: Slothrop: Knuth-Bendix completion with a modern termination checker. In: Pfenning, F. (ed.) RTA 2006. LNCS, vol. 4098, pp. 287–296. Springer, Heidelberg (2006)
17. Winkler, S., Sato, H., Middeldorp, A., Kurihara, M.: Optimizing mkbTT (System description). In: Lynch, C. (ed.) Proc. 21st RTA. LIPIcs (to appear, 2010)
18. Zantema, H.: Total termination of term rewriting is undecidable. Journal of Symbolic Computation 20(1), 43–60 (1995)

Author Index