Edmund M. Clarke
Andrei Voronkov (Eds.)

LNAI 6355

Logic for Programming, Artificial Intelligence, and Reasoning

16th International Conference, LPAR-16
Dakar, Senegal, April/May 2010
Revised Selected Papers

Springer

Lecture Notes in Artificial Intelligence 6355

Edited by R. Goebel, J. Siekmann, and W. Wahlster

Subseries of Lecture Notes in Computer Science

Edmund M. Clarke Andrei Voronkov (Eds.)

Logic for Programming, Artificial Intelligence, and Reasoning

16th International Conference, LPAR-16
Dakar, Senegal, April 25–May 1, 2010
Revised Selected Papers

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

Edmund M. Clarke
Carnegie Mellon University
Computer Science Department
5000 Forbes Avenue, Pittsburgh, PA 15213-3891, USA
E-mail: emc@cs.cmu.edu

Andrei Voronkov
University of Manchester
School of Computer Science
Kilburn Building, Oxford Road, Manchester, M13 9PL, UK
E-mail: andrei.voronkov@manchester.ac.uk

Library of Congress Control Number: 2010939948

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

LNCS Sublibrary: SL 7 – Artificial Intelligence

ISSN 0302-9743
ISBN-10 3-642-17510-4 Springer Berlin Heidelberg New York
ISBN-13 978-3-642-17510-7 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

The 16th LPAR event was held in Africa for the very first time, signalling a bright future for a conference with a glowing history. For many years it was a goal of the steering committee to hold LPAR in Africa and the enthusiasm for our presence far exceeded our expectations. With the help of local organiser Waly Faye, LPAR integrated itself into the surrounding culture and atmosphere, allowing participants to enjoy a collage of music, dancing, food, and logic.

Organisational issues caused LPAR 16, originally intended to be held in 2009, to be delayed to 2010. Despite this, LPAR 16 received 47 submissions. Each submission was reviewed by at least 4, and on average 4.1, programme committee members. The committee members decided to accept 27 regular papers and 9 short papers. They deliberated electronically via the EasyChair system, which continued to provide a platform for smoothly carrying out all aspects of the program selection and finalization and conference registration. It has been a tradition of LPAR to invite some of the most influential researchers in its focus area to discuss their work and their vision for the field. This year's distinguished speakers were Geoff Sutcliffe (University of Miami, USA) and Michel Parigot (PPS, France). This volume contains the revised versions of the accepted full papers as well as the full text of Geoff Sutcliffe's talk.

This conference would not have been possible without the hard work of the many people who relentlessly handled the local arrangements, especially Wale Faye, who offered us the full hospitality of his country, village and extended family. We are most grateful to the 32 members of the Program Committee, who did an excellent job in handling the submissions, and the additional reviewers, who assisted them in their evaluations. We greatly appreciate the generous support of our sponsors, the Office of Naval Research and Microsoft Research. We are also especially grateful to Michael Gabbay for organising, drafting and compiling these proceedings. Finally we are grateful to the authors, the invited speakers and the attendees who made this conference an enjoyable and fruitful event.

June 2010

Edmund Clarke
Andrei Voronkov

Conference Organisation

Programme Chairs

Edmund Clarke
Andrei Voronkov

Programme Committee

Rajeev Alur
Matthias Baaz
Peter Baumgartner
Armin Biere
Nikolaj Bjorner
Iliano Cervesato
Agata Ciabattoni
Hubert Comon-Lundh
Nachum Dershowitz
Jürgen Giesl
Guillem Godoy
Georg Gottlob
Jean Goubault-Larrecq
Reiner Haehnle
Claude Kirchner
Michael Kohlhase
Konstantin Korovin
Laura Kovacs
Orna Kupferman
Leonid Libkin
Aart Middeldorp
Luke Ong
Frank Pfenning
Andreas Podelski
Andrey Rybalchenko
Helmut Seidl
Geoff Sutcliffe
Ashish Tiwari
Toby Walsh
Christoph Weidenbach

External Reviewers

Wolfgang Ahrendt
Martin Avanzini
Andreas Bauer
Ralph Becket
Eli Ben-Sasson
Paul Brauner
Angelo Brillout
Christopher Broadbent
Robert Brummayer
Roberto Bruttomesso
Richard Bubel
Guillaume Burel
Elie Bursztein
Silvia Calegari
Adrian Craciun
Hans de Nivelle
Marc Denecker
Manfred Droste
Stephan Falke
Oliver Fasching
Germain Faure
Arnaud Fietzke
Bernd Finkbeiner
Alain Finkel
Guido Fiorino
Carsten Fuhs
Deepak Garg
Stéphane Gaubert

Samir Genaim
Valentin Goranko
Torsten Grust
Stefan Hetzl
Matthias Horbach
Fulya Horozal
Radu Iosif
Tomi Janhunen
Predrag Janicic
Christian Kern
Michel Ludwig
Ines Lynce
Michael Maher
William McCune
Chris Mears
Georg Moser
Nina Narodytska
Robert Nieuwenhuis
Albert Oliveras
Carsten Otto
Bruno Woltzenlogel Paleo
Luca Paolini
Nicolas Peltier
Florian Rabe
Jason Reed
Andreas Reuss
Zdenek Sawa
Peter Schneider-Kamp
Lutz Schröder
Dirk Seifert
Olivier Serre
Niklas Sörensson
Christoph Sticksel
Lutz Strassburger
Martin Suda
Kazushige Terui
Rene Thiemann
Michael Vanden Boom
Eelco Visser
Bogdan Warinschi
Markus Wedler
Martin Weichert
Benjamin Weiss
Patrick Wischnewski
Harald Zankl

Table of Contents

The TPTP World – Infrastructure for Automated Reasoning

Geoff Sutcliffe

University of Miami, USA

Abstract. The TPTP World is a well known and established infrastructure that supports research, development, and deployment of Automated Theorem Proving (ATP) systems for classical logics. The data, standards, and services provided by the TPTP World have made it increasingly easy to build, test, and apply ATP technology. This paper reviews the core features of the TPTP World, describes key service components of the TPTP World, presents some successful applications, and gives an overview of the most recent developments.

1 Introduction

Automated Theorem Proving (ATP) is concerned with the development and use of computer programs that automate sound reasoning: the derivation of conclusions that follow inevitably from facts. The dual discipline, automated model finding, develops computer programs that establish that a set of statements is consistent, and in this work we consider automated model finding to be part of ATP. These capabilities lie at the heart of many important computational tasks, e.g., formal methods for software and hardware design and verification, the analysis of network security protocols, solving hard problems in mathematics, and inference for the semantic web. High performance ATP systems (for logics supported in the TPTP World) include E/EP [16], LEO-II [1], Paradox [3], SPASS [33], Vampire [14], and Waldmeister [7].

The TPTP World is a well known and established infrastructure that supports research, development, and deployment of Automated Theorem Proving (ATP) systems for classical logics. The TPTP World includes the TPTP problem library, the TSTP solution library, standards for writing ATP problems and reporting ATP solutions, tools for processing ATP problems and solutions, and harnesses for controlling the execution of ATP systems and tools. The TPTP World infrastructure has been deployed in a range of applications, in both academia and industry. Section 2 of this paper reviews the core of the TPTP World, Section 3 describes key service components of the TPTP World, Section 4 presents some applications, and Section 5 gives an overview of the most recent developments in the TPTP World. Section 6 concludes.

E.M. Clarke and A. Voronkov (Eds.): LPAR-16, LNAI 6355, pp. 1–12, 2010.

2 The TPTP World Core Infrastructure

The Thousands of Problems for Theorem Provers (TPTP) problem library [20] is the de facto standard set of test problems for Automated Theorem Proving (ATP) systems for classical logics.[1] It is the original core component of the TPTP World, and is commonly referred to as "the TPTP". The TPTP problem library supplies the ATP community with a comprehensive library of the test problems that are available today, in order to provide an overview and a simple, unambiguous reference mechanism, to support the testing and evaluation of ATP systems, and to help ensure that performance results accurately reflect capabilities of the ATP systems being considered. The current TPTP (v4.0.1) has forty-one domains, in the fields of logic, mathematics, computer science, science and engineering, and social sciences. Each TPTP problem file has a header section that contains information for the user, and the logical formulae are wrapped with annotations that provide a unique name for each formula in the problem, a user role (axiom, conjecture, etc.), and auxiliary user information. The logical formulae are written in the TPTP language (described below), which has a consistent and easily understood notation. Since its first release in 1993, many researchers have used the TPTP as an appropriate and convenient basis for ATP system evaluation. Over the years the TPTP has also increasingly been used as a conduit for ATP users to provide samples of their problems to ATP system developers – users have found that contributing samples of their problems to the TPTP exposes the problems to the developers, who then improve their systems' performances on the problems, which completes a cycle to provide the users with more effective tools.

One of the keys to the success of the TPTP World is the consistent use of the TPTP language for writing problems and solutions [24], which enables convenient communication between different systems and researchers. The language shares many features with Prolog, a language that is widely known in the ATP community. Indeed, with a few operator definitions, units of TPTP data can be read in Prolog using a single `read`/1 call, and written with a single `writeq`/1 call. A principal goal of the development of the TPTP language grammar was to make it easy to translate the BNF into `lex`/`yacc`/`flex`/`bison` input, so that construction of parsers (in languages other than Prolog) can be a reasonably easy task [32]. The TPTP World services described in Section 3 naturally process data written in the TPTP language. Parsers written in C and Java, and the `lex`/`yacc`/`flex`/`bison` input files, are part of the TPTP World. Many ATP system developers and users have adopted the TPTP language.

In order to precisely specify what is known or has been established about a set of formulae, the TPTP World provides the SZS ontologies [19]. These ontologies provide status and dataform values to describe logical data. For example, a TPTP problem might be tagged as a `Theorem`, a model finder might report that a set of formulae is `Satisfiable`, and a parser might report that a formula contains a `SyntaxError`. The SZS standard also recommends the precise way

[1] Available at `http://www.tptp.org`

in which the ontology values should be presented, in order to facilitate easy processing.

The Thousands of Solutions from Theorem Provers (TSTP) solution library, the "flip side" of the TPTP, is a corpus of ATP systems' solutions to TPTP problems.[2] A major use of the TSTP is for ATP system developers to examine solutions to problems, and thus understand how they can be solved, leading to improvements to their own systems. At the time of writing this paper, the TSTP contained the results of running 54 ATP systems and system variants on all the problems in the TPTP that they can, in principle, attempt to solve (therefore, e.g., finite model finding systems are not run on problems that are known to be unsatisfiable). This has produced over 155000 files for solved problems, of which almost 100000 contain explicit proofs or models (rather than only an assurance of a solution). The first section of each TSTP solution file is a header that contains information about the TPTP problem, information about the ATP system, characteristics of the computer used, the SZS status and output dataform from the system, and statistics about the solution. The second section of each TSTP solution file contains the annotated formulae that make up the solution. A key feature of the TSTP is that solutions from many of the ATP systems are written in the TPTP language - the same language as used for TPTP problems.

An important feature of the TPTP is the problem ratings [28]. The ratings provide a well-defined measure of how difficult the problems are for ATP systems, and how effective the ATP systems are for different types of problems. For rating, the TPTP problems are divided into Specialist Problem Classes (SPCs), and the TSTP files for each SPC are analyzed. The performance of systems whose set of solved problems is not a subset of that of any other system is used to rate the problems. The fraction of such systems that fail on a problem is the difficulty rating for a problem: problems that are solved by all/some/none of the systems get ratings of 0.00/0.01-0.99/1.00, and are referred to as easy/difficult/hard problems respectively. Over time, decreasing ratings for individual problems provide an indication of progress in the field [23]. The analysis done for problem ratings also provides ratings for ATP systems, for each SPC: the rating of a system is the fraction of the difficult problems that it solves.

3 TPTP World Services

The TPTP World includes tools, programming libraries, and online services that are used to support the application and deployment of ATP systems.[3] This section describes a few of the components – there are many more!

SystemOnTPTP [17] is a TPTP World utility that allows an ATP problem or solution to be easily and quickly submitted in various ways to a range of ATP systems and tools. The utility uses a suite of currently available ATP systems and tools, whose properties (input format, reporting of result status, etc.) are stored in a simple text database. The implementation of SystemOnTPTP

[2] Available at `http://www.tptp.org/TSTP`

[3] The tools and libraries are available at `http://www.tptp.org/TPTPWorld.tgz`

uses several subsidiary tools to preprocess the input, control the execution of the ATP systems and tools, and postprocess the output. On the input side TPTP2X or TPTP4X (TPTP World tools for parsing and transforming TPTP format formulae) is used to prepare the input for processing. A program called `TreeLimitedRun` is used to monitor the execution of ATP systems and tools, and limit the CPU time and memory used – `TreeLimitedRun` monitors processes more tightly than is possible with standard operating system calls. Finally a program called `X2tptp` converts an ATP system's output to TPTP format, if requested by the user.

The TPTP World ATP system recommendation service uses the ATP system ratings to recommend ATP systems for solving a new problem. A new problem is analysed to determine its SPCs – one for establishing theoremhood or unsatisfiability, and one for establishing countersatisfiability or satisfiability. The systems that contributed to the problem ratings in those SPCs are recommended, in decreasing order of system rating for the SPCs.

GDV [18] is a TPTP World tool that uses structural and then semantic techniques to verify TPTP format derivations. Structural verification checks that inferences have been done correctly in the context of the derivation, e.g., checking that the derivation is acyclic, checking that assumptions have been discharged, and checking that introduced symbols (e.g., in Skolemization) are distinct. Semantic verification checks the expected semantic relationship between the parents and inferred formula of each inference step. This is done by encoding the expectation as a logical obligation in an ATP problem, and then discharging the obligation by solving the problem with trusted ATP systems. The expected semantic relationship between the parents and inferred formula of an inference step depends on the intent of the inference rule used. For example, deduction steps expect the inferred formula to be a theorem of its parent formulae. The expected relationship is recorded as an SZS value in each inferred formula of a derivation. GDV uses the SystemOnTPTP utility to execute the trusted ATP systems.

AGInTRater is a TPTP World tool that evaluates the interestingness of formulae in TPTP format derivations (AGInTRater is a component of the AGInT system that discovers interesting theorems of a given set of axioms [13]). AGInTRater has a filter that measures up to eight "interestingness" features of formulae (some features are inappropriate in some situations): preprocessing detects and discards obvious tautologies, obviousness estimates the difficulty of proving a formula, weight estimates the effort required to read a formula, complexity estimates the effort required to understand a formula, surprisingness measures new relationships between function and predicate symbols in a formula, intensity measures how much a formula summarizes information from its leaf ancestors, adaptivity measures how tightly the universally quantified variables of a formula are constrained, and focus measures the extent to which a formula is making a positive or negative statement. Formulae that pass the majority of the filters

are passed to a static ranker, which combines the measures from the filters with a measure of usefulness, which measures how much a formula has contributed to proofs of further formulae. The scores are then normalized and averaged to produce an interestingness score. AGInTRater is a unique feature of the IDV derivation visualizer, described next.

IDV [29] is a TPTP World tool for graphical rendering and analysis of TPTP format derivations. IDV provides an interactive interface that allows the user to quickly view features of the derivation, and access analysis facilities. The left hand side of Figure 1 shows the rendering of the derivation output by the EP ATP system, for the TPTP problem `PUZ001+1`. The IDV window is divided into three panes: the top pane contains control buttons and sliders, the middle pane shows the rendered DAG, and the bottom pane gives the text of the annotated formula for the node pointed to by the mouse. The rendering of the derivation DAG uses shapes, colors, and tags to provide information about the derivation. The user can interact with the rendering in various ways using mouse-over and mouse clicks. The buttons and sliders in the control pane provide a range of manipulations on the rendering – zooming, hiding and displaying parts of the DAG, and access to GDV for verification. A particularly novel feature of IDV is its ability to provide a synopsis of a derivation by using the AGInTRater to identify interesting lemmas, and hiding less interesting intermediate formulae. A synopsis is shown on the right hand side of Figure 1. IDV is easily appreciated by using it through the online interfaces described next.

All of the TPTP World services described in this section, and a few more besides, are accessible in web browser interfaces.[4] These interfaces execute most of the services using the SystemOnTPTP utility. The SystemB4TPTP interface provides access to problem pre-processing tools, including parsers, pretty-printers, first-order form to clause normal form conversion, type checking, and axiom relevance measures. It additionally provides a facility to convert a problem to TPTP format from some other formats. The SystemOnTPTP interface provides the core functionality of the SystemOnTPTP utility, to submit an ATP problem to ATP systems. It additionally provides system reports and recommendations for which systems to use on a given problem, based on the SPCs and system ratings. The SystemOnTPTP interface also has direct access to the SSCPA ATP meta-system [25], which runs multiple recommended ATP systems in competition parallel. Finally, the SystemOnTSTP interface provides access to derivation post-processing tools, including parsers, pretty-printers, answer extraction, GDV, AGInTRater, and IDV.

As an alternative to interactive access via a web browser, the TPTP World services can also be accessed programmatically using `http POST` multi-part form requests. Perl and Java code for doing this is available as part of the TPTP World. The benefits of using the TPTP World online service include not installing ATP systems and tools locally, having access to the latest version of ATP systems (e.g., versions from CASC and unpublished versions), and full access to all the TPTP World tools. You are encouraged to abuse my server!

[4] Available starting at `http://www.tptp.org/cgi-bin/SystemOnTPTP`

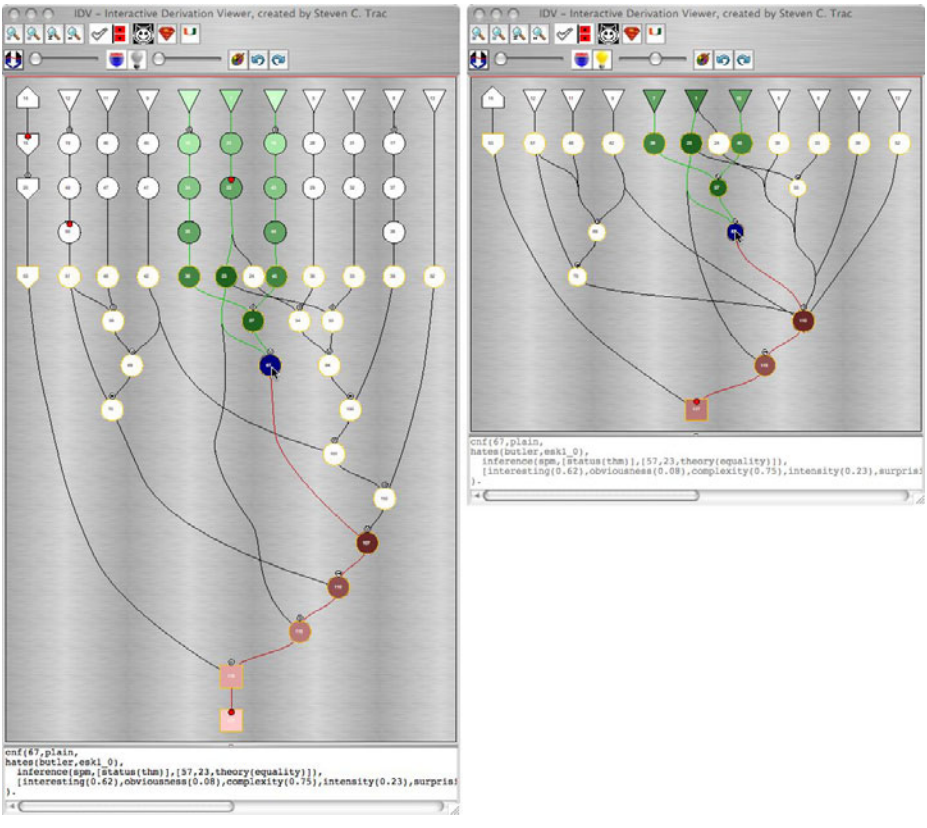


Fig. 1. EP's proof by refutation of PUZ001+1

4 TPTP World Applications

Research scientists in the Robust Software Engineering Group of the Intelligent Systems Division of NASA Ames have developed, implemented, and evaluated a certification approach that uses Hoare-style techniques to formally demonstrate the safety of aerospace programs that are automatically generated from high-level specifications [5]. A verification condition generator processes the automatically generated code, and produces a set of safety obligations in the form of TPTP format problems that are provable if and only if the code is safe. The obligation problems are discharged using the SSCPA ATP meta-system (mentioned in Section 3), to produce TPTP format proofs that are then verified by GDV. The proofs and verification logs serve as safety certificates for authorities like the FAA.

The MPTP project [31] aims to link-up the large formal Mizar Mathematical Library (MML) [15] with ATP technology, and to boost the development of AI-based ATP methods. The MPTP system converts Mizar format problems to an extended TPTP language that adds term-dependent sorts and abstract (Fraenkel) terms to the TPTP syntax. Problems in the extended language are transformed to standard TPTP format by relativization of sorts and deanonymization of abstract terms. Finding proofs for these problems provides cross verification of the underlying Mizar proofs. Mizar proofs are also exported as TPTP format derivations,[5] allowing a number of ATP experiments and use of TPTP tools, e.g., GDV and IDV.

Sledgehammer [12] is a linkup from the interactive theorem prover Isabelle/HOL [11] to first-order ATP systems. Sledgehammer is activated by the user, who wishes to prove a goal from the current background theory. A set of relevant facts is extracted from the background theory – this set is almost inevitably a superset of what is required for a proof of the goal. The goal and background facts are translated into a TPTP format first-order logic problem, which is given to one or more ATP systems. If one of the ATP systems finds a proof, the axioms used in the proof are extracted, and the goal is reproved using the Metis ATP system [8], which natively outputs an Isabelle/HOL proof. One of the options within Sledgehammer is to use the TPTP World service to run ATP systems; in particular, this is done for running the Vampire ATP system.

The Naproche (NAtural language PROof CHEcking) system [4] automatically checks mathematical texts (written in the Naproche controlled natural language) for logical correctness. Each statement of a text is required to follow from preceding information in the text. The text is first translated into a linguistic representation called a Proof Representation Structure, from which TPTP format proof obligations are created. The proof obligations are discharged by calling an ATP system, using the TPTP World service. The TPTP format proofs produced by the ATP system are used to create the Naproche output, and also analyzed (using another TPTP World tool) to help select what preceding information should be provided in subsequent proof obligations. IDV is also used to debug the Naproche system.

SPASS-XDB [26] is a modified version of the SPASS ATP system, with the ability to retrieve world knowledge axioms from a range of external sources (databases, internet, computations, etc.) asynchronously, on demand, during it's deduction. Figure 2 shows the system architecture, which is comprised of the SPASS-XDB ATP system, mediators, and external sources of world knowledge axioms. SPASS-XDB accepts a problem file containing (i) specifications for the external sources, (ii) internal axioms, and (iii) a conjecture to be proved. All the formulae are written in the TPTP language, and commonly axioms from a TPTP format export of the SUMO ontology [10] are provided as internal axioms that allow deep reasoning over the world knowledge. SPASS-XDB augments SPASS' classic CNF saturation algorithm with steps to request and accept world knowledge axioms. Requests are written as TPTP format "questions", and axioms

[5] Available at `http://www.tptp.org/MizarTPTP`

are delivered as TPTP format "answers", using SZS standards. The requests are made and the axioms delivered asynchronously, so that SPASS-XDB continues its deduction process while the axioms are being retrieved from the (comparatively slow) external sources. SPASS-XDB is available in the SystemOnTPTP interface.

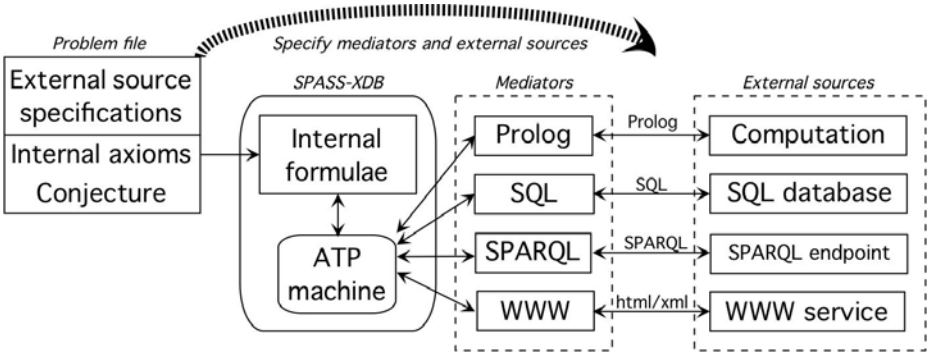


Fig. 2. System Architecture

The CADE ATP System Competition (CASC) [27] is held annually to evaluate the performance of sound, fully automatic ATP systems – it is the world championship for such systems. The design and implementation of CASC is closely linked to the TPTP World. The divisions and problem categories of CASC are similar to the SPCs used in the TPTP problem and system rating scheme. The problems used in CASC are taken from the TPTP problem library, and the TPTP problem ratings are used to select appropriately difficult problems that differentiate between the systems entered into CASC. CASC has been a catalyst for impressive improvements in ATP, stimulating both theoretical and implementation advances [9]. The positive effects of CASC on ATP system development have had reciprocal positive effects on the TPTP. Observers at the event have been encouraged to contribute their problems to the TPTP. The ATP systems entered into CASC are the most recent versions available, and after CASC they are added to the SystemOnTPTP suite. The systems are run over the TPTP problem library to update the TSTP solution library, which in turn provides updated problem and system ratings.

5 Current Developments in the TPTP World

The TPTP (problem library, and later World) was originally developed for untyped first-order logic. In 2008-9 the Typed Higher-order Form (THF) was added, and the first release of the TPTP problem library with THF problems was in 2009 [21]. The TPTP THF language is a syntactically conservative extension of the

untyped first-order language. It has been divided into three layers named THF0, THF, and THFX. THF0 [2] is a core subset based on Church's simply typed lambda calculus. THF provides a richer type system, the ability to reason about types, more term and type constructs, and more connectives. THFX provides "syntactic sugar" that is usefully expressive. In conjunction with the addition of THF problems to the TPTP problem library, other components of the TPTP World were extended to support THF. This includes parsers, pretty-printers, type checkers, export to existing higher-order ATP systems' formats, and proof presentation in IDV. Additionally, four theorem proving ATP systems and two model finders have been produced, and are available in SystemOnTPTP. The addition of THF to the TPTP World has had an immediate impact on progress in the development of automated reasoning in higher-order logic [22].

Following the addition of THF to the TPTP World, steps are now being taken to add support for a Typed First-order Form (TFF) in the TPTP World. The TPTP TFF language is (like the THF language) a syntactically conservative extension of the untyped first-order language. TFF problems are being collected,[6] and a TPTP release including TFF problems is expected in 2010. An ATP system has been built to solve these problems, by translating the problems to an equivalent first-order form and calling an existing first-order ATP system.

As well as being useful in it own right, TFF also provides a foundation for adding support for arithmetic in the TPTP World. For arithmetic, the TFF language includes atomic types for integer, rational and real numbers, with corresponding forms for constants. A core set of predicates and functions for arithmetic relations and operations has been defined. The extent to which ATP systems can do arithmetic is expected to vary, from a simple ability to evaluate ground terms, through an ability to instantiate variables in arithmetic expressions, to extensive algebraic manipulations. Five ATP systems have been configured for solving TFF arithmetic problems (although none are yet able to deal with all the TFF arithmetic constructs). ATP systems have been notorious for their lack of arithmetic capabilities, and it is expected that this development in the TPTP World will provide useful infrastructure that will help developers add arithmetic to their ATP systems, and consequently provide new functionality to ATP system users.

A long time challenge of artificial intelligence has been to provide a system that is capable of reasoning with world knowledge, with a natural language interface [30]. As a small step towards that goal in the TPTP World, SystemB4TPTP now supports input in Attempto Controlled English (ACE) [6]. ACE input is translated to first-order logic by an online service at the University of Zurich, and the TPTP format axioms and conjecture that are returned can be submitted to an ATP system. A modified version of the translation is being developed, which adds axioms that link the predicate and function symbols used in the translated formulae to those used in SUMO and the sources of external data available to SPASS-XDB. This will allow SPASS-XDB to be used, to provide deep reasoning with world knowledge.

[6] Available at `http://www.tptp.org/TPTP/Proposals/SampleProblems/TFF`

6 Conclusion

The TPTP World is a well known and established infrastructure that supports research, development, and deployment of Automated Theorem Proving (ATP) systems for classical logics. This paper has given an overview of the TPTP World, and shown how it can be successfully applied. Recent developments in the TPTP World are expected to broaden the user base of ATP system developers, and attract new ATP system users.

The TPTP problem library is a core component of the TPTP World, and users of the TPTP World are encouraged to contribute their problems to the library. The common adoption of TPTP standards, especially the TPTP language and SZS ontology, has made it easier to build complex reasoning systems. Developers are encouraged to adopt the TPTP standards, to provide compliant components that can be combined to meet users' needs.

References

1. Benzmüller, C., Paulson, L., Theiss, F., 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)
2. Benzmüller, C., Rabe, F., Sutcliffe, G.: THF0 - The Core TPTP Language for Classical Higher-Order Logic. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 491–506. Springer, Heidelberg (2008)
3. Claessen, K., Sörensson, N.: New Techniques that Improve MACE-style Finite Model Finding. In: Baumgartner, P., Fermueller, C. (eds.) Proceedings of the CADE-19 Workshop: Model Computation - Principles, Algorithms, Applications (2003)
4. 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)
5. Denney, E., Fischer, B., Schumann, J.: Using Automated Theorem Provers to Certify Auto-generated Aerospace Software. In: Basin, D., Rusinowitch, M. (eds.) IJCAR 2004. LNCS (LNAI), vol. 3097, pp. 198–212. Springer, Heidelberg (2004)
6. Fuchs, N., Kaljurand, K., Kuhn, T.: Attempto Controlled English for Knowledge Representation. In: Baroglio, C., Bonatti, P.A., Małuszyński, J., Marchiori, M., Polleres, A., Schaffert, S. (eds.) Reasoning Web. LNCS, vol. 5224, pp. 104–124. Springer, Heidelberg (2008)
7. Hillenbrand, T.: Citius altius fortius: Lessons Learned from the Theorem Prover Waldmeister. In: Dahn, I., Vigneron, L. (eds.) Proceedings of the 4th International Workshop on First-Order Theorem Proving. Electronic Notes in Theoretical Computer Science, vol. 86.1, pp. 1–13 (2003)
8. Hurd, J.: First-Order Proof Tactics in Higher-Order Logic Theorem Provers. In: Archer, M., Di Vito, B., Munoz, C. (eds.) Proceedings of the 1st International Workshop on Design and Application of Strategies/Tactics in Higher Order Logics, number NASA/CP-2003-212448 in NASA Technical Reports, pp. 56–68 (2003)

9. Nieuwenhuis, R.: The Impact of CASC in the Development of Automated Deduction Systems. AI Communications 15(2-3), 77–78 (2002)
10. Niles, I., Pease, A.: Towards A Standard Upper Ontology. In: Welty, C., Smith, B. (eds.) Proceedings of the 2nd International Conference on Formal Ontology in Information Systems, pp. 2–9 (2001)
11. Nipkow, T., Paulson, L., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002)
12. Paulson, L., Susanto, K.: 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)
13. Puzis, Y., Gao, Y., Sutcliffe, G.: Automated Generation of Interesting Theorems. In: Sutcliffe, G., Goebel, R. (eds.) Proceedings of the 19th International FLAIRS Conference, pp. 49–54. AAAI Press, Menlo Park (2006)
14. Riazanov, A., Voronkov, A.: The Design and Implementation of Vampire. AI Communications 15(2-3), 91–110 (2002)
15. Rudnicki, P.: An Overview of the Mizar Project. In: Proceedings of the 1992 Workshop on Types for Proofs and Programs, pp. 311–332 (1992)
16. Schulz, S.: E: A Brainiac Theorem Prover. AI Communications 15(2-3), 111–126 (2002)
17. Sutcliffe, G.: SystemOnTPTP. In: McAllester, D. (ed.) CADE 2000. LNCS, vol. 1831, pp. 406–410. Springer, Heidelberg (2000)
18. Sutcliffe, G.: Semantic Derivation Verification. International Journal on Artificial Intelligence Tools 15(6), 1053–1070 (2006)
19. Sutcliffe, G.: The SZS Ontologies for Automated Reasoning Software. In: Sutcliffe, G., Rudnicki, P., Schmidt, R., Konev, B., Schulz, S. (eds.) Proceedings of the LPAR Workshops: Knowledge Exchange: Automated Provers and Proof Assistants, and The 7th International Workshop on the Implementation of Logics. CEUR Workshop Proceedings, vol. 418, pp. 38–49 (2008)
20. 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)
21. Sutcliffe, G., Benzmüller, C.: Automated Reasoning in Higher-Order Logic using the TPTP THF Infrastructure. Journal of Formalized Reasoning 3(1), 1–27 (2010)
22. 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.) Automated Deduction – CADE-22. LNCS, vol. 5663, pp. 116–130. Springer, Heidelberg (2009)
23. Sutcliffe, G., Fuchs, M., Suttner, C.: Progress in Automated Theorem Proving, 1997-1999. In: Hoos, H., Stützle, T. (eds.) Proceedings of the IJCAI 2001 Workshop on Empirical Methods in Artificial Intelligence, pp. 53–60 (2001)
24. Sutcliffe, G., Schulz, S., Claessen, K., Van Gelder, A.: Using the TPTP Language for Writing Derivations and Finite Interpretations. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 67–81. Springer, Heidelberg (2006)
25. Sutcliffe, G., Seyfang, D.: Smart Selective Competition Parallelism ATP. In: Kumar, A., Russell, I. (eds.) Proceedings of the 12th International FLAIRS Conference, pp. 341–345. AAAI Press, Menlo Park (1999)
26. Sutcliffe, G., Suda, M., Teyssandier, A., Dellis, N., de Melo, G.: Progress Towards Effective Automated Reasoning with World Knowledge. In: Murray, C., Guesgen, H. (eds.) Proceedings of the 23rd International FLAIRS Conference. AAAI Press, Menlo Park (2010) (to appear)

27. Sutcliffe, G., Suttner, C.: The State of CASC. AI Communications 19(1), 35–48 (2006)
28. Sutcliffe, G., Suttner, C.B.: Evaluating General Purpose Automated Theorem Proving Systems. Artificial Intelligence 131(1-2), 39–54 (2001)
29. Trac, S., Puzis, Y., Sutcliffe, G.: An Interactive Derivation Viewer. In: Autexier, S., Benzmüller, C. (eds.) Proceedings of the 7th Workshop on User Interfaces for Theorem Provers, 3rd International Joint Conference on Automated Reasoning. Electronic Notes in Theoretical Computer Science, vol. 174, pp. 109–123 (2006)
30. Turing, A.: Computing Machinery and Intelligence. Mind 59(236), 433–460 (1950)
31. Urban, J.: MPTP 0.2: Design, Implementation, and Initial Experiments. Journal of Automated Reasoning 37(1-2), 21–43 (2006)
32. Van Gelder, A., Sutcliffe, G.: Extending the TPTP Language to Higher-Order Logic with Automated Parser Generation. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 156–161. Springer, Heidelberg (2006)
33. Weidenbach, C., Fietzke, A., Kumar, R., Suda, M., Wischnewski, P., Dimova, D.: SPASS Version 3.5. In: Schmidt, R.A. (ed.) Automated Deduction – CADE-22. LNCS, vol. 5663, pp. 140–145. Springer, Heidelberg (2009)

Speed-Up Techniques for Negation in Grounding

Amir Aavani, Shahab Tasharrofi, Gulay Unel, Eugenia Ternovska, and David Mitchell

Simon Fraser University
aaa78@cs.sfu.ca, sta44@cs.sfu.ca, gunel@cs.uwaterloo.ca, ter@cs.sfu.ca, mitchell@cs.sfu.ca

Abstract. Grounding is the task of reducing a first order formula to ground formula that is equivalent on a given universe, and is important in many kinds of problem solving and reasoning systems. One method for grounding is based on an extension of the relational algebra, exploiting the fact that grounding over a given domain is similar to query answering. In this paper, we introduce two methods for speeding up algebraic grounding by reducing the size of tables produced. One method employs rewriting of the formula before grounding, and the other uses a further extension of the algebra that makes negation efficient. We have implemented the methods, and present experimental evidence of their effectiveness.

1 Introduction

Challenging combinatorial search problems are ubiquitous in applications of computer science, including many problems which are NP-hard and challenging in practice. A variety of algorithmic approaches to such problems are used in practice, among these the application of general purpose solvers, such as SAT and ILP solvers. Effective use of these technologies typically requires considerable expertise, both in terms of the application problem at hand and the solving technology. Many users do not have such expertise easily available, and even for experts producing correct and effective reductions to SAT or integer programming is often a time consuming and error-prone exercise.

An approach to improving accessibility and ease of use of combinatorial search methods is to provide a high-level specification or modelling language, in which both expert and non-expert users may specify their problems without direct concern for the underlying solving techniques. The input for a (ground) solver is then generated automatically from a specification together with an instance. Examples of languages for this purpose include ASP languages [7,6], the language of the IDP system [10], SPEC2SAT [2] and ESSENSE [5]. The process of mapping a high-level specification and an instance to a low-level solver input language is grounding.

The input for a combinatorial search problem is a finite structure, and the task is to construct one or more additional relations satisfying a certain property. For example, in the Hamiltonian cycle problem, the input is a graph $G = \langle V; E \rangle$, and the task is to construct a path P of a certain sort in G, thus obtaining an expanded structure $G' = \langle V; E, P \rangle$. When the property is specified by a formula of first order logic (FO), the class of problems which can be specified are those in NP [4]. Examples of systems with specification languages that are natural extensions of FO are described in [8] and [10]. These systems ground to languages that are extensions of SAT. For any fixed FO

E.M. Clarke and A. Voronkov (Eds.): LPAR-16, LNAI 6355, pp. 13–26, 2010.

specification, grounding is polynomial time, but in practice current grounders are often too slow in grounding large instances of some problems.

A method for grounding FO specifications based on an extension of the relational algebra was introduced in [9], and a prototype implementation based on this method is reported in [8]. In this paper, we present refinements to this method to improve the speed of grounding. When naively implemented, the algebraic grounding approach may be slow because very large intermediate tables may be produced. Much work in database query optimization aims to reduce the size of intermediate tables. However, in our observation, the single largest impediment to speed in grounding is due to negation, which is not well studied in query optimization. The problem is that the complement operation, as defined in [8], tends to produce very large tables, which are sometimes universal (containing all possible tuples) or nearly so.

Here, we present two approaches for mitigating the costs of negation. To our knowledge, these have no close analogs in standard database techniques. The first approach, denoted FR for "formula re-writing" constructs a logically equivalent formula which minimizes the cost of negations. The second method, denoted T/F for "True-False Tables", is a further generalization of the relational algebra in which complementation is inexpensive. Both methods are heuristic, meaning there are conditions under which they will produce slow rather than faster grounding, but in practice we generally observe significant speedups.

We have implemented both of the methods in a new grounder. Each method produces order of magnitude speedups over the version of the grounder without these refinements. Moreover, while the naive version of the grounder is slow in comparison with other existing model expansion grounders, the versions with these improvements are highly competitive.

2 Background

We formalize combinatorial search problems in terms of the logical problem of *model expansion (MX)*, defined here for an arbitrary logic $\mathscr{L}$.

Definition 1 (MX). *Given an $\mathscr{L}$-sentence ϕ, over the union of disjoint vocabularies σ and ε, and a finite structure $\mathscr{A}$ for vocabulary σ, find a structure $\mathscr{B}$ that is an expansion of $\mathscr{A}$ to $\sigma \cup \varepsilon$ such that $\mathscr{B} \models \phi$.*

In this paper, ϕ is a problem specification formula, and is fixed for each search problem. $\mathscr{A}$ always denotes a finite σ-structure, called the instance structure, σ is the instance vocabulary, and ε the expansion vocabulary.

Example 1. The following formula ϕ of first order logic constitutes a specification for Graph 3-Colouring:

$$\forall x\,[(R(x) \vee B(x) \vee G(x))] \;\wedge$$
$$\forall x\,\neg[(R(x) \wedge B(x)) \vee (R(x) \wedge G(x)) \vee (B(x) \wedge G(x))] \;\wedge$$
$$\forall x \forall y\,[(E(x,y) \vee E(y,x)) \supset (\neg(R(x) \wedge R(y)) \wedge \neg(B(x) \wedge B(y)) \wedge \neg(G(x) \wedge G(y)))]$$

An instance is a structure for vocabulary $\sigma = \{E\}$, i.e., a graph $\mathscr{A} = \mathscr{G} = (V;E)$. The task is to find an expansion $\mathscr{B}$ of $\mathscr{A}$ that satisfies ϕ:

$$\underbrace{\overbrace{(V;E^{\mathscr{A}}}^{\mathscr{A}}, R^{\mathscr{B}},B^{\mathscr{B}},G^{\mathscr{B}})}_{\mathscr{B}} \models \phi.$$

Interpretations of the expansion vocabulary $\varepsilon = \{R,B,G\}$, for structures $\mathscr{B}$ that satisfy ϕ, are proper 3-colourings of $\mathscr{G}$.

The grounding task is to produce a ground formula $\psi = Gnd(\phi,\mathscr{A})$, such that models of ψ correspond to solutions for instance $\mathscr{A}$. Formally, to ground we bring domain elements into the syntax by expanding the vocabulary with a new constant symbol for each element of the domain. For domain A, the domain of structure $\mathscr{A}$, we denote the set of such constants by $\tilde{A}$. In practice, the ground formula should contain no occurrences of the instance vocabulary, in which case we call it reduced.

Definition 2 (Reduced Grounding for MX). *Formula ψ is a* reduced grounding *of formula ϕ over σ-structure $\mathscr{A} = (A;\sigma^{\mathscr{A}})$ if*
1) ψ is a ground formula over $\varepsilon \cup \tilde{A}$, and
2) for every expansion structure $\mathscr{B} = (A;\sigma^{\mathscr{A}},\varepsilon^{\mathscr{B}})$ over $\sigma \cup \varepsilon$, $\mathscr{B} \models \phi$ iff $(\mathscr{B},\tilde{A}^{\mathscr{B}}) \models \psi$, where $\tilde{A}^{\mathscr{B}}$ is the standard interpretation of the new constants $\tilde{A}$.

Proposition 1. *Let ψ be a reduced grounding of ϕ over σ-structure $\mathscr{A}$. Then $\mathscr{A}$ can be expanded to a model of ϕ iff ψ is satisfiable.*

Producing a reduced grounding with respect to a given structure $\mathscr{A}$ can be done by an algorithm that, for each fixed FO formula, runs in time polynomial in the size of $\mathscr{A}$. Such a grounding algorithm implements a polytime reduction to SAT for each NP search problem. Simple grounding algorithms, however, do not reliably produce groundings for large instances of interesting problems fast enough in practice.

Grounding for MX is a generalization of query answering. Given a structure (database) $\mathscr{A}$, a boolean query is a formula ϕ over the vocabulary of $\mathscr{A}$, and query answering is equivalent to evaluating whether ϕ is true, i.e., $\mathscr{A} \models \phi$. For model expansion, ϕ has some additional vocabulary beyond that of $\mathscr{A}$, and producing a reduced grounding involves evaluating out the instance vocabulary, and producing a ground formula representing the possible expansions of $\mathscr{A}$ for which ϕ is true.

The grounding algorithms in this paper construct a grounding by a bottom-up process that parallels database query evaluation, based on an extension of the relational algebra. For each sub-formula $\phi(\bar{x})$ with free variables $\bar{x}$, we call the set of reduced groundings for ϕ under all possible ground instantiations of $\bar{x}$ *an answer to* $\phi(\bar{x})$. We represent answers with tables on which an extended algebra operates.

An X-relation is a k-ary relation associated with a k-tuple of variables X, representing a set of instantiations of the variables of X. It is a central notion in databases. In extended X-relations, introduced in [9], each tuple γ is associated with a formula ψ. For convenience, we use $\top$ and $\bot$ as propositional formulas which are always true and, respectively, false.

Definition 3 (extended X-relation; function $\delta_{\mathscr{R}}$). *Let A be a domain, and X a tuple of variables with $|X| = k$. An* extended X-relation *$\mathscr{R}$ over A is a set of pairs (γ,ψ) s.t.*

1) $\gamma : X \to A$, *and*
2) ψ *is a formula, and*
3) if $(\gamma, \psi) \in \mathscr{R}$ *and* $(\gamma, \psi') \in \mathscr{R}$ *then* $\gamma = \gamma'$.
The function $\delta_{\mathscr{R}}$ represented by $\mathscr{R}$ *is a mapping from k-tuples* γ *of elements of the domain A to formulas, defined by:*

$$\delta_{\mathscr{R}}(\gamma) = \begin{cases} \psi & \text{if } (\gamma, \psi) \in \mathscr{R}, \\ \bot & \text{if there is no pair } (\gamma, \psi) \in \mathscr{R}. \end{cases}$$

For brevity, we sometimes write $\gamma \in \mathscr{R}$ to mean that there exists ψ such that $(\gamma, \psi) \in \mathscr{R}$. We also sometimes call extended X-relations simply tables. To refer to X-relations for some concrete set X of variables, rather than in general, we write X-relation.

Definition 4 (answer to ϕ wrt $\mathscr{A}$). *Let* ϕ *be a formula in* $\sigma \cup \varepsilon$ *with free variables X,* $\mathscr{A}$ *a* σ*-structure with domain A, and* $\mathscr{R}$ *an extended X-relation over* $\mathscr{A}$. *We say* $\mathscr{R}$ *is* an answer to ϕ wrt $\mathscr{A}$ *if for any* $\gamma : X \to A$, *we have that* $\delta_{\mathscr{R}}(\gamma)$ *is a reduced grounding of* $\phi[\gamma]$ *over* $\mathscr{A}$. *Here,* $\phi[\gamma]$ *denotes the result of instantiating free variables in* ϕ *according to* γ.

Since a sentence has no free variables, the answer to a sentence ϕ is a zero-ary extended X-relation, containing a single pair $(\langle\rangle, \psi)$, associating the empty tuple with formula ψ, which is a reduced grounding of ϕ.

Example 2. Let $\sigma = \{P\}$ and $\varepsilon = \{E\}$, and let $\mathscr{A}$ be a σ-structure with $P^{\mathscr{A}} = \{(1,2,3),(3,4,5)\}$. The following extended relation $\mathscr{R}$ is an answer to $\phi_1 \equiv P(x,y,z) \wedge E(x,y) \wedge E(y,z)$:

x	y	z	ψ
1	2	3	$E(1,2) \wedge E(2,3)$
3	4	5	$E(3,4) \wedge E(4,5)$

Observe that $\delta_{\mathscr{R}}(1,2,3) = E(1,2) \wedge E(2,3)$ is a reduced grounding of $\phi_1[(1,2,3)] = P(1,2,3) \wedge E(1,2) \wedge E(2,3)$, and $\delta_{\mathscr{R}}(1,1,1) = \bot$ is a reduced grounding of $\phi_1[(1,1,1)]$. The following extended relation is an answer to $\phi_2 \equiv \exists z \phi_1$:

x	y	ψ
1	2	$E(1,2) \wedge E(2,3)$
3	4	$E(3,4) \wedge E(4,5)$

Here, $E(1,2) \wedge E(2,3)$ is a reduced grounding of $\phi_2[(1,2)]$. Finally, the following represents an answer to $\phi_3 \equiv \exists x \exists y \phi_2$, where the single formula is a reduced grounding of ϕ_3.

ψ
$[E(1,2) \wedge E(2,3)] \vee [E(3,4) \wedge E(4,5)]$

The relational algebra has operations corresponding to each connective and quantifier in FO, as follows: complement (negation); join (conjunction); union (disjunction), projection (existential quantification); division or quotient (universal quantification). Following [9,8], we generalize each to extended X-relations as follows.

Definition 5 (Extended Relational Algebra). *Let $\mathscr{R}$ be an extended X-relation and $\mathscr{S}$ an extended Y-relation, both over domain A.*

1. *$\neg\mathscr{R}$ is the extended X-relation $\neg\mathscr{R} = \{(\gamma,\psi) \mid \gamma : X \to A, \delta_{\mathscr{R}}(\gamma) \neq \top$, and $\psi = \neg\delta_{\mathscr{R}}(\gamma)\}$*
2. *$\mathscr{R} \bowtie \mathscr{S}$ is the extended $X \cup Y$-relation $\{(\gamma,\psi) \mid \gamma : X \cup Y \to A, \gamma|_X \in \mathscr{R}, \gamma|_Y \in \mathscr{S}$, and $\psi = \delta_{\mathscr{R}}(\gamma|_X) \wedge \delta_{\mathscr{S}}(\gamma|_Y)\}$;*
3. *$\mathscr{R} \cup \mathscr{S}$ is the extended $X \cup Y$-relation $\mathscr{R} \cup \mathscr{S} = \{(\gamma,\psi) \mid \gamma|_X \in \mathscr{R}$ or $\gamma|_Y \in \mathscr{S}$, and $\psi = \delta_{\mathscr{R}}(\gamma|_X) \vee \delta_{\mathscr{S}}(\gamma|_Y)\}$.*
4. *For $Z \subseteq X$, the Z-projection of $\mathscr{R}$, denoted by $\pi_Z(\mathscr{R})$, is the extended Z-relation $\{(\gamma',\psi) \mid \gamma' = \gamma|_Z$ for some $\gamma \in \mathscr{R}$ and $\psi = \bigvee_{\{\gamma \in \mathscr{R} \mid \gamma' = \gamma|_Z\}} \delta_{\mathscr{R}}(\gamma)\}$.*
5. *For $Z \subseteq X$, the Z-quotient of $\mathscr{R}$, denoted by $d_Z(\mathscr{R})$, is the extended Z-relation $\{(\gamma',\psi) \mid \forall\gamma(\gamma : X \to A \wedge \gamma|_Z = \gamma' \Rightarrow \gamma \in \mathscr{R})$, and $\psi = \bigwedge_{\{\gamma \in \mathscr{R} \mid \gamma' = \gamma|_Z\}} \delta_{\mathscr{R}}(\gamma)\}$.*

To ground using this algebra, we apply the algebra inductively on the structure of the formula, just as the standard relational algebra may be applied for query evaluation. We define the answer to atomic formula $P(\bar{x})$ as follows. If P is an instance predicate, the answer to P is the set of tuples $(\bar{a},\top)$, for $\bar{a} \in P^{\mathscr{A}}$. If P is an expansion predicate, the answer is the set of all pairs $(\bar{a},P(\bar{a}))$, where $\bar{a}$ is a tuple of elements from the domain A. Correctness of the method then follows, by induction on the structure of the formula, from the following proposition.

Proposition 2. *Suppose that $\mathscr{R}$ is an answer to ϕ_1 and $\mathscr{S}$ is an answer to ϕ_2, both with respect to (wrt) structure $\mathscr{A}$. Then*

1. *$\neg\mathscr{R}$ is an answer to $\neg\phi_1$ wrt $\mathscr{A}$;*
2. *$\mathscr{R} \bowtie \mathscr{S}$ is an answer to $\phi_1 \wedge \phi_2$ wrt $\mathscr{A}$;*
3. *$\mathscr{R} \cup \mathscr{S}$ is an answer to $\phi_1 \vee \phi_2$ wrt $\mathscr{A}$;*
4. *If Y is the set of free variables of $\exists\bar{z}\phi_1$, then $\pi_Y(\mathscr{R})$ is an answer to $\exists\bar{z}\phi_1$ wrt $\mathscr{A}$.*
5. *If Y is the set of free variables of $\forall\bar{z}\phi_1$, then $d_Y(\mathscr{R})$ is an answer to $\forall\bar{z}\phi_1$ wrt $\mathscr{A}$.*

The straightforward proof for cases 1, 2 and 4 is given in [9]; the other cases follow easily.

The answer to an atomic formula $P(\bar{x})$, where P is from the expansion vocabulary, is formally a universal table, but in practice we may represent this table implicitly and avoid explicitly enumerating the tuples. As operations are applied, some subset of columns remain universal, while others do not. Again, those columns which are universal may be represented implicitly. This could be treated as an implementation detail, but the use of such implicit representations dramatically affects the cost of operations, and so it is useful to further generalize our extended X-relations. Following [8], we call the variables which are implicitly universal "hidden" variables, as they are not represented explicitly in the tuples, and the other variables "explicit" variables.

Definition 6 (Extended Hidden X-Relation $\mathscr{R}_Y$; $\delta_{\mathscr{R}_Y}$). *Let X, Y be tuples of variables, with $Y \subseteq X$ (when viewed as sets), and $|X| = k$. An extended hidden X-relation $\mathscr{R}_Y$ is a set of tuples (γ,ψ) s.t.*

1) $\gamma : X \backslash Y \to A$, and
2) ψ is a formula, and
3) if $(\gamma,\psi) \in \mathscr{R}_Y$ and $(\gamma,\psi') \in \mathscr{R}_Y$, then $\psi = \psi'$.

The function $\delta_{\mathscr{R}_Y}$ represented by $\mathscr{R}_Y$ is a mapping from k-tuples γ' of elements of the domain A to formulas, defined by

$$\delta_{\mathscr{R}_Y}(\gamma') = \begin{cases} \psi & \text{if } (\gamma' \mid_{X\setminus Y}, \psi) \in \mathscr{R}, \\ \bot & \text{if there is no pair } (\gamma' \mid_{X\setminus Y}, \psi) \in \mathscr{R}. \end{cases}$$

So, an extended hidden X-relation $\mathscr{R}_Y$ is a compact representation of an extended X-relation by an extended $X\setminus Y$-relation, which may be used whenever the columns for variables of Y are universal. If $X = Y$, we have a compact representation of a universal relation; if $Y = \emptyset$, we have a normal extended X-relation.

All the operations of the algebra generalize easily. The hidden variables technique does not alter the semantics of the operations. Henceforth, the term table may denote either an extended X-relation or a hidden extended X-relation.

Definition 7. ***Basic Grounding Method (B):*** *We ground a sentence ϕ wrt $\mathscr{A}$ using this algebra, proceeding bottom-up according to the structure of the formula, and applying the operation corresponding to each connective or quantifier. At the top, we obtain the answer to ϕ, which is a relation containing only the pair $(\langle\rangle, \psi)$, where ψ is a reduced grounding of ϕ wrt $\mathscr{A}$.*

The Negation Problem. If we naively negate an extended X-relation, we often end up with a universal table. To see this, consider an extended X-relation $\mathscr{R}$. To construct its negation $\neg\mathscr{R}$, we include $(\gamma, \neg\psi)$ for every $(\gamma, \psi) \in \mathscr{R}$ with $\psi \neq \top$, and include $(\gamma, \top)$, for every γ with $\gamma \notin \mathscr{R}$. If there are no tuples $(\gamma, \top) \in \mathscr{R}$ then $\neg\mathscr{R}$ contains a pair for every possible instantiation of X. Once a universal, or nearly universal, table is constructed, all following operations must deal with it. In the following two sections, we describe two methods for mitigating this problem.

3 Formula Rewriting

In this section, we describe a method for pre-processing the specification formula before running the basic grounder. For each sub-formula ψ of ϕ, consider all possible ways to rewrite ψ using De Morgan laws, generalized to include quantifiers. Of all rewritings, we choose the one for which cost of carrying out negations is minimum.

This is a heuristic method. It is often the case that the total grounding time is dominated by the time for joins, and it is possible that by minimizing negation cost we may increase the overall grounding time because our re-writing increases the join cost. However, minimum join order is NP-hard (see, e.g., [3]), so it is unlikely that we can efficiently minimize a cost function which accurately reflects join cost. As the experiments reported in Section 6 indicate, our heuristic tends to significantly reduce grounding time in practice.

We define the cost of constructing the complement of an extended hidden X-relation $\mathscr{R}_Y$ to be $|A|^{|X\setminus Y|}$ where A is the domain. This is a reasonable approximation of the time to construct the complement, because doing so requires that we visit every instantiation of $X\setminus Y$, and do a small constant amount of work for each. We then define the negation cost of a formula, with respect to a structure $\mathscr{A}$, as the sum of the costs of the complement operations carried by the **B** grounder.

The cost of the complement operation corresponding to the negation of a subformula ϕ is primarily determined by the number of variables that are *not* hidden in the answer to ϕ constructed by the **B** grounder. These are the variables that are free in ϕ and also occur as arguments in atomic subformula $P(\bar{x})$ of ϕ, where P is an instance predicate. (Variables that occur only as arguments to predicate symbols of the expansion vocabulary remain hidden.) Let $nhv(\phi)$ denote this set of variables. The cost of complementing the answer to ϕ is essentially the size of a universal relation of arity $|nhv(\phi)|$. Thus, we define the function $\text{Size}_{\mathscr{A}}(\phi)$ to be $|\mathscr{A}|^{|nhv(\phi)|}$.

Now, we define the *negation cost* of formula ϕ with respect to structure $\mathscr{A}$, denoted $\text{Cost}_{\mathscr{A}}(\phi)$, as:

$$\text{Cost}_{\mathscr{A}}(\phi) = \begin{cases} 0 & \text{if } \phi \text{ is atomic,} \\ \text{Size}_{\mathscr{A}}(\phi) + \text{Cost}_{\mathscr{A}}(\alpha) & \text{if } \phi \text{ is } \neg\alpha, \\ \text{Cost}_{\mathscr{A}}(\alpha) + \text{Cost}_{\mathscr{A}}(\beta) & \text{if } \phi \text{ is } \alpha\{\wedge,\vee\}\beta, \\ \text{Cost}_{\mathscr{A}}(\alpha) & \phi \text{ is } \forall\bar{x}\alpha \text{ or } \exists\bar{x}\alpha. \end{cases}$$

The negation cost is the number of tuples that are visited by the B grounder *while performing complement operations*. (Thus, the cost of atoms is zero.)

Our goal is to produce a formula ψ equivalent to ϕ but with minimum negation cost, by transforming ϕ according to standard equivalences. To be precise, we define the set $\text{Rewitings}(\phi)$ to be the set of formulas which is the closure of set $\{\phi\}$ under application of De Morgan's laws and the equivalences $(\forall x\,\alpha) \equiv (\neg\exists x\,\neg\alpha)$, and $(\exists x\,\alpha) \equiv (\neg\forall x\,\neg\alpha)$. To ground ϕ, we will apply a formula-rewriting function, $FR_{\mathscr{A}}(\psi)$, that maps ϕ to the formula in $\text{Rewitings}(\phi)$ with minimum negation cost.

The size of the set $\text{Rewitings}(\phi)$ is easily seen to be exponential in the size of ϕ, but we can compute $FR_{\mathscr{A}}(\phi)$ efficiently by dynamic programming. Algorithm (1) computes the minimum cost of a formula in $\text{Rewitings}(\phi)$. That is, $MinCost_{\mathscr{A}}(\psi) = \text{Cost}_{\mathscr{A}}(FR_{\mathscr{A}}(\psi))$ for each subformula ψ of ϕ.

Input: Subformula ψ
Output: Minimum costs P,N for $\psi,\neg\psi$ respectively.
`/* Throughout, P_α and N_α denote the values returned by`
`MinCost_A(α). */`
if $\psi \equiv \alpha \wedge \beta$ *or* $\psi \equiv \alpha \vee \beta$ **then**
  $P \leftarrow min(P_\alpha + P_\beta, Size_{\mathscr{A}}(\psi) + N_\alpha + N_\beta)$;
  $N \leftarrow min(Size_{\mathscr{A}}(\psi) + P_\alpha + P_\beta, N_\alpha + N_\beta)$;
else if $\psi \equiv \neg\alpha$ **then**
  $P \leftarrow min(Size_{\mathscr{A}}(\psi) + P_\alpha, N_\alpha)$;
  $N \leftarrow min(P_\alpha, Size_{\mathscr{A}}(\psi) + N_\alpha)$;
else if $\psi \equiv \forall\bar{x}\alpha$ *or* $\psi \equiv \exists\bar{x}\alpha$ **then**
  $P \leftarrow min(P_\alpha, Size_{\mathscr{A}}(\psi) + N_\alpha)$;
  $N \leftarrow min(Size_{\mathscr{A}}(\psi) + P_\alpha, N_\alpha)$;
else if ψ *is atomic* **then** $\langle P,N\rangle \leftarrow \langle 0, Size_{\mathscr{A}}(\psi)\rangle$;
return $\langle P,N\rangle$;

Algorithm 1. The algorithm $MinCost_{\mathscr{A}}(\psi)$ giving minimum costs of computing answers to ψ and $\neg\psi$.

The algorithm to produce the formula $FR_{\mathscr{A}}(\phi)$, is a simple variant of Algorithm (1), in the usual manner for dynamic programming. The algorithm runs in time $O(|\phi|)$.

Proposition 3. *1) Any reduced grounding of $FR_{\mathscr{A}}(\phi)$ wrt $\mathscr{A}$ is a reduced grounding of ϕ wrt $\mathscr{A}$;*
2) The formula in Rewitings(ϕ) *with minimum negation cost can be found in time $O(|\phi|)$.*

Definition 8. ***Formula Re-writing Method (FR):*** *To ground a sentence ϕ wrt $\mathscr{A}$, we compute $FR_{\mathscr{A}}(\phi)$, and then ground $FR_{\mathscr{A}}(\phi)$ wrt $\mathscr{A}$ using the B grounder.*

4 T/F Relational Algebra

Another way to tackle the negation problem is to modify the algebra so that the complement operation can be cheap. Here, we do this by adding a new relation type. We call extended X-relations as defined previously F relations, and the new relation T relations. Absence of a tuple from an F relation is equivalent to being paired with the formula $\bot$, whereas absence from a T relation is equivalent to being paired with $\top$. To construct the complement of an extended X-relation $\mathscr{R}$, we negate each formula occurring in a pair in $\mathscr{R}$, and then change the type of $\mathscr{R}$ (to T if it was F, and vice versa). Thus, the complement operation is linear time, and does not change the number of tuples. All the other operations are adapted to this setting, and in practice their application tends to generate smaller tables than when using the algebra with only F relations.

Definition 9 (T and F relations). *A* default-false extended X-relation *(F relation) is an extended X-relation as defined in Definition 3, extended with a flag that is set to False. A* default-true extended X-relation *(T relation) is an extended X-relation as defined in Definition 3, extended with a flag that is set to True, and with $\delta_{\mathscr{R}}$ defined as:*

$$\delta_{\mathscr{R}}(\gamma) = \begin{cases} \psi & \text{if } \langle \gamma, \psi \rangle \in \mathscr{R}, \\ \top & \text{if there is no pair } \langle \gamma, \psi \rangle \in \mathscr{R}. \end{cases}$$

Rather than explicitly identifying the type of every extended X-relation, we often simply write $\mathscr{R}^F$ if $\mathscr{R}$ is an F relation, and $\mathscr{R}^T$ if $\mathscr{R}$ is a T relation. When the type is clear from the context, or does not matter, we may omit the superscript. The definition of an answer to formula ϕ wrt $\mathscr{A}$ of Definition 4 applies to both T and F extended X-relations.

The operations on F tables to produce F tables are just those for regular extended X-relations, as defined in Definition 5. There are many possible additional operations when we consider both T and F relations. For example, a join may be applied to two F relations, two T relations, or one of each, and in each case we may choose to produce either an F table or a T table. We have adopted a simple heuristic strategy: For each choice of operation and the types of its argument(s), we make a fixed choice to produce either a T or F table. The full set of operations used is as follows.

Definition 10 (Algebra for T and F relations). *For each operation of Definition 5, except complement, we have an equivalently defined operation mapping F relations to F relations. In addition, we have the following. Let $\mathscr{R}^F$ and $\mathscr{R}^T$ be extended X-relations and $\mathscr{S}^F$ and $\mathscr{S}^T$ extended Y-relations, both over domain A. Then*

1. *$\neg\mathscr{R}^F$ is the extended X-relation*

$$\{(\gamma,\psi) \mid \gamma : X \to A, \gamma \in \mathscr{R}^F \text{ and } \psi = \neg\delta_{\mathscr{R}}(\gamma)\}.$$

   $\neg\mathscr{R}^T$ is defined dually.
2. *(a) $\mathscr{R}^F \bowtie \mathscr{S}^T$ is the extended $X \cup Y$-relation*

$$\{(\gamma,\psi) \mid \gamma : X \cup Y \to A,\ \gamma|_X \in \mathscr{R}^F \text{ and } \psi = \delta_{\mathscr{R}}(\gamma|_X) \wedge \delta_{\mathscr{S}}(\gamma|_Y)\}$$

   (b) $\mathscr{R}^T \bowtie \mathscr{S}^T$ is the extended $X \cup Y$-relation

$$\{(\gamma,\psi) \mid \gamma : X \cup Y \to A,\ (\gamma|_X \in \mathscr{R}^T \text{ or } \gamma|_Y \in \mathscr{S}^T) \text{ and } \psi = \delta_{\mathscr{R}}(\gamma|_X) \wedge \delta_{\mathscr{S}}(\gamma|_Y)\}$$

3. *(a) $\mathscr{R}^T \cup \mathscr{S}^T$ is the extended $X \cup Y$-relation*

$$\{(\gamma,\psi) \mid \gamma : X \cup Y \to A,\ \gamma|_X \in \mathscr{R}^T, \gamma|_Y \in \mathscr{S}^T \text{ and } \psi = \delta_{\mathscr{R}}(\gamma|_X) \vee \delta_{\mathscr{S}}(\gamma|_Y)\}$$

   (b) $\mathscr{R}^T \cup \mathscr{S}^F$ is the extended $X \cup Y$-relation

$$\{(\gamma,\psi) \mid \gamma : X \cup Y \to A,\ \gamma|_X \in \mathscr{R}^T \text{ and } \psi = \delta_{\mathscr{R}}(\gamma|_X) \vee \delta_{\mathscr{S}}(\gamma|_Y)\}$$

4. *The Y-*projection *of $\mathscr{R}^T$, denoted $\pi_Y(\mathscr{R}^T)$, is the extended Y-relation:*

$$\{(\gamma,\psi) \mid \gamma' \in \mathscr{R} \text{ for every } \gamma' \text{ with } \gamma'|_Y = \gamma \text{ and } \psi = \bigvee_{\{\gamma' \in \mathscr{R} \mid \gamma'|_Y = \gamma\}} \delta_{\mathscr{R}}(\gamma')\}$$

5. *The Y-*quotient *of $\mathscr{R}^T$, denoted by $d_Y(\mathscr{R}^T)$, is the extended Y-relation*

$$\{(\gamma,\psi) \mid \gamma = \gamma'|_Y \text{ for some } \gamma' \in \mathscr{R}^T \text{ and } \psi = \bigwedge_{\{\gamma' \in \mathscr{R} \mid \gamma'|_Y = \gamma\}} \delta_{\mathscr{R}}(\gamma')\}.$$

Proposition 4. *Suppose that $\mathscr{R}_1^F, \mathscr{R}_2^T, \mathscr{R}_3^T$ are answers to ϕ_1, ϕ_2, ϕ_3, respectively, all wrt structure $\mathscr{A}$. Then*

1. *$\neg\mathscr{R}_1^F$ is an answer to $\neg\phi_1$ wrt $\mathscr{A}$, and $\neg\mathscr{R}_2^T$ is an answer to $\neg\phi_2$ wrt $\mathscr{A}$.*
2. *(a) $\mathscr{R}_1^F \bowtie \mathscr{R}_2^T$ is an answer to $\phi_1 \wedge \phi_2$ wrt $\mathscr{A}$.*
   (b) $\mathscr{R}_2^T \bowtie \mathscr{R}_3^T$ is an answer to $\phi_2 \wedge \phi_3$ wrt $\mathscr{A}$.
3. *If Y is the set of free variables of $\exists\bar{z}\phi_2$, then $\pi_Y(\mathscr{R}_2^T)$ is an answer to $\exists\bar{z}\phi_2$ wrt $\mathscr{A}$.*
4. *If Y is the set of free variables of $\forall\bar{z}\phi_2$, then $d_Y(\mathscr{R}_2^T)$ is an answer to $\forall\bar{z}\phi_2$ wrt $\mathscr{A}$.*
5. *(a) $\mathscr{R}_2^T \cup \mathscr{R}_3^T$ is an answer to $\phi_2 \vee \phi_3$ wrt $\mathscr{A}$.*
   (b) $\mathscr{R}_1^F \cup \mathscr{R}_2^T$ is an answer to $\phi_1 \vee \phi_2$ wrt $\mathscr{A}$.

The proofs are straightforward, and generalize those for the standard version.

Definition 11. ***True/False Table Method (T/F):*** *To ground a sentence ϕ wrt $\mathscr{A}$, we construct an F table as the answer to each atomic formula, and then apply the algebra bottom-up according to the structure of the formula.*

It is, perhaps, interesting to observe that we can use a T/F algebra with all possible operations, and in polynomial time compute the optimum choice for which operation to apply at each connective. However, we are not able to use this fact to speed up grounding

in practice, as any method we see to carry out this computation requires as much work as constructing a grounding. Hence our heuristic application of T and F tables.

Indeed, it can be shown both that there are cases where the B method performs better than our T/F method, and also where algebras with additional operations perform better. Example 3 illustrates this latter case. Nonetheless, as our empirical results in Section 6 illustrate, the method tends to work well in practice.

Example 3. This example shows that by using a larger set of operations for T and F tables we may improve on the performance of our T/F method. Let ϕ be the formula $\neg R(x,y) \wedge \neg S(x,y)$, and $\mathscr{A}$ be the structure with domain $A = \{1,2,3\}$ over vocabulary $\{R,S\}$, where

$$R^{\mathscr{A}} = \{(1,1),(1,2),(2,1),(2,2),(3,1),(3,2)\} \text{ and } S^{\mathscr{A}} = \{(1,3),(2,3),(3,3)\}.$$

The T/F method takes the following steps:

1. $T_1^T(\mathscr{A}) = \neg R^F(\mathscr{A})$, hence $T_1^T(\mathscr{A})$ is: $\{ (1,1,\bot) ,(1,2,\bot),(2,1,\bot),(2,2,\bot), (3,1,\bot), (3,2,\bot) \}$
2. $T_2^T(\mathscr{A}) = \neg S^F(\mathscr{A})$, hence $T_2^F(\mathscr{A})$ is: $\{(1,3,\bot),(2,3,\bot),(3,3,\bot)\}$
3. $T_3^T(\mathscr{A}) = T_1^T(\mathscr{A}) \bowtie T_2^T(\mathscr{A})$, hence $T_3^T(\mathscr{A})$ is: $\{(1,1,\bot),(1,2,\bot),(1,3,\bot), (2,1,\bot),(2,2,\bot), (2,3,\bot),(3,1,\bot),(3,2,\bot),(3,3,\bot)\}$

However, an alternative evaluation could be:

1. $T_1^F(\mathscr{A}) = \neg R^F(\mathscr{A})$, hence $T_1^F(\mathscr{A})$ is: $\{(1,3,\top),(2,3,\top),(3,3,\top)\}$
2. $T_2^T(\mathscr{A}) = \neg S^F(\mathscr{A})$, hence $T_2^F(\mathscr{A})$ is: $\{(1,3,\bot),(2,3,\bot),(3,3,\bot)\}$
3. $T_3^F(\mathscr{A}) = T_1^F(\mathscr{A}) \bowtie T_2^T(\mathscr{A})$, hence $T_3^F(\mathscr{A})$ is: $\{\}$

5 Comparing the Methods: An Example

To illustrate the differences between the basic grounder and the methods FR and T/F, we use the following axiom from Example 1, a specification of 3-Colouring, (re-writing the implication as a disjunction)

$$\forall x \forall y \ (\neg E(x,y) \vee \neg (R(x) \wedge R(y))). \tag{1}$$

Here, $E(x,y)$ is an instance predicate and $R(x)$, for red, is an expansion predicate. We consider the steps taken by each of the three algorithms on the graph $(V;E)$ with

$$V = \{1,\ldots,5\}, \quad E = \{(1,2),(1,3),(1,4),(1,5),(2,1),(3,1),(4,1),(5,1)\}.$$

The Basic Grounder first negates $E(x,y)$ producing a relation $T_1(x,y)$ of size 17 with all the tuples not occurring in $E(x,y)$. Then, from the subformula $\neg(R(x) \wedge R(y))$ produces a relation $T_2(x,y)$ with both variables x and y hidden (because they occur only in expansion predicates). T_2 contains a single pair consisting of the empty tuple and formula $\neg(R(x) \wedge R(y))$. The relations T_1 and T_2 are unioned to produce $T_3(x,y)$. T_3 contains any tuple occurring in either T_1 or T_2. Since x and y are hidden in T_2, it implicitly contains all possible combinations of x and y. Thus, T_3 contains 25 tuples where 17

of them (the ones present in T_1) are paired with formula $\top$ and the others are paired with formula $\neg(R(m) \wedge R(n))$ (with appropriate constant symbols m and n). Then, quotient operator is applied to T_3 to give us a formula (which is then converted to CNF).

The Formula Rewriting method first converts formula (1) to the following formula which minimizes the negation cost:

$$\neg\exists x \exists y \ (E(x,y) \wedge (R(x) \wedge R(y))) \tag{2}$$

The basic grounder is run on the rewritten formula. The first operation is a join to compute an answer to $R(x) \wedge R(y)$, which produces the extended relation $T_1(x,y)$ with both x and y hidden. It has a single pair consisting of the empty tuple and formula $R(x) \wedge R(y)$. Then, E and T_1 are joined to form T_2. As T_2 includes only the tuples present in both E and T_1, it only has the 8 tuples in E. Then, projections are applied producing a zero-ary relation T_3 with only one tuple. Finally, the complement of T_3 is constructed. Since there are no free variables, this operation involves only negating the formula paired with the single tuple in T_3.

The T/F tables method first complements the table for E, producing a T relation $T_1^T(x,y)$ having the same 8 tuples as in E, but the formula paired with each tuple negated (i.e., $\top$ replaced by $\bot$). Then the answer to $\neg(R(x) \wedge R(y))$ is constructed. This is the T relation $T_2^T(x,y)$ with both x and y hidden and formula $\neg(R(x) \wedge R(y))$ paired with the single empty tuple. Then, the union operation is applied on two T relations T_1^T and T_2^T, and a T relation T_3^T is generated. It contains only those tuples that appear in both T_1^T and T_2^T, and therefore contains only the 8 tuples of T_1^T. Finally the quotient is applied to T_3^T to generate the final result.

This example illustrates that the use of either FR or T/F methods can significantly reduce the size of intermediate tables produced during grounding. Such a large benefit is not obtained for all formulas, however, formulas with an instance relation guarding the universal quantifiers, as in the examle, are common.

For this example, the choice of instance structure does not change the result. The basic method takes $O(|V|^2)$ steps for grounding formula 1 while the two other methods take $O(|E|)$ steps for the same task, which is typically much smaller. However, in general the benefits obtained by the new methods are not always independent of the instance structure.

6 Experimental Evaluation

In this section we report an experimental evaluation of the methods. We have implemented a grounder based on the extended relational algebra which can be run in three modes: mode B; mode FR, and mode T/F, corresponding to the three methods of Definitions 7, 8 and 11. We present the results for four problems, Graph Colouring, Hamiltonian Cycle, Blocked Queens, and Latin Square Completion. These represent a variety of combinatorial structures, are easily modelled in a pure FO language, and illustrate the variation in the relative performance of the present grounder relative to similar grounders.

Table 1. Comparision of grounding times for the three modes of our grounder, basic (B), formula re-writing (FR) and True-False approach (T/F). We give mean times for all instances in each benchmark set, in units of 1/10 seconds. Numbers in parentheses are the speedup factor. The best time for each problem is presented in bold.

Problem	B	FR	T/F
Graph Col.	33.7	7.5 (4.5)	**6.3** (5.3)
Ham. Cycle	992	**129** (7.7)	150 (6.6)
Blocked Queens	36.9	**3.7** (10)	4.1 (9.0)
Latin Square Comp.	273	**25.7** (10)	31.8 (8.6)

Table 1 gives the mean grounding times for each mode of operation on each of the problems. The total number of instances represented is 187, as follows: Graph colouring (17 instances); Hamiltonian Cycle (30 instances); Blocked Queens (40 instances); Latin Square Completion (100 instances). The instances are from the Asparagus repository [1]; all instances and axiomatizations are available online at http://www.cs.sfu.ca/research/groups/mxp/examples/. The times are for running on a Sun Java workstation with an Opteron 250 cpu, with 2GB of RAM, running Suse Enterprise Linux 2.6.11.4. The code is written in C++ and compiled with g++ version 3.3.5, on the same machine.

Both FR and T/F are substantially faster than B on all problems we looked at, including others not reported here. In general, we expect the T/F approach to be more powerful than FR and we attribute the slightly better performance of FR over T/F here to the fact that the implementation of FR has been somewhat optimized, while that of T/F has not been.

We also compared the speed of our grounder with other grounders for FO MX, namely GidL [10] and MXG [8], on the same problem instances. The grounding time comparison is shown in Table 2. GidL was run in two modes: with approximation on (G+A), and off (G-A). The approximation method attempts to reduce the size of grounding, and sometimes also affects the speed.

MXG and both versions of GidL failed to ground any instance of Hamiltonian Cycle using the (very standard) axiomatization we used for our grounder. To report running times for these, we modified the axioms by pushing leading quantifiers in as far as possible. (Our grounder does this implicitly.)

Table 2. Comparision of grounding times for our grounder with GidL and MXG. The columns are: GidL with approximation on (G+A), GidL with approximation off (G-A), MXG, and our new grounder with formula re-writing (FR) and T/F tables (T/F). Values are mean times for all instances of each problem, in 1/10-ths of seconds. The best time for each problem is presented in bold.

Problem	G-A	G+A	MXG	FR	T/F
Graph-Col.	10.1	10.2	16.7	7.5	**6.3**
Ham. Cyc.	>1200	603	578.7	**129**	150
Bl. Queens	97.3	**1.3**	25.4	3.7	4.1
Latin Sq. C.	**16.9**	30.2	764	25.7	31.8

Remark 1. The languages of GidL and MXG allow (to different degrees) aggregate operations and inductive definitions. We used pure FO axiomatizations without these here, as the handling of these special forms is a distinct issue.

No grounder dominates in this comparison: Each of G+A, G-A, FR and T/F have minimum mean times for one problem. We also observe that FR appears relatively robust: it does not rank worse than second on any of the problems, whereas G+A ranks third once and each of the others ranks third or worse at least twice.

Our grounder with method B, in comparison with the other relational algebra based grounder MXG, is somewhat slower on three problems and somewhat faster on one. MXG includes some techniques that improve on the method B, and GidL includes a number of techniques to improve speed over naive substitution.

7 Conclusions and Future Work

We have described and implemented two speed-up techniques for grounding based on extended relational algebra. We presented an experimental evaluation, in which the methods exhibited speedup factors of 4 to 10 times. Speedups on other problems we have tried are similar, so we conclude that the methods are effective and probably useful in practice.

Future work on the methods reported here includes:

- A more sophisticated T/F table method. Under most conditions, T/F tables are strictly more powerful than B and FR, but this power is not fully exploited in our current T/F method. We plan to develop a stronger method for T/F tables, probably involving additional operations, and possibly in combination with FR.
- Strategies that effectively combine the techniques reported here with the adaptations of standard query optimization methods such as join ordering and pipelining.

We believe that extended relational algebra has great potential for developing efficient grounders. Since control is distinct from the basic operations, many algorithmic techniques may be applied, possibly in surprising combinations, and advanced database query optimization techniques can be adapted to grounding. We have described the grounding method, and the specific methods introduce in this paper, in terms of FO and reduction to propositional logic, but the ideas can be used for many other choices of languages.

Acknowledgements

The authors are grateful to the Natural Sciences and Engineering Research Council of Canada (NSERC), MITACS and D-Wave Systems for financial support. we also appreciate the anonymous reviewers' insightful comments.

References

1. The Asparagus Library of Examples for ASP Programs, `http://asparagus.cs.uni-potsdam.de/`
2. Cadoli, M., Schaerf, A.: Compiling problem specifications into SAT. Artificial Intelligence 162, 89–120 (2005)

3. Cluet, S., Moerkotte, G.: On the complexity of generating optimal left-deep processing trees with cross products. In: Y. Vardi, M., Gottlob, G. (eds.) ICDT 1995. LNCS, vol. 893, pp. 54–67. Springer, Heidelberg (1995)
4. Fagin, R.: Generalized first-order spectra and polynomial-time recognizable sets. In: SIAM-AMC Proceedings of Complexity of Computation, vol. 7, pp. 43–73 (1974)
5. Frisch, A.M., Harvey, W., Jefferson, C., Hernández, B.M., Miguel, I.: Essence: A constraint language for specifying combinatorial problems. Constraints 13(3), 268–306 (2008)
6. Gebser, M., Schaub, T., Thiele, S.: Gringo: A new grounder for answer set programming. In: Baral, C., Brewka, G., Schlipf, J. (eds.) LPNMR 2007. LNCS (LNAI), vol. 4483, pp. 266–271. Springer, Heidelberg (2007)
7. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The dlv system for knowledge representation and reasoning. ACM Trans. Comput. Log. 7(3), 499–562 (2006)
8. Mohebali, R.: A method for solving np search based on model expansion and grounding. Master's thesis, Simon Fraser University (2006)
9. Patterson, M., Liu, Y., Ternovska, E., Gupta, A.: Grounding for model expansion in k-guarded formulas with inductive definitions. In: Proc. IJCAI 2007, pp. 161–166 (2007)
10. Wittocx, J., Mariën, M., Denecker, M.: Grounding with bounds. In: Proc. AAAI 2008, pp. 572–577 (2008)

Constraint-Based Abstract Semantics for Temporal Logic: A Direct Approach to Design and Implementation*

Gourinath Banda[1] and John P. Gallagher[1,2]

[1] Roskilde University, Denmark
[2] IMDEA Software, Madrid
{gnbanda,jpg}@ruc.dk

Abstract. Abstract interpretation provides a practical approach to verifying properties of infinite-state systems. We apply the framework of abstract interpretation to derive an abstract semantic function for the modal μ-calculus, which is the basis for abstract model checking. The abstract semantic function is constructed directly from the standard concrete semantics together with a Galois connection between the concrete state-space and an abstract domain. There is no need for mixed or modal transition systems to abstract arbitrary temporal properties, as in previous work in the area of abstract model checking. Using the modal μ-calculus to implement CTL, the abstract semantics gives an over-approximation of the set of states in which an arbitrary CTL formula holds. Then we show that this leads directly to an effective implementation of an abstract model checking algorithm for CTL using abstract domains based on linear constraints. The implementation of the abstract semantic function makes use of an SMT solver. We describe an implemented system for proving properties of linear hybrid automata and give some experimental results.

1 Introduction

In this paper we apply the framework of abstract interpretation [12] to design and implement an abstraction of temporal logic, based on linear constraints. We emphasise firstly that abstraction of the concrete semantics of a language such as the modal μ-calculus or CTL gives safe approximations for arbitrary formulas. Some other previous approaches handle only universal formulas (e.g. [11]). Secondly, we do not need to introduce extra conceptual apparatus in the semantics such as mixed or modal transition systems (e.g. [16,25]) in order to approximate the meaning of arbitrary formulas. Thirdly we show that the abstract semantics can be directly implemented, for domains based on constraints, using a constraint solver (a convex polyhedra library) and satisfiability checker

* Work partly supported by the Danish Natural Science Research Council project *SAFT: Static Analysis Using Finite Tree Automata*.

E.M. Clarke and A. Voronkov (Eds.): LPAR-16, LNAI 6355, pp. 27–45, 2010.

(an SMT solver) and applied to prove properties of real-time systems modelled as linear hybrid automata.

The present work is part of an attempt to develop a uniform constraint-based formal modelling and verification framework for verifying infinite state reactive systems. The modelling part of this framework was considered in [3] where it was shown how to model linear hybrid automata (LHA) specifications as constraint logic programs. However the techniques described in the present work are not limited to constraint-based models and properties but applies to any abstraction in the framework of abstract interpretation. This work is also orthogonal to other highly interesting and relevant areas of abstract model checking such as abstract domain construction and the refinement of abstractions. We believe that basing our presentation based on completely standard semantics and abstract interpretation techniques will facilitate cross-fertilisation of techniques from abstract interpretation and model checking.

The structure of this paper is as follows. Section 2 reviews the syntax and semantics of the modal μ-calculus and recalls how this can be used to define the semantics of typical temporal property languages such as CTL. The theory of abstract interpretation is then outlined. Section 3 describes abstract interpretation of the μ-calculus semantic function, providing a basis for abstract model checking. Section 4 shows how to define an abstraction based on linear constraints. Section 5 describes an implementation of the constraint-based abstract semantics and Section 6 gives some experimental results. In Sections 7 and 8 we discuss related work and conclusions.

2 Preliminaries

The (propositional modal) μ-calculus "provides a single, simple and uniform framework subsuming most other logics of interest for reasoning about reactive systems" [20]. The set of μ-calculus formulas is defined by the following grammar.

$$\phi ::= p \mid \neg p \mid Z \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid AX\phi \mid EX\phi \mid \mu Z.\phi \mid \nu Z.\phi$$

where p ranges over a set of atomic formulas $\mathcal{P}$ and Z ranges over a set of propositional variables $\mathcal{V}$. Note that negations can appear only before propositions p. This form is called *negation normal form*; if we extend the grammar to allow expressions of the form $\neg\phi$ in which occurrences of Z in ϕ in formulas fall under an even number of negations, an equivalent formula in negated normal form can be obtained using rewrites (which are justified by the semantics below), namely $\neg\mu Z.\phi \Rightarrow \nu Z.\neg\phi$, $\neg\nu Z.\phi \Rightarrow \mu Z.\neg\phi$, $\neg EX\phi \Rightarrow AX\neg\phi$, $\neg AX\phi \Rightarrow EX\neg\phi$ together with De Morgan's laws and elimination of double negation.

2.1 Semantics of μ-Calculus

There are various presentations of the semantics of the μ-calculus, e.g. [20,30]. We restrict our attention here to state-based semantics, which means that given a Kripke structure K, a μ-calculus formula evaluates to the set of states in K

at which the formula holds. The semantics is presented in a functional style similar to that given in [15] Section 9, suitable for applying abstraction. A μ-calculus formula is interpreted with respect to a Kripke structure, which is a state transition system whose states are labelled with atomic propositions that are true in that state.

Definition 1 (Kripke structure). *A Kripke structure is a tuple $\langle S, \Delta, I, L, \mathcal{P}\rangle$ where S is the set of states, $\Delta \subseteq S \times S$ is a total transition relation (i.e. every state has a successor), $I \subseteq S$ is the set of initial states, $\mathcal{P}$ is the set of propositions and $L : S \to 2^{\mathcal{P}}$ is the labelling function which returns the set of propositions that are true in each state. The set of atomic propositions is closed under negation.*

We first define some subsidiary functions that depend on K.

Definition 2. *Given a Kripke structure $K = \langle S, \Delta, I, L, \mathcal{P}\rangle$ we define functions $pre : 2^S \to 2^S$, $\widetilde{pre} : 2^S \to 2^S$ and $\mathsf{states} : \mathcal{P} \to 2^S$ as follows.*

- $pre(S') = \{s \mid \exists s' \in S' : (s, s') \in \Delta\}$ *returns the set of states having at least one of their successors in the set $S' \subseteq S$;*
- $\widetilde{pre}(S') = \mathsf{compl}(pre(\mathsf{compl}(S')))$ *returns the set of states all of whose successors are in the set $S' \subseteq S$; the function $\mathsf{compl}(X) = S \setminus X$.*
- $\mathsf{states}(p) = \{s \in S \mid p \in L(s)\}$ *returns the set of states where $p \in \mathcal{P}$ holds.*

The functions *pre* and $\widetilde{pre}$ are defined by several authors (e.g. [32,15]) and are also used with other names by other authors (e.g. they are called $pre_\exists$ and $pre_\forall$ by Huth and Ryan [30]).

Lemma 1. *pre and $\widetilde{pre}$ are monotonic.*

Let Mu be the set of μ-calculus formulas, and $\mathcal{V} \to 2^S$ be the set of environments for the free variables. The meaning of a formula is a mapping from an environment giving values to its free variables to a set of states. The semantics function $[\![.]\!]_\mu : \mathsf{Mu} \to (\mathcal{V} \to 2^S) \to 2^S$ is defined as follows.

$$
\begin{array}{ll}
[\![Z]\!]_\mu\sigma = \sigma(Z) & [\![\neg p]\!]_\mu\sigma = \mathsf{states}(\neg p) \\
[\![p]\!]_\mu\sigma = \mathsf{states}(p) & [\![\phi_1 \vee \phi_2]\!]_\mu\sigma = [\![\phi_1]\!]_\mu\sigma \ \cup \ [\![\phi_2]\!]_\mu\sigma \\
[\![EX\phi]\!]_\mu\sigma = pre([\![\phi]\!]_\mu\sigma) & [\![\phi_1 \wedge \phi_2]\!]_\mu\sigma = [\![\phi_1]\!]_\mu\sigma \ \cap \ [\![\phi_2]\!]_\mu\sigma \\
[\![AX\phi]\!]_\mu\sigma = \widetilde{pre}([\![\phi]\!]_\mu\sigma) & [\![\nu Z.\phi]\!]_\mu\sigma = \mathsf{gfp}(F) \\
[\![\mu Z.\phi]\!]_\mu\sigma = \mathsf{lfp}(F) & \qquad \text{where } F(S') = [\![\phi]\!]_\mu\sigma[Z/S'] \\
\qquad \text{where } F(S') = [\![\phi]\!]_\mu\sigma[Z/S'] &
\end{array}
$$

The expressions $\mathsf{lfp}(F)$ and $\mathsf{gfp}(F)$ return the least fixed point and greatest fixed point respectively of the monotonic function $F : 2^S \to 2^S$ on the lattice $\langle 2^S, \subseteq, \cup, \cap, S, \emptyset\rangle$. The Knaster-Tarski fixed point theorem [41] guarantees the existence of least and greatest fixed points for a monotonic function on a complete lattice, and also their constructive forms $\bigcup_{i=0}^{\infty} F^i(\emptyset)$ and $\bigcap_{i=0}^{\infty} F^i(S)$ respectively. The environment $\sigma[Z/S']$ above is such that $\sigma[Z/S']Y = S'$ if $Y = Z$ and $\sigma(Y)$ otherwise.The functions F above are monotonic due the restricted occurrence of negation symbols in negated normal form.

When a formula contains no free variables we can evaluate it in a trivial environment $\sigma_\emptyset$ in which all variables are mapped (say) to the empty set. If ϕ contains no free variables we thus define its meaning $[\![\phi]\!] = [\![\phi]\!]_\mu \sigma_\emptyset$.

2.2 CTL Syntax and Semantics

The set of CTL formulas ϕ in *negation normal form* is inductively defined by the following grammar:

$$\begin{aligned}\phi ::= & \ p \mid \neg p \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid AX\phi \mid EX\phi \mid AF\phi \mid EF\phi \mid AG\phi \mid EG\phi \\ & \mid AU(\phi_1, \phi_2) \mid EU(\phi_1, \phi_2) \mid AR(\phi_1, \phi_2) \mid ER(\phi_1, \phi_2)\end{aligned}$$

where p ranges over a set of atomic formulas $\mathcal{P}$. We assume familiarity with the intended meanings of the various temporal operators. Note that we sometimes refer to arbitrary negations $\neg\phi$ in what follows, but such formulas can be transformed to equivalent negation normal form formulas.

CTL is interpreted by translating to μ-calculus, using the following function C.

$$\begin{array}{ll|ll}
\mathsf{C}(p) & = p & \mathsf{C}(\neg p) & = \neg p \\
\mathsf{C}(EX\phi) & = EX\ \mathsf{C}(\phi) & \mathsf{C}(\phi_1 \vee \phi_2) & = \mathsf{C}(\phi_1) \vee \mathsf{C}(\phi_2) \\
\mathsf{C}(AX\phi) & = AX\ \mathsf{C}(\phi) & \mathsf{C}(\phi_1 \wedge \phi_2) & = \mathsf{C}(\phi_1) \wedge \mathsf{C}(\phi_2) \\
\mathsf{C}(EF\phi) & = \mu Z.(\mathsf{C}(\phi) \vee EX\ Z) & \mathsf{C}(ER(\phi_1, \phi_2)) & = \nu Z.(\mathsf{C}(\phi_2) \wedge (\mathsf{C}(\phi_1) \vee EX\ Z) \\
\mathsf{C}(AF\phi) & = \mu Z.(\mathsf{C}(\phi) \vee AX\ Z) & \mathsf{C}(AU(\phi_1, \phi_2)) & = \mu Z.(\mathsf{C}(\phi_2) \vee (\mathsf{C}(\phi_1) \wedge AX\ Z) \\
\mathsf{C}(AG\phi) & = \nu Z.(\mathsf{C}(\phi) \wedge AX\ Z) & \mathsf{C}(EU(\phi_1, \phi_2)) & = \mu Z.(\mathsf{C}(\phi_2) \vee (\mathsf{C}(\phi_1) \wedge EX\ Z) \\
\mathsf{C}(EG\phi) & = \nu Z.(\mathsf{C}(\phi) \wedge EX\ Z) & \mathsf{C}(AR(\phi_1, \phi_2)) & = \nu Z.(\mathsf{C}(\phi_2) \wedge (\mathsf{C}(\phi_1) \vee AX\ Z)
\end{array}$$

A semantic function for CTL is then obtained by composing the translation with the semantics of the μ-calculus. The translated formulas contain no free variables (all variables Z are introduced in the scope of a μ or ν). Thus we define the semantic function for a CTL formula ϕ as $[\![\phi]\!]_{CTL} = [\![\mathsf{C}(\phi)]\!]$.

It is of course possible to partially evaluate the translation function. In this way we obtain state semantics for CTL directly. For example, the meaning of $EF\phi$ and $AG\phi$ are:

$$\begin{aligned}[\![EF\phi]\!]_{CTL} &= \mathsf{lfp}(F) \text{ where } F(S') = [\![\phi]\!]_{CTL} \cup pre(S')) \\ [\![AG\phi]\!]_{CTL} &= \mathsf{gfp}(F) \text{ where } F(S') = [\![\phi]\!]_{CTL} \cap \widetilde{pre}(S'))\end{aligned}$$

We present the semantics via μ-calculus to emphasise the generality of the approach; we can now restrict our attention to considering μ-calculus semantics.

2.3 Model Checking

Model checking consists of checking whether the Kripke structure K possesses a property ϕ, written $K \models \phi$. This is defined to be true iff $I \subseteq [\![\phi]\!]$, where I is the set of initial states, or equivalently, that $I \cap [\![\neg\phi]\!] = \emptyset$. (Note that $\neg\phi$ should be converted to negation normal form). Thus model-checking requires implementing the μ-calculus semantics function. Specifically, the implementation of

the expressions $\mathsf{lfp}(F)$ and $\mathsf{gfp}(F)$ is performed by computing a Kleene sequence $F^i(\emptyset)$ or $F^i(S)$ respectively, iterating until the values stabilise.

When the state-space powerset 2^S has infinite $\subseteq$-chains, these iterations might not terminate and hence the model checking of infinite state systems becomes undecidable. In this case we try to approximate $[\![.]\!]$ using the theory of *abstract interpretation*.

2.4 Abstract Interpretation

In abstract interpretation we develop an abstract semantic function systematically from the standard ("concrete") semantics with respect to a Galois connection. We present the formal framework briefly.

Definition 3 (Galois Connection). $\langle L, \sqsubseteq_L\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle M, \sqsubseteq_M\rangle$ *is a* Galois Connection *between the lattices* $\langle L, \sqsubseteq_L\rangle$ *and* $\langle M, \sqsubseteq_M\rangle$ *if and only if* $\alpha : L \to M$ *and* $\gamma : M \to L$ *are monotonic and* $\forall l \in L, m \in M, \alpha(l) \sqsubseteq_M m \Leftrightarrow l \sqsubseteq_L \gamma(m)$.

In abstract interpretation, $\langle L, \sqsubseteq_L\rangle$ and $\langle M, \sqsubseteq_M\rangle$ are the concrete and abstract semantic domains respectively. Given a Galois connection $\langle L, \sqsubseteq_L\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle M, \sqsubseteq_M\rangle$ and a monotonic concrete semantics function $f : L \to L$, then we define an abstract semantic function $f^\sharp : M \to M$ such that for all $m \in M$, $(\alpha \circ f \circ \gamma)(m) \sqsubseteq_M f^\sharp(m)$. Furthermore it can be shown that $\mathsf{lfp}(f) \sqsubseteq_L \gamma(\mathsf{lfp}(f^\sharp))$ and that $\mathsf{gfp}(f) \sqsubseteq_L \gamma(\mathsf{gfp}(f^\sharp))$ where $\mathsf{lfp}(f), \mathsf{gfp}(f)$ are the least and greatest fixed points respectively of f.

Thus the abstract function $f^\sharp$ can be used to compute over-approximations of f, and it can be interpreted using the γ function. The case where the abstract semantic function is defined as $f^\sharp = (\alpha \circ f \circ \gamma)$ gives the most precise approximation with respect to the Galois connection. We next apply this general framework to abstraction of the μ-calculus semantics, and illustrate with a specific abstraction in Section 4.

3 Abstract Interpretation of μ-Calculus Semantics

We consider abstractions based on Galois connections $\langle 2^S, \subseteq\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle 2^A, \subseteq\rangle$, where the abstract domain 2^A consists of sets of abstract states. In fact the abstract domain could be any lattice but for the purposes of this paper we consider such state-based abstractions, which will be further discussed in Section 4.

Definition 4. *Let* $pre : 2^S \to 2^S$, $\widetilde{pre} : 2^S \to 2^S$, *and* $\mathsf{states} : \mathcal{P} \to 2^S$ *be the functions defined in Definition 2. Given a Galois connection* $\langle 2^S, \subseteq\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle 2^A, \subseteq\rangle$, *we define* $apre : 2^A \to 2^A$, $\widetilde{apre} : 2^A \to 2^A$ *and* $\mathsf{astates} : \mathcal{P} \to 2^A$ *as*

$$apre = \alpha \circ pre \circ \gamma \qquad \widetilde{apre} = \alpha \circ \widetilde{pre} \circ \gamma \qquad \mathsf{astates} = \alpha \circ \mathsf{states}$$

The properties of Galois connections imply that for all $S' \subseteq S$, $\alpha(pre(S')) \subseteq apre(\alpha(S'))$ and $\alpha(\widetilde{pre}(S')) \subseteq \widetilde{apre}(\alpha(S'))$. We simply substitute $apre$, $\widetilde{apre}$ and $\mathsf{astates}$ for their concrete counterparts in the μ-calculus semantic function to obtain abstract semantics for the μ-calculus.

Given a Galois connection $\langle 2^S, \subseteq\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle 2^A, \subseteq\rangle$, the abstract μ-calculus semantic function $[\![\cdot]\!]^a_\mu : \mathsf{Mu} \to (\mathcal{V} \to 2^A) \to 2^A$ is defined as follows.

$$
\begin{array}{ll}
[\![Z]\!]^a_\mu\sigma & = \sigma(Z) \\
[\![p]\!]^a_\mu\sigma & = \mathsf{astates}(p) \\
[\![EX\phi]\!]^a_\mu\sigma & = apre([\![\phi]\!]^a_\mu\sigma) \\
[\![AX\phi]\!]^a_\mu\sigma & = \widetilde{apre}([\![\phi]\!]^a_\mu\sigma) \\
[\![\mu Z.\phi]\!]^a_\mu\sigma & = \mathsf{lfp}(F_a) \\
 & \quad \text{where } F_a(A') = [\![\phi]\!]^a_\mu\sigma[Z/A']
\end{array}
\qquad
\begin{array}{ll}
[\![\neg p]\!]^a_\mu\sigma & = \mathsf{astates}(\neg p) \\
[\![\phi_1 \vee \phi_2]\!]^a_\mu\sigma & = [\![\phi_1]\!]^a_\mu\sigma \cup [\![\phi_2]\!]^a_\mu\sigma \\
[\![\phi_1 \wedge \phi_2]\!]^a_\mu\sigma & = [\![\phi_1]\!]^a_\mu\sigma \cap [\![\phi_2]\!]^a_\mu\sigma \\
[\![\nu Z.\phi]\!]^a_\mu\sigma & = \mathsf{gfp}(F_a) \\
 & \quad \text{where } F_a(A') = [\![\phi]\!]^a_\mu\sigma[Z/A']
\end{array}
$$

As before, for formulas containing no free variables we define the function $[\![\phi]\!]^a = [\![\phi]\!]^a_\mu\sigma_\emptyset$. The abstract semantics for a CTL formula ϕ is $[\![\mathsf{C}(\phi)]\!]^a$.

The functions α and γ are extended to apply to environments $\sigma : \mathcal{V} \to A$. $\alpha(\sigma)$ is defined as $\alpha(\sigma)(Z) = \alpha(\sigma(Z))$ and $\gamma(\sigma)$ is defined as $\gamma(\sigma)(Z) = \gamma(\sigma(Z))$.

Theorem 1 (Safety of Abstract Semantics). *Let $K = \langle S, \Delta, I, L, \mathcal{P}\rangle$ be a Kripke structure, $\langle 2^S, \subseteq\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle 2^A, \subseteq\rangle$ be a Galois connection and ϕ any μ-calculus formula in negation normal form. Then $\alpha([\![\phi]\!]_\mu\sigma) \subseteq [\![\phi]\!]^a_\mu\alpha(\sigma)$ and $\gamma([\![\phi]\!]^a_\mu\sigma) \supseteq [\![\phi]\!]_\mu\gamma(\sigma)$, for all environments σ.*

The proof is by structural induction on ϕ. First we establish a subsidiary result.

Lemma 2. *Let $F(S') = [\![\phi]\!]_\mu\sigma[Z/S']$ and $F_a(A') = [\![\phi]\!]^a_\mu\alpha(\sigma)[Z/A']$ and let $\langle 2^S, \subseteq\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle 2^A, \subseteq\rangle$ be a Galois connection. Assume $\alpha([\![\phi]\!]_\mu\sigma) \subseteq [\![\phi]\!]^a_\mu\alpha(\sigma)$. Then for all $A' \subseteq A$, $(\alpha \circ F \circ \gamma)(A') \subseteq F_a(A')$.*

Proof.

$$
\begin{array}{lll}
(\alpha \circ F \circ \gamma)(A') & = \alpha(F(\gamma(A'))) & \\
 & = \alpha([\![\phi]\!]_\mu\sigma[Z/\gamma(A')]) & \\
 & \subseteq [\![\phi]\!]^a_\mu\alpha(\sigma)[Z/\alpha(\gamma(A'))] & \text{by assumption that} \\
 & & \alpha([\![\phi]\!]_\mu\sigma) \subseteq [\![\phi]\!]^a_\mu\alpha(\sigma) \\
 & \subseteq [\![\phi]\!]^a_\mu\alpha(\sigma)[Z/A'] & \text{by properties of Galois connections} \\
 & & \text{and monotonicity of } [\![\phi]\!]^a_\mu\alpha(\sigma) \\
 & = F_a(A') &
\end{array}
$$

The proof of Theorem 1 is just an exercise in applying the properties of Galois connections and monotonic functions. We show a few representative cases.

Proof. (Theorem 1). We show that $\alpha([\![\phi]\!]_\mu\sigma) \subseteq [\![\phi]\!]^a_\mu\alpha(\sigma)$ by structural induction on ϕ. The proof for $\gamma([\![\phi]\!]^a_\mu\sigma) \supseteq [\![\phi]\!]_\mu\gamma(\sigma)$ is similar.

Base Cases.

– $\phi = Z$.

$$\begin{aligned}\alpha([\![Z]\!]_\mu\sigma) &= \alpha(\sigma(Z))\\ &= \alpha(\sigma)(Z)\\ &= [\![Z]\!]^a_\mu\alpha(\sigma)\end{aligned}$$

– $\phi = p'$ where $p' = p$ or $p' = \neg p$

$$\begin{aligned}\alpha([\![p']\!]_\mu\sigma) &= \alpha(\mathsf{states}(p'))\\ &= \mathsf{astates}(p')\\ &= [\![p']\!]^a_\mu\alpha(\sigma)\end{aligned}$$

Inductive Cases

– $\phi = EX\phi$.

$$\begin{aligned}\alpha([\![EX\phi]\!]_\mu\sigma) &= \alpha(pre([\![\phi]\!]_\mu\sigma))\\ &\subseteq \alpha(pre(\gamma(\alpha([\![\phi]\!]_\mu\sigma)))) && \text{by Galois connection}\\ & && \text{and monotonicity of } pre, \alpha\\ &= apre(\alpha([\![\phi]\!]_\mu\sigma))\\ &\subseteq apre([\![\phi]\!]^a_\mu\alpha(\sigma)) && \text{by ind. hyp.}\\ & && \text{and monotonicity of } apre\\ &= [\![EX\phi]\!]^a_\mu\alpha(\sigma)\end{aligned}$$

– $\phi = \mu Z.\phi$.

In this case, and the case for $\phi = \nu Z.\phi$, we make use of the general property of a Galois connection $\langle 2^S, \subseteq\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle 2^A, \subseteq\rangle$ that if $f : 2^S \to 2^S$ and $f^\sharp : 2^A \to 2^A$ are such that $(\alpha \circ f \circ \gamma) \subseteq f^\sharp$, then $\alpha(\mathsf{lfp}(f)) \subseteq \mathsf{lfp}(f^\sharp)$ and $\alpha(\mathsf{gfp}(f)) \subseteq \mathsf{gfp}(f^\sharp)$. The relevant condition $(\alpha \circ f \circ \gamma) \subseteq f^\sharp$ is established in Lemma 2.

$$\begin{aligned}\alpha([\![\mu Z.\phi]\!]_\mu\sigma) &= \alpha(\mathsf{lfp}(F)) && \text{where } F(S') = [\![\phi]\!]_\mu\sigma[Z/S']\\ &\subseteq \mathsf{lfp}(F_a) && \text{where } F_a(A') = [\![\phi]\!]^a_\mu\alpha(\sigma)[Z/A']\\ & && \text{by Lemma 2, ind. hyp., and}\\ & && \text{properties of Galois connections}\\ &= [\![\mu Z.\phi]\!]^a_\mu\alpha(\sigma)\end{aligned}$$

Corollary 1. *Let $K = \langle S, \Delta, I, L, \mathcal{P}\rangle$ be a Kripke structure and ϕ be a μ-calculus formula with no free variables. Then if $\gamma([\![\neg\phi]\!]^a) \cap I = \emptyset$ then $K \models \phi$.*

Proof.

$$\begin{aligned}\gamma([\![\neg\phi]\!]^a_\mu) \cap I = \emptyset &\equiv \gamma([\![\neg\phi]\!]^a_\mu\sigma_\emptyset) \cap I = \emptyset\\ &\Rightarrow [\![\neg\phi]\!]_\mu\gamma(\sigma_\emptyset) \cap I = \emptyset && \text{by Theorem 1}\\ &\equiv [\![\neg\phi]\!] \cap I = \emptyset && \text{since } \gamma(\sigma_\emptyset) = \sigma_\emptyset\\ &\equiv I \supseteq [\![\phi]\!]\\ &K \models \phi\end{aligned}$$

This result provides us with a sound abstract model checking procedure for any μ-calculus formula ϕ. Of course, if $\gamma([\![\neg\phi]\!]^a) \cap I \supset \emptyset$ nothing can be concluded.

4 An Abstract Constraint-Based Domain

The abstract semantics given in Section 3 is not always implementable in practice for a given Galois connection $\langle 2^S, \subseteq\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle 2^A, \subseteq\rangle$. In particular, the function γ yields a value in the concrete domain, which is typically an infinite object such as an infinite set. Thus evaluating the functions $\alpha(\mathsf{states}(p))$, $(\alpha \circ pre \circ \gamma)$ and $(\alpha \circ \widetilde{pre} \circ \gamma)$ might not be feasible. In general in abstract interpretation one designs functions that safely approximate these constructions. For example one would design computable functions $apre'$ and $\widetilde{apre}'$ such that for all A, $apre'(A') \supseteq apre(A')$ and $\widetilde{apre}'(A') \supseteq \widetilde{apre}(A')$. In this section we show that the abstract semantics is implementable directly, without any further approximation, for transition systems and abstract domains expressed using linear constraints.

4.1 Abstract Domains Based on a State-Space Partitions

Consider transition systems whose states are n-tuples of real numbers; we take as the *concrete domain* the complete lattice $\langle 2^C, \subseteq\rangle$ where $C \subseteq 2^{R^n}$ is some nonempty, possibly infinite set of n-tuples including all the reachable states of the system.

We build an abstraction of the state space based on a finite partition of C say $A = \{d_1, \ldots, d_k\}$ such that $\bigcup A = C$. Such a partition could be obtained in various ways, including predicate abstraction or Moore closures (see [23] for a discussion). Define a representation function $\beta : C \to 2^A$, such that $\beta(\bar{x}) = \{d \in A \mid \bar{x} \in d\}$. We extend the representation function [36] to sets of points, obtaining the abstraction function $\alpha : 2^C \to 2^A$ given by $\alpha(S) = \bigcup\{\beta(\bar{x}) \mid \bar{x} \in S\}$. Define the concretisation function $\gamma : 2^A \to 2^C$, as $\gamma(V) = \{\bar{x} \in C \mid \beta(\bar{x}) \subseteq V\}$. As shown in [36,13], $\langle 2^C, \subseteq\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle 2^A, \subseteq\rangle$ is a Galois connection. Because A is a partition the value of $\beta(\bar{x})$ is a singleton for all $\bar{x}$, and the γ function can be written as $\gamma(V) = \bigcup\{\gamma(\{d\}) \mid d \in V\}$.

4.2 Constraint Representation of Transition Systems

We consider the set of linear arithmetic constraints (hereafter simply called constraints) over the real numbers.

$$c ::= t_1 \leq t_2 \mid t_1 < t_2 \mid c_1 \wedge c_2 \mid c_1 \vee c_2 \mid \neg c$$

where t_1, t_2 are linear arithmetic terms built from real constants, variables and the operators $+$, $*$ and $-$. The constraint $t_1 = t_2$ is an abbreviation for $t_1 \leq t_2 \wedge t_2 \leq t_1$. Note that $\neg(t_1 \leq t_2) \equiv t_2 < t_2$ and $\neg(t_1 < t_2) \equiv t_2 \leq t_2$, and so the negation symbol $\neg$ can be eliminated from constraints if desired by moving negations inwards by Boolean transformations and then applying this equivalence.

A constraint c is *satisfied* by an assignment of real numbers to its variables if the constraint evaluates to *true* under this assignment, and is *satisfiable* (written $SAT(c)$) if there exists some assignment that satisfies it. A constraint can be

identified with the set of assignments that satisfy it. Thus a constraint over n real variables represents a set of points in R^n.

A constraint can be projected onto a subset of its variables. Denote by $\mathsf{proj}_V(c)$ the projection of c onto the set of variables V.

Let us consider a transition system defined over the state-space R^n. Let $\bar{x}, \bar{x}_1, \bar{x}_2$ etc. represent n-tuples of distinct variables, and $\bar{r}, \bar{r}_1, \bar{r}_2$ etc. represent tuples of real numbers. Let $\bar{x}/\bar{r}$ represent the assignment of values $\bar{r}$ to the respective variables $\bar{x}$. We consider transition systems in which the transitions can be represented as a finite set of *transition rules* of the form $\bar{x}_1 \overset{c(\bar{x}_1,\bar{x}_2)}{\longrightarrow} \bar{x}_2$. This represents the set of all transitions from state $\bar{r}_1$ to state $\bar{r}_2$ in which the constraint $c(\bar{x}_1, \bar{x}_2)$ is satisfied by the assignment $\bar{x}_1/\bar{r}_1, \bar{x}_2/\bar{r}_2$. Such transition systems can be used to model real-time control systems [27,3].

4.3 Constraint Representation of the Semantic Functions

We consider abstract semantics based on linear partitions, so that each element d_i of the partition $A = \{d_1, \ldots, d_n\}$ is representable as a linear constraint c_{d_i}. We first provide definitions of the functions pre, $\widetilde{pre}$, α, γ and states in terms of constraint operations. In the next section the optimisation and effective implementation of these operations is considered. Let T be a finite set of transition rules. Let $c'(\bar{y})$ be a constraint over variables $\bar{y}$. We express the functions pre, $\widetilde{pre}$ and states using constraint operations as follows.

$$
\begin{aligned}
&pre(c'(\bar{y})) = \bigvee\{\mathsf{proj}_{\bar{x}}(c'(\bar{y}) \wedge c(\bar{x}, \bar{y})) \mid \bar{x} \overset{c(\bar{x},\bar{y})}{\longrightarrow} \bar{y} \in T\} \\
&\widetilde{pre}(c'(\bar{y})) = \neg(pre(\neg c'(\bar{y}))) \\
&\mathsf{states}(p) = p
\end{aligned}
$$

In the definition of states we use p both as the proposition (the argument of states) and as a set of points (the result).

The β function introduced in Section 4.1 can be rewritten as $\beta(\bar{x}) = \{d \in A \mid \bar{x} \text{ satisfies } c_d\}$. Assuming that we only need to apply α to sets of points represented by a linear constraint c, we can rewrite the α and γ functions as follows.

$$
\alpha(c) = \{d \in A \mid \mathsf{SAT}(c_d \wedge c)\} \qquad \gamma(A') = \bigvee\{c_d \mid d \in A'\} \text{ for } A' \subseteq A
$$

5 Implementation

The abstract μ-calculus semantic function $[\![.]\!]^a$ can be implemented directly as a recursive function following the definition in Section 3. The structure of the algorithm is independent of any particular abstraction. With standard iterative techniques for computing greatest and least fixpoints [7] the algorithm has the same complexity as a concrete model checker for μ-calculus. In practice the effectiveness of the algorithm as an abstract model checker depends on the effective implementation of the functions accessing the transition system and performing abstraction and concretisation, namely $pre, \widetilde{pre}, \alpha$ and γ.

5.1 Computation of α and γ Functions Using Constraint Solvers

The constraint formulations of the α and γ functions allows them to be effectively computed. The expression $\mathsf{SAT}(c_d \wedge c)$ occurring in the α function means "$(c_d \wedge c)$ is satisfiable" and can be checked by an SMT solver. In our experiments we use the SMT solver Yices [19]. The γ function simply collects a disjunction of the constraints associated with the given set of partitions; no solver is required.

5.2 Optimisation of Constraint-Based Evaluation

Combining the constraint-based evaluation of the functions pre and $\widetilde{pre}$ with the constraint-based evaluation of the α and γ functions gives us (in principle) a method of computing the abstract semantic counterparts of pre and $\widetilde{pre}$, namely $(\alpha \circ pre \circ \gamma)$ and $(\alpha \circ \widetilde{pre} \circ \gamma)$. The question we now address is the feasibility of this approach. Taken naively, the evaluation of these constraint-based functions (in particular $\widetilde{pre}$) does not scale up. We now show how we can transform these definitions to a form which can be computed much more efficiently, with the help of an SMT solver.

Consider the evaluation of $(\alpha \circ \widetilde{pre} \circ \gamma)(A')$ where $A' \in 2^A$ is a set of disjoint partitions represented by constraints.

$$\begin{aligned}(\alpha \circ \widetilde{pre} \circ \gamma)(A') &= (\alpha \circ \widetilde{pre})(\bigvee\{c_d \mid d \in A'\}) \\ &= \alpha(\neg(pre(\neg(\bigvee\{c_d \mid d \in A'\})))) \\ &= \alpha(\neg(pre(\bigvee\{c_d \in A \setminus A'\}))\end{aligned}$$

In the last step, we use the equivalence $\neg(\bigvee\{c_d \mid d \in A'\}) \leftrightarrow \bigvee\{c_d \in A \setminus A'\}$, which is justified since the abstract domain A is a disjoint partition of the concrete domain; thus $A \setminus A'$ represents the negation of A' restricted to the state space of the system. The computation of $pre(\bigvee\{c_d \in A \setminus A'\})$ is much easier to compute (with available tools) than $pre(\neg(\bigvee\{c_d \mid d \in A'\}))$. The latter requires the projection operations proj to be applied to complex expressions of the form $\mathsf{proj}_{\bar{x}}(\neg(c_1(\bar{y}) \vee \cdots \vee c_k(\bar{y})) \wedge c(\bar{x}, \bar{y}))$, which involves expanding the expression (to d.n.f. for example); by contrast the former requires evaluation of simpler expressions of the form $\mathsf{proj}_{\bar{x}}(c_d(\bar{y}) \wedge c(\bar{x}, \bar{y}))$.

We can improve the computation of the abstract function $(\alpha \circ pre \circ \gamma)$. Let $\{c_i\}$ be a set of constraints, each of which represents a set of points. It can easily seen that $pre(\bigvee\{c_i\}) = \bigvee\{pre(c_i)\}$. Consider the evaluation of $(\alpha \circ pre \circ \gamma)(A')$ where $A' \in 2^A$ is a set of disjoint partitions represented by constraints.

$$\begin{aligned}(\alpha \circ pre \circ \gamma)(A') &= (\alpha \circ pre)(\bigvee\{c_d \mid d \in A'\}) \\ &= \alpha(\bigvee\{pre(c_d) \mid d \in A'\})\end{aligned}$$

Give a finite partition A, we pre-compute the constraint $pre(c_d)$ for all $d \in A$. Let $Pre(d)$ be the predecessor constraint for partition element d. The results can be stored as a table, and whenever it is required to compute $(\alpha \circ pre \circ \gamma)(A')$ where $A' \in 2^A$, we simply evaluate $\alpha(\bigvee\{Pre(d) \mid d \in A'\})$. The abstraction function α is evaluated efficiently using the SMT solver, as already discussed.

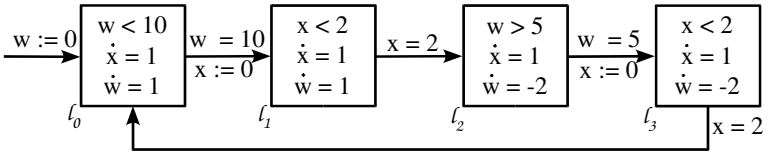


Fig. 1. A Water-level Monitor [27]

```
rState1(A,B,C,D):- rState4(E,F,G,H),
                   D=1,H=4,G≤I,1*J=1*E+1*(I-G),1*K=1*F+ -2*(I-G),
                   J=2,L=J,M=K,0≤C,1*A=1*L+1*(C-0),1*B=1*M+1*(C-0),B≤10.
rState1(A,B,C,D):- D=1,0≤C,1*A=1*0+1*(C-0),1*B=1*1+1*(C-0),B≤10.
rState2(A,B,C,D):- rState1(E,F,G,H),
                   D=2,H=1,G≤I,1*J=1*E+1*(I-G),1*K=1*F+1*(I-G),
                   K=10,L=0,M=K,0≤C,1*A=1*L+1*(C-0),1*B=1*M+1*(C-0),A≤2.
rState3(A,B,C,D):- rState2(E,F,G,H),
                   D=3,H=2,G≤I,1*J=1*E+1*(I-G),1*K=1*F+1*(I-G),J=2,
                   L=J,M=K,0≤C,1*A=1*L+1*(C-0),1*B=1*M+ -2*(C-0),B>=5.
rState4(A,B,C,D):- rState3(E,F,G,H),
                   D=4,H=3,G≤I,1*J=1*E+1*(I-G),1*K=1*F+ -2*(I-G),K=5,
                   L=0,M=K,0≤C,1*A=1*L+1*(C-0),1*B=1*M+ -2*(C-0),A≤2.
```

Fig. 2. The Water-Level Controller transition rules, automatically generated from Figure 1 [4]

Note that expressions of the form $\alpha(pre(\bigvee\{\cdots\}))$ occur in the transformed expression for $(\alpha \circ \widetilde{pre} \circ \gamma)(A')$ above. The same optimisation can be applied here too. Our experiments show that this usually yields a considerable speedup (2-3 times faster) compared to dynamically computing the *pre* function during model checking.

Our implementation of the abstract μ-calculus semantic function $[\![.]\!]^a_{CTL}$ was in Prolog, with interfaces to external libraries to perform constraint-solving functions. In implementing the *pre* operation we make use of a Ciao-Prolog interface to the PPL library [2]. In particular, this is used to compute the proj function. The α function is implemented using the SMT solver Yices [19]. We implemented an interface predicate `yices_sat(C,Xs)`, where `C` is a constraint and `Xs` is the set of variables in `C`. This predicate simply translates `C` to the syntax of Yices, and succeeds if and only if Yices finds that the constraint is satisfiable. Using this predicate the definition of α, that is $\alpha(c) = \{d \mid \mathsf{SAT}(c_d \wedge c)\}$ can be implemented directly as defined.

6 Experiments Using an SMT Constraint Solver

A small example is shown with all details to illustrate the process of proving a property. In Figure 1 is shown a linear hybrid automaton (LHA) for a water

```
region(1,rState1(A,B,C,D),[D=1,-1*A≥ -9,1*A≥0,1*A-1*B= -1,1*A-1*C=0]).
region(2,rState2(A,B,C,D),[D=2,-1*C≥ -2,1*C≥0,1*A-1*C=0,1*B-1*C=10]).
region(3,rState3(A,B,C,D),[D=3,-2*C≥ -7,1*C≥0,1*A-1*C=2,1*B+2*C=12]).
region(4,rState4(A,B,C,D),[D=4,-1*C≥ -2,1*C≥0,1*A-1*C=0,1*B+2*C=5]).
region(5,rState1(A,B,C,D),[D=1,-1*C≥ -9,1*C≥0,1*A-1*C=2,1*B-1*C=1]).
```

Fig. 3. Disjoint Regions of the Water-Level Controller States

level controller taken from [27]. Figure 2 shows transition rules represented as constraint logic program (CLP) clauses generated automatically from the LHA in Figure 1, as explained in detail in [3]. The state variables in an atomic formula of form `rState(X,W,T,L)` represent the rate of flow (`X`), the water-level (`W`), the elapsed time (`T`) and the location identifier (`L`). The meaning of a clause of form

`rState(X,W,T,L) :- rState(X1,W1,T1,L1),` c`(X,W,T,L,X1,W1,T1,L1)`

is a transition rule $(\mathtt{X1},\mathtt{W1},\mathtt{T1},\mathtt{L1}) \xrightarrow{c(\mathtt{X},\mathtt{W},\mathtt{T},\mathtt{L},\mathtt{X1},\mathtt{W1},\mathtt{T1},\mathtt{L1})} (\mathtt{X},\mathtt{W},\mathtt{T},\mathtt{L})$. The initial state is given by the clause `rState(0,0,_,1)`.

Figure 3 shows the result of an analysis of the reachable states of the system, based on computing an approximation of the minimal model of the constraint program in Figure 2. This approximation is obtained by a tool for approximating the minimal model of an arbitrary CLP program [6,28]. There are 5 *regions*, which cover the reachable states of the controller starting in the initial state (which is region 1). The term `region(N, rState(A,B,C,D),[...])` means that the region labelled `N` is defined by the constraint in the third argument, with constraint variables `A,B,C,D` corresponding to the given state variables. The 5 regions are disjoint. We use this partition to construct the abstract domain as described in Section 4.1.

Our implementation of the abstract semantics function is in Ciao-Prolog with external interfaces to the Parma Polyhedra Library [2] and the Yices SMT solver [19].[1] Using this prototype implementation we successfully checked many CTL formulas including those with CTL operators nested in various ways, which in general is not allowed in UPPAAL [5], HYTECH [29] or PHAVER [22] (though special-purpose constructs such as a "leads-to" operator can be used to handle some cases).

Table 1 gives the results of proving properties using abstract model checking two systems, namely, a water level monitor and a task scheduler. Both of these systems are taken from [27]. In the table: (i) the columns *System* and *Property* indicate the system and the formula being checked; (ii) the columns A and Δ, respectively, indicate the number of abstract regions and original transitions in a system and (iii) the column *time* indicates the computation time to prove a formula on the computer with an Intel XEON CPU running at 2.66GHz and with 4GB RAM.

Water level controller. The water level system has 4 state variables and 4 transition rules. A variety of different properties that were proved is shown in Table 1.

[1] Prolog code available at `http://akira.ruc.dk/~jpg/Software/amc_all.pl`.

In some cases we derived a more precise partition than the one originally returned for the transition system, using the technique described below in Section 6.1.

The formula $AF(W \geq 10)$ means "on all paths the water level (W) reaches at least 10", while $AG(W = 10 \rightarrow AF(W < 10 \vee W > 10))$ means "on every path the water level cannot get stuck at 10". The formula $AG(0 \leq W \wedge W \leq 12)$ is a global safety property stating that the water level remains within the bounds 0 and 12. A more precise invariant is reached some time after the initial state and this is proved by the property $AF(AG(1 \leq W \wedge W \leq 12))$, in other words "on all paths, eventually the water level remains between 1 and 12". Since for all ϕ, $AG(\phi) \equiv AG(AG(\phi))$ we are able to prove $AG^5(0 \leq W \wedge W \leq 12)$ which is only shown in order to indicate the capability of the prover to handle arbitrarily deeply nested formulas. The formula $EF(W = 10)$ shows that the water level can reach exactly 10. Finally, the formula $EU(W < 12, AU(W < 12, W \geq 12))$ shows another example of a verified progress property (which could be formulated more simply but is shown just to exercise the prover's capabilities).

Scheduler. The scheduler system for two processes has 8 state variables, 18 abstract regions and 12 transition rules. We proved a number of safety and liveness properties, again successfully checking properties of a form beyond the capability of other model checkers. For example the nested formula $AG(K2 > 0 \rightarrow AF(K2 = 0))$ is a critical correctness property meaning that tasks of high priority (whose presence is indicated by a strictly positive value of $K2$) do not get starved (that is, the value of $K2$ eventually returns to zero on all paths).

6.1 Property-Specific Refinements

The topic of refinement is a very relevant and much-studied area in proof by abstraction. Refinement is applied when a given abstraction is too coarse to prove some property. In this case we seek to derive a more precise refinement, that is somehow more relevant to the property being proved. Refinement is not the main focus in this paper, but we discuss briefly the use of a property-specific refinement that we can apply to increase the number of provable properties. Consider the property $EF(W = 10)$ in the water level controller, which holds on the system. But this formula cannot be verified when the state space is abstracted with regions that cannot distinguish states where $W = 10$. The negation of the formula, namely $AG(W > 10 \vee W < 10)$, holds in the abstract initial state since there are infinite paths from the initial region which always stay in regions that are consistent with $W \neq 10$.

One approach to solving such cases is to make a property-specific refinement to the abstraction. For a given constraint property p each region is split into further regions by adding the constraints p and $\neg p$ respectively to each region. Only the satisfiable regions need to be retained. With this refined abstraction using $(W = 10)$ for p, the property $EF(W = 10)$ can then successfully be checked.

Table 1. Experimental Results

System	*Property*	A	Δ	*Time (secs.)*
Waterlevel	$AF(W \geq 10)$	5	4	0.02
Monitor	$AG(0 \leq W \wedge W \leq 12)$	5	4	0.01
	$AF(AG(1 \leq W \wedge W \leq 12))$	5	4	0.02
	$AG(W = 10 \rightarrow AF(W < 10 \vee W > 10))$	10	4	0.05
	$AG(AG(AG(AG(AG(0 \leq W \wedge W \leq 12)))))$	5	4	0.02
	$EF(W = 10)$	10	4	0.01
	$EU(W < 12, AU(W < 12, W \geq 12))$	7	4	0.04
Task Scheduler	$EF(K2 = 1)$	18	12	0.53
	$AG(K2 > 0 \rightarrow AF(K2 = 0))$	18	12	0.30
	$AG(K2 \leq 1)$	18	12	0.04

The CEGAR approach to refinement [9] is quite compatible with our approach, but we are also interested in investigating more general forms of refinement investigated in the theory of abstract interpretation [23].

7 Related Work

The topic of *model-checking infinite state systems* [2] and using some form of abstraction has been already widely studied. Abstract model checking is described by Clarke *et al.* [10,11]. In this approach a state-based abstraction is defined where an abstract state is a set of concrete states. A state abstraction together with a concrete transition relation Δ induces an *abstract transition relation* Δ_{abs}. Specifically, if X_1, X_2 are abstract states, $(X_1, X_2) \in \Delta_{abs}$ iff $\exists x_1 \in X_1, x_2 \in X_2$ such that $(x_1, x_2) \in \Delta$. From this basis an abstract Kripke structure can be built; the initial states of the abstract Kripke structure are the abstract states that contain a concrete initial state, and the property labelling function of the abstract Kripke structure is induced straightforwardly as well. Model checking CTL properties over the abstract Kripke structure is correct for *universal* temporal formulas (ACTL), that is, formulas that do not contain operators EX, EF, EG or EU. Intuitively, the set of paths in the abstract Kripke structure represents a superset of the paths of the concrete Kripke structure. Hence, any property that holds for all paths of the abstract Kripke structure also holds in the concrete structure. If there is a finite number of abstract states, then the abstract transition relation is also finite and thus a standard (finite-state) model checker can be used to perform model-checking of ACTL properties. Checking properties containing existential path quantifiers is not sound in such an approach.

[2] When we say model checking of (continuous) infinite state systems, it means *model-checking the discrete abstractions of infinite state systems*. In [1], it is established that hybrid systems can be safely abstracted with discrete systems preserving all the temporal properties expressed in branching-time temporal logics as well as linear-time temporal logics.

This technique for abstract model checking can be reproduced in our approach, although we do not explicitly use an abstract Kripke structure. Checking an ACTL formula is done by negating the formula and transforming it to negation normal form, yielding an *existential* temporal formula (ECTL formula). Checking such a formula using our semantic function makes use of the *pre* function but not the $\widetilde{pre}$ function. For this kind of abstraction the relation on abstract states $s \rightarrow s'$ defined as $s \in (\alpha \circ pre \circ \gamma)(\{s'\})$ is identical to the abstract transition relation defined by Clarke *et al.* Note that whereas abstract model checking the ACTL formula with an abstract Kripke structure yields an under-approximation of the set of states where the formula holds, our approach yields the complement, namely an over-approximation of the set of states where the negation of the formula holds.

There have been different techniques proposed in order to overcome the restriction to universal formulas. Dams *et al.* [16] present a framework for constructing abstract interpretations for μ-calculus properties in transition systems. This involves constructing a *mixed transition system* containing two kinds of transition relations, the so-called free and constrained transitions. Godefroid *et al.* [25] proposed the use of *modal transition systems* [33] which consist of two components, namely *must*-transitions and *may*-transitions. In both [16] and [25], given an abstraction together with a concrete transition system, a mixed transition system, or an (abstract) modal transition system respectively, is automatically generated. Following this, a modified model-checking algorithm is defined in which any formula can be checked with respect to the dual transition relations. Our approach by contrast is based on the standard semantics of the μ-calculus. The *may*-transitions and the *must*-transitions of [25] could be obtained from the functions $(\alpha \circ pre \circ \gamma)$ and $(\alpha \circ \widetilde{pre} \circ \gamma)$ respectively. For the case of an abstraction given by a partition $A = \{d_1, \ldots, d_n\}$ it seems that an abstract modal transition system could be constructed with set of states A such that there is a *may*-transition $d_i \rightarrow d_j$ iff $d_i \in (\alpha \circ pre \circ \gamma)(\{d'_j\}$ and a *must*-transition $d_i \rightarrow d_j$ iff $d_i \in (\alpha \circ \widetilde{pre} \circ \gamma)(\{d_j\}$. However the two approaches are not interchangeable; in [25] a concrete modal transition system has the same set of *must*-transitions and *may*-transitions, but applying the above constructions to the concrete state-space (with α and γ as the identity function) does not yield the same sets of *must*- and *may*-transitions (unless the transition system is deterministic). We have shown that the construction of abstract transition systems as in [10,11], and abstract modal transition systems in particular [16,25] is an avoidable complication in abstraction. Probably the main motivation for the definition of abstract transition systems is to re-use existing model checkers, as remarked by Cousot and Cousot [15] (though this argument does not apply to modal or mixed transition systems in any case).

Property-preserving abstraction using Galois connections was applied in a μ-calculus setting by Loiseaux *et al.* [35]. Our aim and approach are similar, but are both more general and more direct. The cited work develops sounds abstractions for universal properties only whereas we handle arbitrary properties. On the other hand it uses Galois connections to develop a simulation relation

between concrete and abstract systems, which goes beyond the scope of the current work. The application of the theory of abstract interpretation to temporal logic, including abstract model checking, is thoroughly discussed by Cousot and Cousot [14,15]. Our abstract semantics is inspired by their approach, in that we also proceed by direct abstraction of a concrete semantic function using a Galois connection, without constructing any abstract transition relations. The technique of constructing abstract functions based on the pattern $(\alpha \circ f \circ \gamma)$, while completely standard in abstract interpretation [13], is not discussed explicitly in the temporal logic context. We focus only on state-based abstractions (Section 9 of [15]) and we ignore abstraction of traces. Our contribution compared to these works is to work out the abstract semantics for a specific class of constraint-based abstractions, and point the way to effective abstract model checking implementations using SMT solvers. Kelb [32] develops a related abstract model checking algorithm based on abstraction of universal and existential predecessor functions.

Giacobazzi and Quintarelli [24] discuss abstraction of temporal logic and their refinement, but deal only with checking universal properties. Saïdi and Shankar [40] also develop an abstract model checking algorithm integrated with a theorem proving system for handling property-based abstractions. Their approach also uses abstract interpretation but develops a framework that uses both over- and under-approximations for handling different kinds of formula.

Our technique for modelling and verifying real time and concurrent systems using constraint logic programs [3] builds on the work of a number of other authors, including Gupta and Pontelli [26], Jaffar *et al.* [31] and Delzanno and Podelski [17]. However we take a different direction from them in our approach to abstraction and checking of temporal properties, in that we use abstract CLP program semantics when abstracting the state space (only briefly mentioned in the present work), but then apply this abstraction in a temporal logic framework, which is the topic of this work. Other authors have encoded both the transition systems and CTL semantics as constraint logic programs [8,34,37,18,21,38,39]. However none of these develops a comprehensive approach to abstract semantics when dealing with infinite-state systems. Perhaps a unified CLP-based approach to abstract CTL semantics could be constructed based on these works, but sound abstraction of negation in logic programming remains a significant complication in such an approach.

8 Conclusion

We have demonstrated a practical approach to abstract model checking, by constructing an abstract semantic function for the μ-calculus based on a Galois connection. Much previous work on abstract model checking is restricted to verifying *universal properties* and requires the construction of an abstract transition system. In other approaches in which arbitrary properties can be checked [25,16], a dual abstract transition system is constructed. Like Cousot and Cousot [15] we do not find it necessary to construct any abstract transition system, but rather

abstract the concrete semantic function systematically. Using abstract domains based on constraints we are able to implement the semantics directly. The use of an SMT solver adds greatly to the effectiveness of the approach.

Acknowledgements. We gratefully acknowledge discussions with Dennis Dams, César Sánchez, Kim Guldstrand Larsen and suggestions by the LPAR-16 referees.

References

1. Alur, R., Henzinger, T.A., Lafferriere, G., Pappas, G.J.: Discrete abstractions of hybrid systems. In: Proceedings of the IEEE, pp. 971–984 (2000)
2. Bagnara, R., Hill, P.M., Zaffanella, E.: The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. Science of Computer Programming 72(1-2), 3–21 (2008)
3. Banda, G., Gallagher, J.P.: Analysis of Linear Hybrid Systems in CLP. In: Hanus, M. (ed.) LOPSTR 2008. LNCS, vol. 5438, pp. 55–70. Springer, Heidelberg (2009)
4. Banda, G., Gallagher, J.P.: Constraint-based abstraction of a model checker for infinite state systems. In: Wolf, A., Geske, U. (eds.) Proceedings of the 23rd Workshop on (Constraint) Logic Programming, University of Potsdam (online Technical Report series) (2009)
5. Behrmann, G., David, A., Larsen, K.G.: A tutorial on Uppaal. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004)
6. Benoy, F., King, A.: Inferring argument size relationships with CLP(R). In: Gallagher, J.P. (ed.) LOPSTR 1996. LNCS, vol. 1207, pp. 204–223. Springer, Heidelberg (1997)
7. Browne, A., Clarke, E.M., Jha, S., Long, D.E., Marrero, W.R.: An improved algorithm for the evaluation of fixpoint expressions. Theor. Comput. Sci. 178(1-2), 237–255 (1997)
8. Brzoska, C.: Temporal logic programming in dense time. In: ILPS, pp. 303–317. MIT Press, Cambridge (1995)
9. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000)
10. Clarke, E.M., Grumberg, O., Long, D.E.: Model checking and abstraction. In: POPL, pp. 342–354 (1992)
11. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (2000)
12. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM Symposium on Principles of Programming Languages, Los Angeles, pp. 238–252 (1977)
13. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: POPL 1979, pp. 269–282. ACM Press, New York (1979)
14. Cousot, P., Cousot, R.: Refining model checking by abstract interpretation. Autom. Softw. Eng. 6(1), 69–95 (1999)
15. Cousot, P., Cousot, R.: Temporal abstract interpretation. In: POPL 2000, pp. 12–25 (2000)

16. Dams, D., Gerth, R., Grumberg, O.: Abstract interpretation of reactive systems. ACM Trans. Program. Lang. Syst. 19(2), 253–291 (1997)
17. Delzanno, G., Podelski, A.: Model checking in CLP. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 223–239. Springer, Heidelberg (1999)
18. Du, X., Ramakrishnan, C.R., Smolka, S.A.: Real-time verification techniques for untimed systems. Electr. Notes Theor. Comput. Sci. 39(3) (2000)
19. Dutertre, B., de Moura, L.M.: A fast linear-arithmetic solver for DPLL(T). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer, Heidelberg (2006)
20. Emerson, E.A.: Model checking and the mu-calculus. In: Immerman, N., Kolaitis, P.G. (eds.) Descriptive Complexity and Finite Models. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 31, pp. 185–214. American Mathematical Society, Providence (1996)
21. Fioravanti, F., Pettorossi, A., Proietti, M.: Verifying CTL properties of infinite-state systems by specializing constraint logic programs. In: Leuschel, M., Podelski, A., Ramakrishnan, C., Ultes-Nitsche, U. (eds.) Proceedings of the Second International Workshop on Verification and Computational Logic (VCL 2001), pp. 85–96. Tech. Report DSSE-TR-2001-3, University of Southampton (2001)
22. Frehse, G.: PHAVer: Algorithmic verification of hybrid systems past HyTech. In: Morari, M., Thiele, L. (eds.) HSCC 2005. LNCS, vol. 3414, pp. 258–273. Springer, Heidelberg (2005)
23. Ganty, P.: The Fixpoint Checking Problem: An Abstraction Refinement Perspective. PhD thesis, Université Libre de Bruxelles, Département d'Informatique (2007)
24. Giacobazzi, R., Quintarelli, E.: Incompleteness, counterexamples, and refinements in abstract model-checking. In: Cousot, P. (ed.) SAS 2001. LNCS, vol. 2126, pp. 356–373. Springer, Heidelberg (2001)
25. Godefroid, P., Huth, M., Jagadeesan, R.: Abstraction-based model checking using modal transition systems. In: Larsen, K.G., Nielsen, M. (eds.) CONCUR 2001. LNCS, vol. 2154, pp. 426–440. Springer, Heidelberg (2001)
26. Gupta, G., Pontelli, E.: A constraint-based approach for specification and verification of real-time systems. In: IEEE Real-Time Systems Symposium, pp. 230–239 (1997)
27. Halbwachs, N., Proy, Y.E., Raymound, P.: Verification of linear hybrid systems by means of convex approximations. In: LeCharlier, B. (ed.) SAS 1994. LNCS, vol. 864, pp. 223–237. Springer, Heidelberg (1994)
28. Henriksen, K.S., Banda, G., Gallagher, J.P.: Experiments with a convex polyhedral analysis tool for logic programs. In: Workshop on Logic Programming Environments, Porto (2007)
29. Henzinger, T.A., Ho, P.-H., Wong-Toi, H.: HYTECH: A model checker for hybrid systems. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 460–463. Springer, Heidelberg (1997)
30. Huth, M.R.A., Ryan, M.D.: Logic in Computer Science: Modelling and reasoning about systems. Cambridge University Press, Cambridge (2000)
31. Jaffar, J., Santosa, A.E., Voicu, R.: A CLP proof method for timed automata. In: Anderson, J., Sztipanovits, J. (eds.) The 25th IEEE International Real-Time Systems Symposium, pp. 175–186. IEEE Computer Society, Los Alamitos (2004)
32. Kelb, P.: Model checking and abstraction: A framework preserving both truth and failure information. Technical report, Carl yon Ossietzky Univ. of Oldenburg, Oldenburg, Germany (1994)

33. Larsen, K.G., Thomsen, B.: A modal process logic. In: Proceedings, Third Annual Symposium on Logic in Computer Science, Edinburgh, Scotland, UK, July 5-8, pp. 203–210. IEEE Computer Society, Los Alamitos (1988)
34. Leuschel, M., Massart, T.: Infinite state model checking by abstract interpretation and program specialisation. In: Bossi, A. (ed.) LOPSTR 1999. LNCS, vol. 1817, pp. 63–82. Springer, Heidelberg (2000)
35. Loiseaux, C., Graf, S., Sifakis, J., Bouajjani, A., Bensalem, S.: Property preserving abstractions for the verification of concurrent systems. Formal Methods in System Design 6(1), 11–44 (1995)
36. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer, New York (1999)
37. Nilsson, U., Lübcke, J.: Constraint logic programming for local and symbolic model-checking. In: Palamidessi, C., Moniz Pereira, L., Lloyd, J.W., Dahl, V., Furbach, U., Kerber, M., Lau, K.-K., Sagiv, Y., Stuckey, P.J. (eds.) CL 2000. LNCS (LNAI), vol. 1861, pp. 384–398. Springer, Heidelberg (2000)
38. Pemmasani, G., Ramakrishnan, C.R., Ramakrishnan, I.V.: Efficient real-time model checking using tabled logic programming and constraints. In: Stuckey, P.J. (ed.) ICLP 2002. LNCS, vol. 2401, pp. 100–114. Springer, Heidelberg (2002)
39. Peralta, J.C., Gallagher, J.P.: Convex hull abstractions in specialization of CLP programs. In: Leuschel, M. (ed.) LOPSTR 2002. LNCS, vol. 2664, pp. 90–108. Springer, Heidelberg (2003)
40. Saïdi, H., Shankar, N.: Abstract and model check while you prove. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 443–454. Springer, Heidelberg (1999)
41. Tarski, A.: A lattice-theoretical fixpoint theorem and its applications. Pacific Journal of Mathematics 5, 285–309 (1955)

On the Equality of Probabilistic Terms*

Gilles Barthe[1], Marion Daubignard[2], Bruce Kapron[3], Yassine Lakhnech[2], and Vincent Laporte[4]

[1] IMDEA Software, Madrid, Spain
[2] VERIMAG, Grenoble, France
[3] University of Victoria, Canada
[4] ENS Cachan, France

Abstract. We consider a mild extension of universal algebra in which terms are built both from deterministic and probabilistic variables, and are interpreted as distributions. We formulate an equational proof system to establish equality between probabilistic terms, show its soundness, and provide heuristics for proving the validity of equations. Moreover, we provide decision procedures for deciding the validity of a system of equations under specific theories that are commonly used in cryptographic proofs, and use concatenation, truncation, and xor. We illustrate the applicability of our formalism in cryptographic proofs, showing how it can be used to prove standard equalities such as optimistic sampling and one-time padding as well as non-trivial equalities for standard schemes such as OAEP.

1 Introduction

Provable security [15] is a methodology used by cryptographers for providing rigorous mathematical proofs of the correctness of cryptographic schemes. One of the popular tools for provable security is the game-based technique [4], in which cryptographic proofs are organized as a sequence of game/event pairs:

$$G_0, A_0 \rightarrow^{h_1} G_1, A_1 \rightarrow \cdots \rightarrow^{h_n} G_n, A_n$$

where G_0, A_0 formalises the security goal—e.g. IND-CPA and IND-CCA for an encryption scheme or UF-CMA and EF-CMA for signature schemes—and the scheme under study, and h_i are monotonic functions such that $\Pr_{G_i}[A_i] \leq h_{i+1}(\Pr_{G_{i+1}}[A_{i+1}])$. By composition, $h_1 \circ \cdots \circ h_n(\Pr_{G_n}[A_n])$ is an upper bound for $\Pr_{G_n}[A_n]$.

While the game-based technique does not advocate any formalism for games, some authors find convenient to model games as probabilistic programs. In this setting, game-based cryptographic proofs often proceed by replacing a set of algebraic expressions $s_1 \ldots s_n$ by another set of expressions $t_1 \ldots t_n$ in the program. The correctness of the transformation is guaranteed provided the tuples of terms $s_1 \ldots s_n$ and $t_1 \ldots t_n$ yield equal distributions. Notable examples include:

* This work was partially supported by French ANR SESUR-012, SCALP, Spanish project TIN2009-14599 DESAFIOS 10, and Madrid Regional project S2009TIC-1465 PROMETIDOS.

E.M. Clarke and A. Voronkov (Eds.): LPAR-16, LNAI 6355, pp. 46–63, 2010.

One-time padding: for every cyclic group G of prime order and generator g of G, the distributions g^x and $c \cdot g^x$, where the variable x is sampled randomly over $\mathbb{Z}_q$, are equal;

Optimistic sampling: for every k, the distributions $(x, x \oplus y)$ and $(x \oplus y, x)$ are equal, where x is sampled uniformly over the set of bitstrings of size k, and y is an arbitrary but fixed bitstring of size k—here $\oplus$ denotes the bitwise xor on bitstrings.

The purpose of this article is to provide a formalism that captures and justifies the equational reasonings that pervade cryptographic proofs. To this end, we consider an extension of universal algebra that distinguishes between probabilistic variables and deterministic variables. While deterministic variables are interpreted in the usual way via valuations, the interpretation of probabilistic variables is through sampling, so that the intepretation $[\![t]\!]_{\boldsymbol{y} \mapsto \boldsymbol{b}}$ of a term t with probabilistic variables $\boldsymbol{x}$ and deterministic variables $\boldsymbol{y}$ under the valuation $\boldsymbol{y} \mapsto \boldsymbol{b}$ is defined as

$$\lambda c \in \sigma.\ \Pr_{\boldsymbol{a} \in \boldsymbol{\tau}}[t[\boldsymbol{x}, \boldsymbol{y} := \boldsymbol{a}, \boldsymbol{b}] = c]$$

where $\boldsymbol{\tau}$ is the type of $\boldsymbol{x}$ and σ is the type of t, and where $\cdot[\cdot := \cdot]$ denotes substitution of variables by values. In the case of optimistic sampling, where the variable x is probabilistic and the variable y is deterministic, the interpretation $[\![x \oplus y]\!]_{y \mapsto b}$ of the expression $x \oplus y$ w.r.t. a valuation $y \mapsto b$ is defined as $a \xleftarrow{\$} \{0,1\}^k$, $[\![x \oplus y]\!]_{x \mapsto a, y \mapsto b}$, i.e. the distribution obtained by monadic composition of the uniform distribution over $\{0,1\}^k$, and of the (deterministic) interpretation of $\langle\!\langle x \oplus y \rangle\!\rangle_{x \mapsto a, y \mapsto b}$. Equivalently, $[\![x \oplus y]\!]_{y \mapsto b} = \lambda c.\ \Pr_{a \in \{0,1\}^k}[a \oplus b = c]$. Under this interpretation, one can show that

$$[\![\langle x \oplus y, x \rangle]\!]_{y \mapsto b} = \lambda c, d.\ \Pr_{a \in \{0,1\}^k}[a \oplus b = c, a = d]$$

is equal to

$$[\![\langle x, x \oplus y \rangle]\!]_{y \mapsto b} = \lambda c, d.\ \Pr_{a \in \{0,1\}^k}[a = c, a \oplus b = d]$$

Note that the equational theory of probabilistic terms reveals some subtleties: for example, the equation $x \doteq y$ is valid whenever x and y are probabilistic variables of the same type; however, the equation $\langle x, x \rangle \doteq \langle y, y' \rangle$ is not valid in general—as a result, it is important to consider systems of equations rather than single equations, as further explained below.

Our main contributions are:

- the definition of a proof system for reasoning about equations, and systems of equations. We prove that the system is sound and provide useful heuristics for establishing the validity of a system of equations;
- for specific theories, including the theory of xor and concatenation, the definition of decision procedures for deciding the validity of a system of equations; and sufficient conditions for the decidability of the validity of a system of equations.

2 A Motivating Example

We illustrate the need for proving equality between distributions with one classical example of encryption scheme, namely RSA-OAEP [5,9]. Recall that an asymmetric encryption scheme is specified by a triple $(\mathcal{KG}, \mathcal{E}, \mathcal{D})$ where $\mathcal{KG}$ is a key generation algorithm which outputs a pair of public and private keys, $\mathcal{E}$ is an encryption algorithm that takes an input a public key and a plaintext algorithm and outputs a ciphertext, and a decryrption algorithm that takes the private key and the ciphertext and produces the corresponding plaintext. An asymmetric encryption scheme $(\mathcal{KG}, \mathcal{E}, \mathcal{D})$ is said to be indistinguishable between real or random (IND-ROR) if the difference between the final distribution of the two games is small:[1]

$$(sk, pk) \leftarrow \mathcal{KG}; m \leftarrow A(pk); c \leftarrow \mathcal{E}(pk, m); \mathsf{return}\ c$$
$$(sk, pk) \leftarrow \mathcal{KG}; m \leftarrow A(pk); y \xleftarrow{\$} \{0,1\}^k; c \leftarrow \mathcal{E}(pk, y); \mathsf{return}\ c$$

where A is the procedure that represents the adversary.

OAEP is a famous padding scheme that is used for increasing robustness of RSA encryption. The OAEP algorithm relies on two random oracles G and H, which are sampled during initialization—we gloss over the size of the arguments and images of H and G. Key generation, encryption and decryption are respectively defined as:

$$\begin{aligned}
\mathcal{KG} &= (f, f^{-1}) \xleftarrow{\$} \Lambda,\ \mathsf{return}\ (f, f^{-1}) \\
\mathcal{E}(m, f) &= r^* \xleftarrow{\$} \{0,1\}^{k_0};\ s^* \leftarrow (m \mid 0^{k_1}) \oplus G(r^*);\ t^* \leftarrow H(s^*) \oplus r^* \\
&\quad \mathsf{return}\ f(s^* \mid t^*) \\
\mathcal{D}(y) &= s|t := f^{-1}(y); r := H(s) \oplus t; \\
&\quad \mathsf{if}\ [G(r)]_{k_1} = [s]_{k_1}\ \mathsf{then}\ (\mathsf{return}\ [s \oplus G(r)]^{k-k_1})\ \mathsf{else}\ \mathsf{reject}
\end{aligned}$$

where where Λ denotes the set of trapdoor permutations—for the purpose of this paper, it is sufficient to know that f and f^{-1} are inverse to each other—-and $[.]_k$ and $[.]^k$ respectively denote taking and removing the first k bits of a bitstring.

The first step in the proof of IND-ROR for OAEP is to show that the two code snippets below yield the same distribution:

$$\begin{array}{ll}
r^* \xleftarrow{\$} \{0,1\}^{k_0}; m \leftarrow A(f); g^* \xleftarrow{\$} \{0,1\}^{k-k_0}; & m \leftarrow A(f); y \xleftarrow{\$} \{0,1\}^k; \\
\mathsf{return}\ f((m \mid 0^{k_1}) \oplus g^* | H((m \mid 0^{k_1}) \oplus g^*) \oplus r^*) & \mathsf{return}\ y
\end{array}$$

In order to prove the equality, one must show the validity of the equation:

$$f((m \mid 0^{k_1}) \oplus g^* | H((m \mid 0^{k_1}) \oplus g^*) \oplus r^*) \doteq y$$

where g^*, r^*, y are random variables. More formally, one must show that the distribution induced by the left hand side by sampling uniformly g^* and r^* over

[1] Technically, games are indexed by a security parameter η and IND-ROR states that the distance between the families of distributions induced by the indexed games are negligible in η.

their respective sets is the uniform distribution. The informal argument goes as follows: since r^* is uniformly distributed and only occurs once, therefore the expression $H((m \mid 0^{k_1}) \oplus g^*) \oplus r^*$ is uniformly distributed and can be replaced by a fresh random variable hr^*. Thus, we are left to prove

$$f((m \mid 0^{k_1}) \oplus g^* | hr^*) \doteq y$$

Now, g^* is uniformly distributed and only occurs once, therefore the expression $(m \mid 0^{k_1}) \oplus g^*$ is uniformly distributed and can be replaced by a fresh random variable mg^*. Thus, we are left to prove

$$f(mg^* | hr^*) \doteq y$$

The concatenation of random variables being random, one can subsitute $mg^*|hr^*$ by a fresh variable z^*, so that one is left to prove

$$f(z^*) \doteq y$$

To conclude, observe that f is a bijection so $f(z^*)$ is uniformly distributed, and hence we indeed have $f(z^*) \doteq y$. In the course of the paper, we will develop a procedure that formalizes this reasoning.

3 Preliminaries

We refer to e.g. Chapter 8 of [14] for an introduction to finite distributions, with examples from cryptography. Throughout the paper, we only consider (sub)distributions over finite sets: let A be a finite set; the set $\mathcal{D}(A)$ of distributions over A is the set of functions $d : A \to [0, 1]$ such that $\sum_{a \in A} d(a) \leq 1$. Given a distribution $d \in \mathcal{D}(A)$ and an element $a \in A$, we write $\mathsf{Pr}[d = a]$ for $d(a)$.

Let A be a finite set of cardinal q. The uniform distribution over A assigns to each element of A probability q^{-1}. We write $x \xleftarrow{\$} A$ to denote the uniform distribution on A. The monadic composition of the uniform distribution and of a function $f : A \to \mathcal{D}(B)$ is the distribution $y \xleftarrow{\$} A,\ f(y)$, which is defined by the clause $\mathsf{Pr}[y \xleftarrow{\$} A,\ f(y) = b] = \frac{q'}{q}$ where q' is the cardinal of $f^{-1}(b)$. Intuitively, this is the distribution of a random variable which is obtained by sampling A uniformly at random to obtain a value y, and then evaluating f at y.

The product distribution $d_1 \times \cdots \times d_n$ of the distributions $d_1 \ldots d_n$ is defined as $x_1 \xleftarrow{\$} d_1\ \ldots\ x_n \xleftarrow{\$} d_n,\ (x_1, \ldots, x_n)$. Conversely, the i-th projection of a distribution d over $A_1 \times \cdots \times A_n$ is the distribution $x \xleftarrow{\$} d,\ \pi_i(x)$, where π_i denotes the usual projection.

The following observation, which only holds for finite domains and uniform distributions, is the cornerstone of the general decision procedure for deciding equality of distributions.

Proposition 1. *For all finite sets A and B, and functions $f, g : A \to B$, the following are equivalent:*

– $x \xleftarrow{\$} A,\ f(x) = x \xleftarrow{\$} A,\ g(x)$
– *there exists a bijection* $h : A \to A$ *such that* $f = g \circ h$.

Note that since A is finite, h is bijective iff it is injective iff it is surjective.

A remark on products. Throughout the paper, we use the vector notation to denote tuples of terms. Accordingly, we use tuple notations to denote the product of their types, thus $\boldsymbol{t}$ denotes a tuple of terms and $\boldsymbol{\sigma}$ denotes the product of their types.

4 Syntax and Semantics

This section introduces the syntax and semantics of probabilistic terms, and gives a precise formulation of the satisfaction problem for systems of equations of probabilistic terms. For an introduction to equational logic and term rewriting see e.g. [1].

4.1 Syntax

We start from the notion of many-sorted signature. We allow function symbols to be overloaded, but impose restrictions to ensure that terms have at most one sort.

Definition 1 (Signature). *A signature is a triple* $\Sigma = (\mathcal{S}, \mathcal{F}, :)$*, where* $\mathcal{S}$ *is a set of sorts,* $\mathcal{F}$ *is a set of function symbols, and* $:$ *is a typing relation between function symbols and arities of the form* $\sigma_1 \times \ldots \times \sigma_n \to \tau$*, with* $\sigma_1 \ldots \sigma_n\ \tau \in \mathcal{S}$*.*

We require that the typing relation is functional, i.e. if $f : \sigma_1 \times \ldots \times \sigma_n \to \tau$ *and* $f : \sigma_1 \times \ldots \times \sigma_n \to \tau'$*, then* $\tau = \tau'$*. In particular, we assume that constants have a single type.*

Terms are built in the usual way, except that we distinguish between two sets of variables: the set $\mathcal{R}$ denotes variables that are interpreted probabilistically, and the set $\mathcal{D}$ denotes variables that are interpreted deterministically. It is convenient to assume that there are infinitely many deterministic and probabilistic variables of each sort. Moreover, we assume that for every $x \in \mathcal{R}$ there exists a distinguished variable $\bar{x} \in \mathcal{D}$ of the same sort.

Definition 2 (Terms and substitutions). *Let* $\Sigma = (\mathcal{S}, \mathcal{F}, :)$ *be a signature and let* X *be a collection of variables. The set* $\mathcal{T}_X$ *of terms over* X *is built from the syntax:* $t ::= x \mid f(\boldsymbol{t})$ *where* f *ranges over* $\mathcal{F}$ *and* x *ranges over* X*. In the sequel, we consider the set of terms over* $\mathcal{V} = \mathcal{D} \cup \mathcal{R}$*, and write* $\mathcal{T}$ *instead of* $\mathcal{T}_{\mathcal{D}\cup\mathcal{R}}$*. Elements of* $\mathcal{T}_{\mathcal{D}}$ *are called* $\mathcal{D}$*-terms.*

Substitutions over X *(to* $\mathcal{T}_Y$*) are defined as functions from* X *to* $\mathcal{T}_Y$*; we let* $\rho\ t$ *denote the result of applying the substitution* ρ *to* t*.*

Given $Y \subseteq X$, we let $\mathsf{var}_Y(t)$ denote $\mathsf{var}(t) \cap Y$, where $\mathsf{var}(t)$ is defined in the usual way. Moreover, we say that $t \equiv_{\alpha(Y)} t'$ iff there exists a 1-1 renaming $\rho : \mathsf{var}_Y(t) \to \mathsf{var}_Y(t')$ such that $\rho\, t = t'$.

Terms are subject to a simple typing discipline that ensures that functions are applied to arguments of the correct types. In the sequel, we implicitly assume that each variable x has a unique sort σ_x and that terms are well-typed; we adopt the standard notations $t : \sigma$ (resp. $t \in \mathcal{T}_X(\sigma)$) to denote that a term t has type σ (resp. t has type σ and $\mathsf{var}(t) \subseteq X$). Thanks to requiring that typing is functional, every term has at most one type.

Definition 3 (System of equations). *A system of equations over a set X, or X-system of equations, is a statement $s_1 \doteq t_1 \wedge \ldots \wedge s_n \doteq t_n$ where, for $i = 1 \ldots n$, s_i and t_i have the same type, i.e. $s_i, t_i \in \mathcal{T}_X(\sigma_i)$ for some σ_i. We often use $\boldsymbol{s} \doteq \boldsymbol{t}$ as a shorthand for systems of equations.*

Unlike equational logic, it is important to consider systems of equations rather than single equations. Because of the possible dependencies between terms, the conjunction of two valid equalities may not be valid.

Consider the probabilistic variables x, y, z of type σ: the system of equations $x \doteq y \wedge x \doteq z$ is not valid, whereas the two equations $x \doteq y$ and $x \doteq z$ are valid; this is because the distribution $y \xleftarrow{\$} \sigma, z \xleftarrow{\$} \sigma,\ \langle y, z\rangle$ yields the uniform distribution over $\sigma \times \sigma$ whereas $x \xleftarrow{\$} \sigma,\ \langle x, x\rangle$ does not.

Definition 4 (Theory). *A theory is a pair $\mathbb{T} = (\Sigma, E)$ where Σ is a signature and E is a (possibly infinite) set of of systems of equations.*

4.2 Semantics

The semantics of probabilistic terms is adapted immediately from equational logic. In particular, algebras provide the natural semantics for signatures.

Definition 5 (Algebra). *Let $\Sigma = (\mathcal{S}, \mathcal{F}, :)$ be a signature. A Σ-algebra is a pair $\mathbb{A} = ((\mathcal{A}_\sigma)_{\sigma\in\mathcal{S}}, (f_\mathcal{A})_{f\in\mathcal{F}})$ where $\mathcal{A}_\sigma$ is a finite set that interprets the sort σ and $f_\mathcal{A} \in \mathcal{A}_{\sigma_1} \times \cdots \times \mathcal{A}_{\sigma_n} \to \mathcal{A}_\tau$ for every $f \in \mathcal{F}$ such that $f : \sigma_1 \times \ldots \times \sigma_n \to \tau$. In the sequel, we let $\mathcal{A} = \bigcup_{\sigma\in\mathcal{S}} \mathcal{A}_\sigma$ and write $[\![\sigma]\!]$ instead of $\mathcal{A}_\sigma$.*

Terms are interpreted as distributions, by taking a probabilistic interpretation of variables in $\mathcal{R}$.

Definition 6 (Interpretation of terms). *Let $\Sigma = (\mathcal{S}, \mathcal{F}, :)$ be a signature and $\mathbb{A} = ((\mathcal{A}_\sigma)_{\sigma\in\mathcal{S}}, (f_\mathcal{A})_{f\in\mathcal{F}})$ be a Σ-algebra.*

- *An X-valuation is a function $\rho : X \to \mathcal{A}$ such that $\rho(x) \in \mathcal{A}_{\sigma_x}$ for every $x \in X$. We let Val_X denote the set of X-valuations. In the sequel, we often omit the subscript; moreover, we often use the notation $\boldsymbol{x} \mapsto \boldsymbol{a}$ to denote any valuation ρ such that $\rho(x_i) = a_i$.*

– *Let $\rho \in \mathsf{Val}_X$. The pre-interpretation $\langle\!\langle t \rangle\!\rangle_\rho$ of a term $t \in \mathcal{T}_X$ is defined as:*

$$\langle\!\langle t \rangle\!\rangle_\rho = \begin{cases} \rho(t) & \text{if } t \in X \\ f_{\mathcal{A}}(\langle\!\langle t_1 \rangle\!\rangle_\rho, \ldots, \langle\!\langle t_n \rangle\!\rangle_\rho) & \text{if } t = f(t_1, \ldots, t_n) \end{cases}$$

– *Let $\rho \in \mathsf{Val}_{\mathcal{D}}$. The interpretation $[\![\boldsymbol{t}]\!]_\rho$ of a tuple of terms $\boldsymbol{t}$ of type $\boldsymbol{\sigma}$ is defined by the clause:*

$$[\![\boldsymbol{t}]\!]_\rho = \lambda \boldsymbol{a} : \boldsymbol{\sigma}. \Pr_{\rho' \in \mathsf{Val}_{\mathcal{R}}}[\langle\!\langle \boldsymbol{t} \rangle\!\rangle_{\rho+\rho'} = \boldsymbol{a}]$$

where the valuation $\rho + \rho'$ denotes the (disjoint) union of ρ and ρ', and for every tuple of terms $\boldsymbol{t} = \langle t_1, \ldots, t_n \rangle$ and for every valuation ρ, $\langle\!\langle \boldsymbol{t} \rangle\!\rangle_\rho$ denotes

$$\langle \langle\!\langle t_1 \rangle\!\rangle_\rho, \ldots, \langle\!\langle t_n \rangle\!\rangle_\rho \rangle$$

Note that the interpretation of a tuple of terms needs not coincide with the product distribution of their interpretations. For example, $[\![x, x]\!]_\rho \neq [\![x]\!]_\rho \times [\![x]\!]_\rho$ for every $x \in \mathcal{R}$.

Definition 7 (Model). *Let $\mathbb{T} = (\Sigma, E)$ be a theory; let $\mathbb{A} = ((\mathcal{A}_\sigma)_{\sigma \in \mathcal{S}}, (f_{\mathcal{A}})_{f \in \mathcal{F}})$ be a Σ-algebra.*

– *A system of equations $\boldsymbol{s} \doteq \boldsymbol{t}$ is valid in $\mathbb{A}$, written $\mathbb{A} \models \boldsymbol{s} \doteq \boldsymbol{t}$ iff for every $\rho_{\mathcal{D}} \in \mathsf{Val}_{\mathcal{D}}$, we have $[\![\boldsymbol{s}]\!]_{\rho_{\mathcal{D}}} = [\![\boldsymbol{t}]\!]_{\rho_{\mathcal{D}}}$.*
– *$\mathbb{A}$ is a $\mathbb{T}$-algebra (or $\mathbb{T}$-model) iff for every system of equations $\boldsymbol{s} \doteq \boldsymbol{t} \in E$, we have $\mathbb{A} \models \boldsymbol{s} \doteq \boldsymbol{t}$.*

The notion of model for an equational theory coincides with that of equational logic for theories with $\mathcal{D}$-systems of equations.

Note that one can prove that the following equations are valid: $x \doteq y$ for every probabilistic variables x and y of the same type, $x \oplus x' \doteq y$ for every probabilistic variables x, x' and y of type $\{0,1\}^k$.

4.3 Satisfaction Problem

The problem addressed in this paper can now be stated formally: given a theory $\mathbb{T} = (\Sigma, E)$, a collection of Σ-algebras $(\mathbb{A}_i)_{i \in \mathcal{I}}$ and a system of equations $\boldsymbol{s} \doteq \boldsymbol{t}$, can we decide whether $\forall i \in \mathcal{I}, \mathbb{A}_i \models \boldsymbol{s} \doteq \boldsymbol{t}$. We write $\mathsf{Dec}_{\mathrm{Sat}(\mathbb{T}, (\mathbb{A}_i)_{i \in \mathcal{I}})}$ if the problem is decidable.

Stating the satisfaction problem relative to a collection of models rather than a single one is somewhat unusual, and is motivated by the need to carry cryptographic proofs parametrically in the size of the security parameter.

5 Exclusive or, Concatenation, and Projection

The purpose of this section is to present decision procedures for the theories of exclusive or, and the theory of exclusive or, concatenation, and projection.

$$
\begin{array}{rll}
(s \mid t) \oplus (s' \mid t') & = (\downarrow_{(1,\#s')}\ s \mid (\downarrow_{(\#s'+1,\#s)}\ s \mid t)) \oplus (s' \mid t') & \text{if } \#s' < \#s \\
(s \mid t) \oplus (s' \mid t') & = (s \mid t) \oplus (\downarrow_{(1,\#s)}\ s' \mid (\downarrow_{(\#s+1,\#s')}\ s' \mid t')) & \text{if } \#s < \#s' \\
(s \mid t) \oplus (s' \mid t') & = (s \oplus s') \mid (t \oplus t') & \text{if } \#s = \#s' \\
\downarrow_{(i_1,i_2)}\ (s \oplus t) & = (\downarrow_{(i_1,i_2)}\ s) \oplus (\downarrow_{(i_1,i_2)}\ t) & \\
\downarrow_{(i_1,i_2)}\ (s \mid t) & = \downarrow_{(i_1,i_2)}\ s & \text{if } i_2 \leq \#s \\
\downarrow_{(i_1,i_2)}\ (s \mid t) & = \downarrow_{(i_1-\#s,i_2-\#s)}\ t & \text{if } \#s < i_1 \\
\downarrow_{(i_1,i_2)}\ (s \mid t) & = \downarrow_{(i_1,\#s)}\ s \mid\mid \downarrow_{(1,i_2-\#s)}\ t & \text{if } i_1 \leq \#s < i_2 \\
\downarrow_{(i_1,i_2)}\ (\downarrow_{(j_1,j_2)}\ s) & = \downarrow_{(i_1+j_1,i_2+j_1)}\ s & \\
\downarrow_{(1,\#s)}\ (s) & = s &
\end{array}
$$

Fig. 1. Theory of concatenation and projection

5.1 Exclusive or

The first theory $\mathbb{T}_{\oplus}$ has a single sort bs, a constant $0 : \mathsf{bs}$ and a binary symbol $\oplus : \mathsf{bs} \times \mathsf{bs} \to \mathsf{bs}$. Its axioms are:

$$
\begin{array}{cc}
x \oplus (y \oplus z) \doteq (x \oplus y) \oplus z & x \oplus y \doteq y \oplus x \\
x \oplus 0 \doteq x & x \oplus x \doteq 0
\end{array}
$$

We consider the family of algebras $(\mathcal{BS}_k)_{k\in\mathbb{N}}$, where $\mathcal{BS}_k$ is the set of bistrings of size k, with the obvious interpretation for terms. By abuse of notation, we write $\models \boldsymbol{s} \doteq \boldsymbol{t}$ instead of $\forall k \in \mathbb{N},\ \mathcal{BS}_k \models \boldsymbol{s} \doteq \boldsymbol{t}$.

We begin by stating some simple facts. First, one can decide whether $\mathcal{D}$-equations hold.

Lemma 1. *Let $s, t \in \mathcal{T}_{\mathcal{D}}$. It is decidable whether $\models s \doteq t$.*

Second, one can decide whether a term is semantically equal to a variable in $\mathcal{R}$. We write $\mathcal{U}(t)$ iff for all $\rho \in \mathsf{Val}$, $[\![t]\!]_\rho$ is uniformly distributed.

Lemma 2. *Let $t \in \mathcal{T}$. It is decidable whether $\mathcal{U}(t)$.*

Proof. Every term t can be reduced to a normal form t', in which variables appear at most once. Then $[\![t]\!]_\rho$ is uniformly distributed iff t' contains at least one $\mathcal{R}$-variable.

It follows that one can decide equality of two terms.

Lemma 3. *Let s and t be terms. It is decidable whether $\models s \doteq t$.*

Proof. If $\mathcal{U}(s)$ and $\mathcal{U}(t)$, return true. If $\neg\mathcal{U}(s)$ and $\mathcal{U}(t)$ or $\mathcal{U}(s)$ and $\neg\mathcal{U}(t)$, return false. If $\neg\mathcal{U}(s)$ and $\neg\mathcal{U}(t)$, then s and t can be reduced to normal forms $s' \in \mathcal{T}_{\mathcal{D}}$ and $t' \in \mathcal{T}_{\mathcal{D}}$. Return true if $\models s' \doteq t'$ and false otherwise.

In order to extend the result to tuples of terms, we rely on the following lemma. The result is used e.g. in [6].

Proposition 2. *Let $t_1 \dots t_n \in \mathcal{T}$ such that $\mathcal{U}(t_i)$ for $1 \leq i \leq n$. Exactly one of the following statements hold:*

- $\mathsf{indep}\ (\boldsymbol{t})$*: for every* $\rho \in \mathsf{Val}$, $[\![(t_1 \dots t_n)]\!]_\rho = [\![t_1]\!]_\rho \times \cdots \times [\![t_n]\!]_\rho$*;*
- $\mathsf{dep}\ (\boldsymbol{t})$*: there is a non-null vector* $\boldsymbol{\lambda} \in \{0,1\}^n$ *and* $s \in \mathcal{T}_\mathcal{D}$ *such that* $\mathsf{dep}_{\boldsymbol{\lambda},s}\ (\boldsymbol{t})$*, where* $\mathsf{dep}_{\boldsymbol{\lambda},s}\ (\boldsymbol{t})$ *holds iff for every* $\rho \in \mathsf{Val}$*,*

$$[\![\sum_{1 \le i \le n} \lambda_i t_i]\!]_\rho = [\![s]\!]_\rho$$

where $\sum$ *denotes summation* mod 2*.*

Proof. For simplicity, assume that $t_1 \dots t_n \in \mathcal{T}_{\{y_1 \dots y_l\}}$ with $y_1 \dots y_l \in \mathcal{R}$, and consider bistrings of length k. Then $\mathsf{indep}\ (\boldsymbol{t})$ iff for all bitstrings of length k $a_1 \dots a_n$, the system of equations

$$(*) \quad \begin{cases} t_1 = a_1 \\ \quad \vdots \\ t_n = a_n \end{cases}$$

has exactly $2^{k(l-n)}$ solutions. Indeed, $\mathsf{Pr}[\bigwedge_{i=1}^n t_i = a_i] = \alpha 2^{-kl}$, where α is the number of solutions of $(*)$. It is now easy to prove by induction on n that $\alpha = 2^{k(l-n)}$ is equivalent to the linear independence of $\boldsymbol{t}$, which is equivalent to $\neg\mathsf{dep}\ (\boldsymbol{t})$.

For example, one can prove that the distribution induced by the triple of terms $(x \oplus y, y \oplus z, z \oplus x)$, where x, y, and z are probabilistic variables of type $\{0,1\}^k$ is not uniformly distributed, i.e. the system of equations:

$$x \oplus y \doteq w_1 \wedge y \oplus z \doteq w_2 \wedge z \oplus x \doteq w_3$$

is not valid, since we have $(x \oplus y) \oplus (y \oplus z) \oplus (z \oplus x) = 0$.

Note that one can effectively decide which of the two conditions hold, since there are only finitely many $\boldsymbol{\lambda}$ to test—and s, if it exists, can be computed from $\sum_{1 \le i \le n} \lambda_i t_i$. Decidability follows.

Proposition 3. $\mathsf{Dec}_{\mathrm{Sat}(\mathbb{T}_\oplus, (\mathcal{BS}_k)_{k \in \mathbb{N}})}$.

Proof. The decision procedure works as follows:

1. If the system only contains a single equation $s \doteq t$, invoke Lemma 3;
2. If $\mathsf{indep}\ (\boldsymbol{s})$ and $\mathsf{indep}\ (\boldsymbol{t})$, return true;
3. If $\mathsf{dep}_{\boldsymbol{\lambda},s}\ (\boldsymbol{s})$ and $\mathsf{dep}_{\boldsymbol{\lambda},s}\ (\boldsymbol{t})$ *for the same* $\boldsymbol{\lambda}$ and s, then pick $\lambda_k \neq 0$, and recursively check the smaller system without the equation $s_k \doteq t_k$;
4. Otherwise, return false.

Since terms of sort bs are only built from variables of sort bs, decidability extends immediately to the multi-sorted theory $\mathbb{T}^+_\oplus$, with set of sorts bs_k for all k, constants $0^k : \mathsf{bs}_k$, and a—single but overloaded—binary function symbol $\oplus : \mathsf{bs}_i \times \mathsf{bs}_i \to \mathsf{bs}_i$. The axioms are those of $\mathbb{T}_\oplus$. Finally, we consider the algebras $(\mathcal{BS}_k)_{k \in \mathbb{N}}$ with the obvious interpretation.

Proposition 4. $\mathsf{Dec}_{\mathrm{Sat}(\mathbb{T}^+_\oplus, (\mathcal{BS}_k)_{k \in \mathbb{N}})}$.

5.2 Exclusive or, Concatenation, and Projection

Next, we prove decidability for exclusive or and concatenation. Thus, the theory $\mathbb{T}_{\{\oplus,|,\downarrow\}}$ has infinitely many sorts bs^k and infinitely many function symbols:

$$0^k : \mathsf{bs}^k \quad \oplus : \mathsf{bs}^k \times \mathsf{bs}^k \to \mathsf{bs}^k \quad |: \mathsf{bs}^k \times \mathsf{bs}^{k'} \to \mathsf{bs}^{k+k'} \quad \downarrow_{(i,j)}: \mathsf{bs}^k \to \mathsf{bs}^{j-i+1}$$

where $k', i, j \in \mathbb{N}$ are such that $i \leq j \leq k$. Its axioms are those of the theory $\mathbb{T}_{\oplus}$, together with axioms for the associativity and neutral for concatenation, and with axioms for relating concatenation, projection and $\oplus$, which are given in Figure 1. Finally, we consider the indexed family of algebras $(\mathcal{BS}_i)_{i\in\mathbb{N}}$ in which the interpretation of bs^k is the set of bitstrings of length ki, with the obvious interpretation of function symbols.

Proposition 5. $\mathsf{Dec}_{\mathrm{Sat}(\mathbb{T}_{\{\oplus,|,\downarrow\}},(\mathcal{BS}_{\leq k})_{k\in\mathbb{N}})}$.

Proof. The proof proceeds by a reduction to the previous case, and relies on a set of rewrite rules that transform an arbitrary system of equations into an equivalent system without concatenation and projection. There are two sets of rewrite rules; both rely on typing information that provides the length of bitstrings; we let $\#s$ denote the length of the bistring s. The first set of rewrite rules, is obtained by orienting the rules of Figure 1 from left to right, and pushes concatenations to the outside and projections to the inside. The second set of rewrite rules, given in Figure 2, aims to eliminate concatenation and projection by transforming equations of the form $s \mid t \doteq s' \mid t'$ with $\#s = \#s'$ and $\#t = \#t'$ into a system of equations $s \doteq s' \mid t \doteq t'$, and by replacing expressions of the form $\downarrow_{(i,j)} x$ by fresh variables $x_{(i,j)}$—in order to get an equivalent system, the replacement is performed by a global substitution $[x := x_{1,i-1} \mid x_{i,j} \mid x_{j+1,\#x}]$.

The procedure terminates: intuitively, the rule for splitting variables can only be applied a finite number of times, and the remaining rules are clearly terminating. Upon termination, one obtains a system of equations of the form $\boldsymbol{s} \doteq \boldsymbol{t} \wedge \boldsymbol{x} \doteq \boldsymbol{u}$ where the ss and ts only contain $\oplus$-terms and the us, are concatenations of variables, and variables on the left hand side, i.e. the xs, do not appear in the first system of equations, and moreover variables arise at most once globally in the us. Thus, the validity of the system is equivalent to the validity of $\boldsymbol{s} \doteq \boldsymbol{t}$ which can be decided by Proposition 3.

$$\begin{array}{ll}
s_1 \mid s_2 \doteq t_1 \mid t_2 \to \langle \downarrow_{(1,\#t_1)} s_1, (\downarrow_{(\#t_1+1,\#s_1)} s_1) \mid s_2\rangle \doteq \langle t_1, t_2\rangle & \text{if } \#t_1 < \#s_1 \\
s_1 \mid s_2 \doteq t_1 \mid t_2 \to \langle s_1, s_2\rangle \doteq \langle \downarrow_{(1,\#s_1)} t_1, (\downarrow_{(\#s_1+1,\#t_1)} t_1) \mid t_2\rangle & \text{if } \#s_1 < \#t_1 \\
s_1 \mid s_2 \doteq t_1 \mid t_2 \to \langle s_1, s_2\rangle \doteq \langle t_1, t_2\rangle & \text{if } \#t_1 = \#s_1 \\
s_1 \mid s_2 \doteq t \oplus t' \to \langle s_1, s_2\rangle \doteq \langle \downarrow_{(1,\#s_1)} (t \oplus t'), \downarrow_{(\#s_1+1,\#s_1+\#s_2)} (t \oplus t')\rangle & \\
s_1 \mid s_2 \doteq \downarrow_{(i_1,i_2)} t \to \langle s_1, s_2\rangle \doteq \langle \downarrow_{(i_1,i_1+\#s_1)} t, \downarrow_{(i_1+1+\#s_1,i_1+1+\#s_2-\#s_1)} t\rangle &
\end{array}$$

$$\frac{s \doteq t \to \Delta}{\Gamma \wedge s \doteq t \to \Gamma \wedge \Delta}$$

$$\Gamma \wedge \downarrow_{(i,j)} x \doteq t \to (\Gamma \wedge x_{i,j} \doteq t)[x := x_{1,i-1} \mid x_{i,j} \mid x_{j+1,\#x}] \wedge x \doteq x_{1,i-1} \mid x_{i,j} \mid x_{j+1,\#x}$$

Fig. 2. Normalization of equation systems with concatenation and projection

6 An Equational Logic for Systems of Equations

The purpose of this section is to provide a sound proof system for proving the validity of a system of equations, and to study the conditions under which the proof system is complete and decidable.

6.1 Proof System

The proof system contains structural rules, equational rules that generalize those of equational logic, and specific rules for probabilistic terms.

Structural rules specifically deal with systems of equations; the rule [Struct] allows us to duplicate, permute, or eliminate equations. Formally $\boldsymbol{s} \doteq \boldsymbol{t} \subseteq \boldsymbol{s'} \doteq \boldsymbol{t'}$ iff for every j there exists i such that the i-th equation of $\boldsymbol{s'} \doteq \boldsymbol{t'}$ is syntactically equal to the j-th equation of $\boldsymbol{s} \doteq \boldsymbol{t}$. Moreover, the rule [Merge] allows us to merge systems of equations, provided they do not share any variables in $\mathcal{R}$. Note that the side condition of the [Merge] rule is necessary for soundness; without the side condition, one could derive for probabilistic variables x, y, z of the same type that $x \doteq y \wedge x \doteq z$ is valid (since from $x \doteq y$ and $x \doteq z$ are), which is unsound as mentioned earlier.

The equational rules include reflexivity, symmetry and transitivity of equality, congruence rules for function symbols, a rule for axioms, and a substitution rule. Note that the rule for functions is stated for ensuring soundness, and that the following rule is unsound:

$$\frac{\vdash s_1 \doteq t_1 \ldots \vdash s_n \doteq t_n}{\vdash f(s_1 \ldots s_n) \doteq f(t_1 \ldots t_n)}$$

because it would allow to derive that $\vdash x \oplus x \doteq y \oplus z$ for x, y, z probabilistic variables of type $\{0,1\}^k$. Note also that we allow in the application of the [Fun] rule to have side equations $\boldsymbol{u} \doteq \boldsymbol{v}$, which is required to have successive applications of the [Fun] rule.

Likewise, the rule for substitutions requires that the subsituted terms are deterministic; without this restriction, the rule would be unsound as for every deterministic variable y of type $\{0,1\}^k$ and probabilistic variable x of the same type, one could derive $\vdash x \doteq x \oplus y[y := x]$ from $\vdash x \doteq x \oplus y$. Note that one can combine the rule for substitution with the rule [Rand] below to allow substitutions of terms that contain *fresh* probabilistic variables, in the style of [11].

Finally, the rules for probabilistic variables include rules for α-conversion, and the rule [Bij], that is the syntactical counterpart of Proposition 1. It assumes that $\mathsf{var}_{\mathcal{R}}(\boldsymbol{s}) \cup \mathsf{var}_{\mathcal{R}}(\boldsymbol{t}) \subseteq \boldsymbol{x}$, and requires that there are $\mathcal{D}$-terms $\boldsymbol{u}$ and $\boldsymbol{v}$ that represent bijections, and such that the composition of $\boldsymbol{u}$ with $\boldsymbol{s}$ is equal to $\boldsymbol{t}[\boldsymbol{x} := \bar{\boldsymbol{x}}]$—where $\bar{\boldsymbol{x}} \in \mathcal{D}$ is a type-preserving renaming of $\boldsymbol{x}$. In the side condition, we let V_R denote $\mathsf{var}_{\mathcal{R}}(\boldsymbol{s}) \cup \mathsf{var}_{\mathcal{R}}(\boldsymbol{t})$ and V_D denote $\mathsf{var}_{\mathcal{D}}(\boldsymbol{s}) \cup \mathsf{var}_{\mathcal{D}}(\boldsymbol{t})$.

Here is an example of the use of this system to prove optimistic sampling, i.e. for every deterministic variable y of type $\{0,1\}^k$ and probabilistic variable x of the same type, $\vdash x \oplus y \doteq x$. The last step of the proof is an application of

$$\frac{\vdash \boldsymbol{s} \doteq \boldsymbol{t}}{\vdash \boldsymbol{s}' \doteq \boldsymbol{t}'}[\text{Struct}] \text{ where } \boldsymbol{s}' \doteq \boldsymbol{t}' \subseteq \boldsymbol{s} \doteq \boldsymbol{t} \qquad \frac{\boldsymbol{s} \doteq \boldsymbol{t} \subseteq E}{\vdash \boldsymbol{s} \doteq \boldsymbol{t}}[\text{Axm}]$$

$$\frac{\vdash \boldsymbol{s} \doteq \boldsymbol{t} \qquad \vdash \boldsymbol{s}' \doteq \boldsymbol{t}'}{\vdash \boldsymbol{s} \doteq \boldsymbol{t} \wedge \boldsymbol{s}' \doteq \boldsymbol{t}'}[\text{Merge}] \text{ where } (\mathsf{var}_{\mathcal{R}}(\boldsymbol{s}) \cup \mathsf{var}_{\mathcal{R}}(\boldsymbol{t})) \cap (\mathsf{var}_{\mathcal{R}}(\boldsymbol{s}') \cup \mathsf{var}_{\mathcal{R}}(\boldsymbol{t}')) = \emptyset$$

$$\frac{}{\vdash \boldsymbol{s} \doteq \boldsymbol{s}}[\text{Refl}] \qquad \frac{\vdash \boldsymbol{s} \doteq \boldsymbol{t}}{\vdash \boldsymbol{t} \doteq \boldsymbol{s}}[\text{Sym}] \qquad \frac{\vdash \boldsymbol{s} \doteq \boldsymbol{t} \qquad \vdash \boldsymbol{t} \doteq \boldsymbol{u}}{\vdash \boldsymbol{s} \doteq \boldsymbol{u}}[\text{Trans}]$$

$$\frac{\vdash \boldsymbol{u} \doteq \boldsymbol{v} \wedge s_1 \doteq t_1 \wedge \ldots \wedge s_n \doteq t_n}{\vdash \boldsymbol{u} \doteq \boldsymbol{v} \wedge f(s_1 \ldots s_n) \doteq f(t_1 \ldots t_n)}[\text{Fun}]$$

$$\frac{\vdash \boldsymbol{s} \doteq \boldsymbol{t}}{\vdash \rho\boldsymbol{s} \doteq \rho\boldsymbol{t}}[\text{Subst}] \text{where } \rho : \mathcal{D} \to \mathcal{T}_{\mathcal{D}}$$

$$\frac{\boldsymbol{s} \equiv_{\alpha(\mathcal{R})} \boldsymbol{s}' \qquad \boldsymbol{t} \equiv_{\alpha(\mathcal{R})} \boldsymbol{t}' \qquad \vdash \boldsymbol{s} \doteq \boldsymbol{t}}{\vdash \boldsymbol{s}' \doteq \boldsymbol{t}'}[\text{Alpha}]$$

$$\frac{\vdash \boldsymbol{s}[\boldsymbol{x} := \boldsymbol{u}] \doteq \boldsymbol{t}[\boldsymbol{x} := \bar{\boldsymbol{x}}] \qquad \vdash \boldsymbol{u}[\bar{\boldsymbol{x}} := \boldsymbol{v}] \doteq \bar{\boldsymbol{x}}}{\vdash \boldsymbol{s} \doteq \boldsymbol{t}}[\text{Bij}] \text{where } \begin{cases} V_R \subseteq \boldsymbol{x} \\ V_D \cap \bar{\boldsymbol{x}} = \emptyset \\ (\mathsf{var}(\boldsymbol{u}) \cup \mathsf{var}(\boldsymbol{v})) \subseteq (\bar{\boldsymbol{x}} \cup V_D) \end{cases}$$

Fig. 3. Proof system

the [Bij] rule, with $u = \bar{x} \oplus y$ and $v = \bar{x} \oplus y$. It is easy to check that the premises hold, i.e. $\vdash x \oplus y[x := \bar{x} \oplus y] \doteq y$, and $\vdash \bar{x} \oplus y[\bar{x} := \bar{x} \oplus y] \doteq \bar{x}$.

One application of the [Bij] rule is to lift equality of deterministic terms to equality of distributions. Concretely, we have:

$$\frac{\vdash \boldsymbol{s}[\boldsymbol{x} := \bar{\boldsymbol{x}}] \doteq \boldsymbol{t}[\boldsymbol{x} := \bar{\boldsymbol{x}}]}{\vdash \boldsymbol{s} \doteq \boldsymbol{t}}[\text{Rand}] \text{where } \begin{cases} V_R \subseteq \boldsymbol{x} \\ V_D \cap \bar{\boldsymbol{x}} = \emptyset \end{cases}$$

Using this rule, one can also conclude that for every distinct probabilistic variable x and y of type $\{0,1\}^k$, one has $\vdash x \oplus y \doteq x$.

In order to apply optimistic sampling in context, we must rely on a derived rule for linear variables. Given a tuple of terms $\boldsymbol{s}$ in which x of type σ appears exactly once, and assuming that $\vdash x \doteq t$ with $\mathsf{var}(t) \cup \mathsf{var}(\boldsymbol{s}) \subseteq \boldsymbol{y}$, then:

$$\frac{x \doteq t \wedge \boldsymbol{y} \doteq \boldsymbol{y}}{\vdash \boldsymbol{s} \doteq \boldsymbol{s}[x := t]}[\text{Linear}] \; x \notin \mathsf{var}(\boldsymbol{t})$$

The rule [Linear] can be proved by induction on the structure of the terms, or using the [Bij+] rule in the next section. In particular, one can prove that for every theory that contains the $\oplus$ operator and its associated equations that:

$$\frac{}{\vdash \boldsymbol{s} \doteq \boldsymbol{s}[x := x \oplus t]} \; x \notin \mathsf{var}(t) \wedge x \text{ linear in } \boldsymbol{s}$$

Note that the conjunct $\boldsymbol{y} \doteq \boldsymbol{y}$ is required in the rule [Linear] because one could otherwise take s to be $x \oplus y$ and t to be y, to prove $(x \oplus y)[x := y] \doteq x$, which is of course not valid.

6.2 Soundness

The proof system is sound.

Proposition 6. *Let $\mathbb{T} = (\Sigma, E)$ be a theory and assume that $\vdash \boldsymbol{s} \doteq \boldsymbol{t}$. For every $\mathbb{T}$-algebra $\mathbb{A}$, we have $\mathbb{A} \models \boldsymbol{s} \doteq \boldsymbol{t}$.*

Proof (Sketch). By induction on the length of derivations. We only consider the case [Bij]. Assume that we have $\mathbb{A} \models \boldsymbol{s}[\boldsymbol{x} := \boldsymbol{u}] \doteq \boldsymbol{t}[\boldsymbol{x} := \bar{\boldsymbol{x}}]$ and $\mathbb{A} \models \boldsymbol{u}[\bar{\boldsymbol{x}} := \boldsymbol{v}] \doteq \bar{\boldsymbol{x}}$. To show that $\mathbb{A} \models \boldsymbol{s} \doteq \boldsymbol{t}$, i.e. $[\![\boldsymbol{s}]\!]_\rho = [\![\boldsymbol{t}]\!]_\rho$ for every valuation $\rho \in \mathsf{Val}_{\mathcal{D}}$. We have (the second equality holds by induction hypothesis):

$$[\![\boldsymbol{s}]\!]_{\boldsymbol{x} \mapsto [\![\boldsymbol{u}]\!]_\rho} = [\![\boldsymbol{s}[\boldsymbol{x} := \boldsymbol{u}]]\!]_\rho = [\![\boldsymbol{t}[\boldsymbol{x} := \bar{\boldsymbol{x}}]]\!]_\rho$$

To conclude, it is sufficient to show that for every $\boldsymbol{a} \in [\![\sigma_{\boldsymbol{x}}]\!]$, and partial valuation ρ' with domain $(\mathsf{var}(\boldsymbol{u}) \cup \mathsf{var}(\boldsymbol{v})) \setminus \boldsymbol{x}$ the function $[\![\boldsymbol{u}]\!]_{\boldsymbol{\rho' + \bar{x} \mapsto a}}$ is a bijection from $[\![\sigma_{\boldsymbol{x}}]\!]$ to itself. By induction hypothesis, we have that $[\![\boldsymbol{u}[\bar{\boldsymbol{x}} := \boldsymbol{v}]]\!]_{\rho' + \bar{\boldsymbol{x}} \mapsto \boldsymbol{a}} = (\rho' + \bar{\boldsymbol{x}} \mapsto \boldsymbol{a})\bar{\boldsymbol{x}}$, or equivalently $[\![u]\!]_{\rho' + \bar{\boldsymbol{x}} \mapsto [\![\boldsymbol{v}]\!]_{\bar{\boldsymbol{x}} \mapsto \boldsymbol{a}}} = \boldsymbol{a}$. Hence $[\![\boldsymbol{u}]\!]_{\boldsymbol{\rho' + \bar{x} \to a}}$ is a bijection.

6.3 Products

Our proof system does not make any specific provision with product, thus it is not possible to prove that for every probabilistic variables x, y and z of respective types $\{0,1\}^k$, $\{0,1\}^{k'}$ and $\{0,1\}^{k+k'}$ one has $\vdash x|y = z$. Thus, the proof system is incomplete.

One can remedy to this issue by considering theories with products, and enriching the proof system for such theories.

Definition 8 (Theory with products). *A theory $\mathbb{T} = (\Sigma, E)$ has products iff for every sorts σ and σ', there exists a sort τ and function symbols $\pi : \tau \to \sigma$, $\pi' : \tau \to \sigma'$ and $\mathsf{pair} : \sigma \times \sigma' \to \tau$ such that the following $\mathcal{D}$-equations hold:*

$$\mathsf{pair}(\pi(y), \pi'(y)) \doteq y \qquad \pi(\mathsf{pair}(x, x')) \doteq x \qquad \pi'(\mathsf{pair}(x, x')) \doteq x'$$

Concatenation and truncation of bitstrings are the primary examples of function symbols that yield a product structure. Given a theory with products, one can show that the rules for products are sound:

$$\frac{\vdash \boldsymbol{s}[x, x' := \pi(y), \pi'(y)] \doteq \boldsymbol{t}}{\vdash \boldsymbol{s} \doteq \boldsymbol{t}}[\text{ProdE}]$$

The [ProdE] rule implicitly assumes that products exist, and that y is a fresh variable. The rule allows to collate two probabilistic variables x and x' of respective sorts σ and σ' by a probabilistic variable y of sort $\sigma \times \sigma'$, and is useful to prove the previous example. There is a dual rule [ProdI], which allows to introduce projections, and is ommitted.

6.4 Example Revisited

The example of Section 2 can be established through successive applications of the [Linear] rule, the [Prod] rule, and finally the [Bij] rule. The signature is that of bitstrings with exclusive or, concatenation, and truncation, extended with two function symbols f and f^{-1}, with additional axioms that state that f and f^{-1} are mutually inverse bijections.

The first step in the derivation is to show that the equation $f(z^*) \doteq y$ is derivable, for y and z^* probabilistic variables. The equation is established using the [Bij] rule, and relies on the axioms on f and f^{-1}. Formally, we prove:

$$f(z^*) \doteq y$$

Then, the second step in the proof is to derive from the above equality the equation:

$$f(mg^* \mid hr^*) \doteq y$$

The proof proceeds as follows: we use the [Prod] rule to establish that $mg^* \mid hr^* \doteq z^*$, and then the [Fun] rule to prove that $f(mg^* \mid hr^*) \doteq f(z^*)$, so by transitivity, we have $f(mg^* \mid hr^*) \doteq y$. Then, we can apply the [Linear] rule to conclude that

$$f((m \mid 0^{k_1}) \oplus g^* | hr^*) \doteq y$$

By a further application of the [Linear] rule, one concludes as expected that:

$$f((m \mid 0^{k_1}) \oplus g^* | H((m \mid 0^{k_1}) \oplus g^*) \oplus r^*) \doteq y$$

6.5 Towards Completeness

The purpose of this section is to define completeness, and to provide some partial results towards completeness. Unfortunately, we have not been able to prove completeness for any theory of interest.

Recall that a proof system is complete w.r.t. a set of models if all systems of equations that are valid in the models are also provable.

Definition 9 (Completeness). *Let $\mathbb{T} = (\Sigma, E)$ be a theory. The proof system is complete (resp. $\mathcal{D}$-complete) wrt an indexed family $(\mathbb{A}_i)_{i\in\mathcal{I}}$ of $\mathbb{T}$-algebras iff for every system of equations (resp. $\mathcal{D}$-equations) $\boldsymbol{s} \doteq \boldsymbol{t}$, if for all $i \in \mathcal{I}$, $\mathbb{A}_i \models \boldsymbol{s} \doteq \boldsymbol{t}$ then $\vdash \boldsymbol{s} \doteq \boldsymbol{t}$.*

There are two main issues with completeness. The first issue is the existence of products, which is discussed above. The second and main difficulty is the representation of bijections in the syntax. Indeed, one must show that the rule [Bij] does indeed provide a syntactic counterpart to Proposition 1. In other words, completeness requires that one can represent some bijections by a tuple of terms, so that the rule [Bij] applies. A stronger hypothesis, namely that all functions are representable by terms, is captured by the definition of primal algebra, which is used e.g. in [13]: an algebra $\mathbb{A}$ is primal iff for every function $f : \sigma_1 \times \ldots \times \sigma_n \to \tau$ (with $n > 0$, and $\sigma_1 \ldots \sigma_n, \tau$ interpretations of sorts) there

exists a $\mathcal{D}$-term u with free variables $x_1 : \sigma_1 \ \ldots \ x_n : \sigma_n$ such that for every $(a_1, \ldots, a_n) \in \sigma_1 \times \cdots \times \sigma_n$, we have:

$$[\![u]\!]_{(x_1 := a_1, \ldots, x_n := a_n)} = f(a_1, \ldots, a_n)$$

Unfortunately, the notion of primal algebra is too strong for our setting, because proving completeness would require that *all* the algebras of the indexed family $(\mathbb{A}_i)_{i \in \mathcal{I}}$ are primal. Since the size of the algebras is unbounded, it is not clear, even for the case of bitstrings considered in Section 5, how to define the signature so to meet this requirement. One can instead consider a weaker notion, called weak primality.

Definition 10 (Weakly primal). *An algebra $\mathbb{A}$ is weakly primal iff for every $f_1, f_2 : \sigma_1 \times \ldots \times \sigma_n \to \tau$ (with $n > 0$, and $\sigma_1 \ldots \sigma_n, \tau$ interpretations of sorts) that are interpretations of $\mathcal{D}$-terms, and for every bijection $h : \sigma_1 \times \ldots \times \sigma_n \to \sigma_1 \times \ldots \times \sigma_n$ such that $f_2 = f_1 \circ h$, there exist terms $u_1, \ldots, u_n$ with free variables $x_1, \ldots, x_n$ such that $f_2 = f_1 \circ [\![(u_1, \ldots, u_n)]\!]$, and $[\![(u_1, \ldots, u_n)]\!]$ is a bijection over $\sigma_1 \times \ldots \times \sigma_n$.*

Note that weak primality does not require that h is representable, but instead that there exist terms that satisfy the same equation as h. This weakening of the original definition is necessary to prove that weak primality holds for the signature of $\oplus$. The proof uses similar arguments to the proof of decidability of validity of equations, and yields a process to build the terms $u_1 \ldots u_n$. We illustrate the process on two examples: assume that $s = x_1 \oplus x_2 \oplus x_3$ and $t = x_2 \oplus x_3$. Then one takes the terms $u_1 = x_1$, $u_2 = x_1 \oplus x_2$, $u_3 = x_3$, which provide a bijection. Now, assume that $s = x_1 \oplus x_2 \oplus x_3$ and $t = x_3$. Then one takes the terms $u_1 = x_1$, $u_2 = x_2$, $u_3 = x_3 \oplus x_1 \oplus x_2$, which provide a bijection.

Weak primality is sufficient to prove that every valid equation (*not* system of equations) is derivable, provided that completeness holds for every $\mathcal{D}$-equation. The idea of the proof is as follows. Consider an equation $s \doteq t$ with deterministic variables $x_1 \ldots x_n$ of type $\sigma_1 \ldots \sigma_n$, and probabilistic variables $y_1 \ldots y_m$ of type $\tau_1 \ldots \tau_m$. Assume that for all $i \in \mathcal{I}$, $\mathbb{A}_i \models \boldsymbol{s} \doteq \boldsymbol{t}$. By Proposition 1, there exists a bijection $f_{a_1 \ldots a_n} : [\![\tau_1]\!] \times \ldots \times [\![\tau_m]\!] \to [\![\tau_1]\!] \times \ldots \times [\![\tau_m]\!]$ for every $(a_1 \ldots a_n) \in [\![\sigma_1]\!] \times \cdots \times [\![\sigma_n]\!]$ such that:

$$\langle\!\langle \boldsymbol{s} \rangle\!\rangle_{\boldsymbol{x} \mapsto \boldsymbol{a}, \boldsymbol{y} \mapsto \boldsymbol{b}} = \langle\!\langle \boldsymbol{t} \rangle\!\rangle_{\boldsymbol{x} \mapsto \boldsymbol{a}, \boldsymbol{y} \mapsto f_{a_1 \ldots a_n}(\boldsymbol{b})}$$

for every $(b_1, \ldots, b_m) \in [\![\tau]\!]$. By weak primality, there exist $\mathcal{D}$-terms $\boldsymbol{u}$ and $\boldsymbol{v}$ with free variables $x_1 \ldots x_n$ and $\bar{y_1} \ldots \bar{y_m}$ such that for every $a_1 \ldots a_n \ b_1 \ldots b_m$, we have:

- $f_{a_1 \ldots a_n}(b_1, \ldots, b_m) = \langle\!\langle \boldsymbol{u} \rangle\!\rangle_{\boldsymbol{x} \mapsto \boldsymbol{a}, \bar{\boldsymbol{y}} \mapsto \boldsymbol{b}}$,
- $f^{-1}_{a_1 \ldots a_n}(b_1, \ldots, b_m) = \langle\!\langle \boldsymbol{v} \rangle\!\rangle_{\boldsymbol{x} \mapsto \boldsymbol{a}, \bar{y} \mapsto b}$.

By $\mathcal{D}$-completeness, we have that $\vdash \boldsymbol{s} \doteq \boldsymbol{t}[\boldsymbol{y} := \boldsymbol{u}]$ and $\vdash \boldsymbol{u}[\bar{\boldsymbol{y}} := \boldsymbol{v}] \doteq \bar{\boldsymbol{y}}$, and hence by applying the [Bij] rule it follows that $\vdash \boldsymbol{s} \doteq \boldsymbol{t}$ is provable.

6.6 Proof Automation

One strategy for proving a system of equations is to apply the [Bij] rule so as to fall back in the realm of universal algebra—i.e. of $\mathcal{D}$-systems of equations. Thus, applicability of the [Bij] rule is the key to automation. This section considers conditions for automating the [Bij] rule. Our starting point is a mild generalization of the [Bij] rule, which does not require that all probabilistic variables are eliminated simulateneously (recall that V_D is a shorthand for $\mathsf{var}_{\mathcal{D}}(\boldsymbol{s}) \cup \mathsf{var}_{\mathcal{D}}(\boldsymbol{t})$):

$$\frac{\vdash \boldsymbol{s}[\boldsymbol{x} := \boldsymbol{u}] \doteq \boldsymbol{t}[\boldsymbol{x} := \bar{\boldsymbol{x}}] \qquad \vdash \boldsymbol{u}[\bar{\boldsymbol{x}} := \boldsymbol{v}] \doteq \bar{\boldsymbol{x}}}{\vdash \boldsymbol{s} \doteq \boldsymbol{t}}[\text{Bij}+]$$

where $V_D \cap \bar{\boldsymbol{x}} = \emptyset$ and $\mathsf{var}(\boldsymbol{u}) \cup \mathsf{var}(\boldsymbol{v}) \subseteq (\bar{\boldsymbol{x}} \cup V_D)$. The difference with the rule [Bij] is that the side condition $\mathsf{var}_{\mathcal{R}}(\boldsymbol{s}) \cup \mathsf{var}_{\mathcal{R}}(\boldsymbol{t}) \subseteq \boldsymbol{x}$ is dropped. Informally, this rule allows constructing the underlying bijection of Proposition 1 incrementally. It does not increase the power of the logic, but makes it easier to use, and is important for injectivity, as suggested below. There is a close connection between the rule [Bij+] and matching, see e.g. [1].

Definition 11 (1-1 matching problem). *Let s, t be two terms and $\boldsymbol{x} \subseteq \mathsf{var}_{\mathcal{R}}(s) \cup \mathsf{var}_{\mathcal{R}}(t)$. A solution to a 1-1 matching problem is a pair $(\boldsymbol{u}, \boldsymbol{v})$ of $\mathcal{D}$-terms such that:*

- $\mathsf{var}_{\mathcal{D}}(\boldsymbol{u}) \cup \mathsf{var}_{\mathcal{D}}(\boldsymbol{v}) \subseteq \bar{\boldsymbol{x}}$,
- $\vdash \boldsymbol{s}[\boldsymbol{x} := \boldsymbol{u}] \doteq \boldsymbol{t}[\boldsymbol{x} := \bar{\boldsymbol{x}}]$,
- $\vdash \boldsymbol{u}[\bar{\boldsymbol{x}} := \boldsymbol{v}] \doteq \bar{\boldsymbol{x}}$.

We let $\mathcal{S}ol(\boldsymbol{s} \prec\!\!\prec^{1-1}_{\boldsymbol{x}} \boldsymbol{t})$ denote the set of solutions.

The rule [Bij+] can be rephrased equivalently as:

$$\frac{\mathcal{S}ol(\boldsymbol{s} \prec\!\!\prec^{1-1}_{\boldsymbol{x}} \boldsymbol{t}) \neq \emptyset}{\vdash \boldsymbol{s} \doteq \boldsymbol{t}}$$

Hence, for every system of equations $\boldsymbol{s} \doteq \boldsymbol{t}$, we have that $\mathcal{S}ol(\boldsymbol{s} \prec\!\!\prec^{1-1}_{\boldsymbol{x}} \boldsymbol{t}) \neq \emptyset$ implies that for every $i \in \mathcal{I}$, we have $\mathbb{A}_i \models \boldsymbol{s} \doteq \boldsymbol{t}$. Thus, one can prove a system of equations $\boldsymbol{s} \doteq \boldsymbol{t}$ by exhibiting an element of $\mathcal{S}ol(\boldsymbol{s} \prec\!\!\prec^{1-1}_{\boldsymbol{x}} \boldsymbol{t})$.

Call a tuple of $\mathcal{D}$-terms $\boldsymbol{s}$ injective (w.r.t. variables $\boldsymbol{x}$ and theory E) iff for every $\boldsymbol{e}$, $\vdash \boldsymbol{s} \doteq \boldsymbol{s}[\boldsymbol{x} := \boldsymbol{e}]$ implies $\vdash \boldsymbol{e} \doteq \boldsymbol{x}$. Note that every vector of terms is injective whenever E has unitary matching. Moreover, for every single variable x and expression s in the theory of bitstrings, s is injective w.r.t. x, provided x is provably equal to $x \oplus s_0$, and x does not occur in s_0. On the other hand, one cannot prove injectivity for terms that contain two variables x and y: indeed, let s be $x \oplus y$. Then for every constant bitstring c one can derive $x \oplus y \doteq x \oplus y[x, y := x \oplus c, y \oplus c]$ whereas we do not have $x \doteq x \oplus c$ and $y \doteq y \oplus c$. This explains why it is important to consider the rule [Bij+] instead of [Bij].

The rule [Match] below, that uses the notion of injective term as a side condition, is derivable:

$$\frac{\vdash \boldsymbol{s}[\boldsymbol{x} := \boldsymbol{u}] \doteq \boldsymbol{t}[\boldsymbol{x} := \bar{\boldsymbol{x}}] \qquad \vdash \boldsymbol{s}[\boldsymbol{x} := \bar{\boldsymbol{x}}] \doteq \boldsymbol{t}[\boldsymbol{x} := \boldsymbol{v}]}{\vdash \boldsymbol{s} \doteq \boldsymbol{t}}[\text{Match}] \text{if } \boldsymbol{s} \text{ injective w.r.t. } \boldsymbol{x}$$

Assume that $\vdash \boldsymbol{s}[\boldsymbol{x} := \boldsymbol{u}] \doteq \boldsymbol{t}[\boldsymbol{x} := \bar{\boldsymbol{x}}]$ and $\vdash \boldsymbol{s}[\boldsymbol{x} := \bar{\boldsymbol{x}}] \doteq \boldsymbol{t}[\boldsymbol{x} := \boldsymbol{v}]$. Then, $\vdash \boldsymbol{s}[\boldsymbol{x} := \boldsymbol{u}][\bar{\boldsymbol{x}} := \boldsymbol{v}] \doteq \boldsymbol{t}[\bar{\boldsymbol{x}} := \boldsymbol{v}]$ by substitution. That is, $\vdash \boldsymbol{s}[\boldsymbol{x} := \boldsymbol{u}[\bar{\boldsymbol{x}} := \boldsymbol{v}]] \doteq \boldsymbol{t}[\bar{\boldsymbol{x}} := \boldsymbol{v}]$. By transitivity, $\vdash \boldsymbol{s}[\boldsymbol{x} := \boldsymbol{u}[\bar{\boldsymbol{x}} := \boldsymbol{v}]] \doteq \boldsymbol{s}$ and by injectivity $\vdash \boldsymbol{u}[\bar{\boldsymbol{x}} := \boldsymbol{v}] \doteq \bar{\boldsymbol{x}}$. One concludes by applying the rule [Bij+].

Thus, one can automate proofs in our logic by performing matching on injective terms.

7 Conclusion

We have considered a mild extension of universal algebra in which variables are given a probabilistic interpretation. We have given a sound proof system and useful heuristics to carry equational reasoning between such probabilistic terms; moreover, we have provided decision procedures for specific theories that arise commonly in cryptographic proofs.

Related work. Equational logic [10] is a well-established research field, and there has been substantial work to develop proof systems and decision procedures for differents flavours of the logic: many-sorted, multi-sorted, etc. Yet there seems to have been few works that consider probabilistic extensions of equational logic; for example, P-Maude [12] is an extension of Maude that supports probabilistic rewrite theories, an extension of term rewriting where a term rewrites to another term with a given probability. However, none of these works seems to have been motivated by cryptography.

Equational theories have been thoroughly studied in the setting of cryptographic protocols; see e.g. [7]. In particular, computational and probabilisitc semantics for an equational theory of exclusive or is given, in the context of a more general approach to such semantics for general equational theories, is given in [3]. However, this work does not consider equational logics with probabilistic terms.

Future work. The formalism of probabilistic terms seems new and deserves further investigation in its own right. It would be interesting to develop further the proof theory of probabilistic terms, and in particular to establish sufficient conditions for completeness and decidability. In addition, it seems relevant to study its relationship with other probabilistic extensions of equational logic, such as P-Maude [12]. The connection between matching and 1-1 matching also deserves further attention.

Our work is part of a larger effort to carry a proof-theoretical study of logical methods for cryptographic proofs, and our main focus will be to exploit our results in cryptography. Further work is required to extend the scope of our results to other theories of interest for cryptography, see e.g. [7], including permutations, exponentiation, etc. We also intend to extend our results to (loop-free) probabilistic programs, and to develop automated proof methods to decide observational equivalence between such programs. A further step would be to consider, instead of observational equivalence, a notion of statistical distance between programs and to develop automated approximation methods.

In the long term, our goal is to implement our methods and integrate them in tools to reason about cryptographic schemes and protocols, e.g. CertiCrypt [2], or our automated tool to reason about encryption [8].

Acknowledgements. The authors are grateful to the anonymous reviewers for their comments, and to Santiago Escobar, Pascal Lafourcade, and José Meseguer for interesting discussions.

References

1. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press, Cambridge (1998)
2. Barthe, G., Grégoire, B., Béguelin, S.Z.: Formal certification of code-based cryptographic proofs. In: Proceedings of POPL 2009, pp. 90–101. ACM Press, New York (2009)
3. Baudet, M., Cortier, V., Kremer, S.: Computationally sound implementations of equational theories against passive adversaries. Inf. Comput. 207(4), 496–520 (2009)
4. Bellare, M., Rogaway, P.: The security of triple encryption and a framework for code-based game-playing proofs. In: Vaudenay, S. (ed.) EUROCRYPT 2006. LNCS, vol. 4004, pp. 409–426. Springer, Heidelberg (2006)
5. Bellare, M., Rogaway, P.: Optimal asymmetric encryption – How to encrypt with RSA. In: De Santis, A. (ed.) EUROCRYPT 1994. LNCS, vol. 950, pp. 92–111. Springer, Heidelberg (1995)
6. Bresson, E., Lakhnech, Y., Mazaré, L., Warinschi, B.: A generalization of ddh with applications to protocol analysis and computational soundness. In: Menezes, A. (ed.) CRYPTO 2007. LNCS, vol. 4622, pp. 482–499. Springer, Heidelberg (2007)
7. Cortier, V., Delaune, S., Lafourcade, P.: A survey of algebraic properties used in cryptographic protocols. Journal of Computer Security 14(1), 1–43 (2006)
8. Courant, J., Daubignard, M., Ene, C., Lafourcade, P., Lakhnech, Y.: Towards automated proofs for asymmetric encryption schemes in the random oracle model. In: Proceedings of CCS 2008, pp. 371–380. ACM Press, New York (2008)
9. Fujisaki, E., Okamoto, T., Pointcheval, D., Stern, J.: RSA-OAEP is secure under the RSA assumption. Journal of Cryptology 17(2), 81–104 (2004)
10. Goguen, J.A., Meseguer, J.: Completeness of many-sorted equational logic. SIGPLAN Not. 16(7), 24–32 (1981)
11. Impagliazzo, R., Kapron, B.M.: Logics for reasoning about cryptographic constructions. Journal of Computer and Systems Sciences 72(2), 286–320 (2006)
12. Kumar, N., Sen, K., Meseguer, J., Agha, G.: A rewriting based model for probabilistic distributed object systems. In: Najm, E., Nestmann, U., Stevens, P. (eds.) FMOODS 2003. LNCS, vol. 2884, pp. 32–46. Springer, Heidelberg (2003)
13. Nipkow, T.: Unification in primal algebras, their powers and their varieties. J. ACM 37(4), 742–776 (1990)
14. Shoup, V.: A Computational Introduction to Number Theory and Algebra. Cambridge University Press, Cambridge (2008)
15. Stern, J.: Why provable security matters? In: Biham, E. (ed.) EUROCRYPT 2003. LNCS, vol. 2656, pp. 449–461. Springer, Heidelberg (2003)

Program Logics for Homogeneous Meta-programming

Martin Berger[1] and Laurence Tratt[2]

[1] University of Sussex
[2] Middlesex University

Abstract. A meta-program is a program that generates or manipulates another program; in homogeneous meta-programming, a program may generate new parts of, or manipulate, itself. Meta-programming has been used extensively since macros were introduced to Lisp, yet we have little idea how formally to reason about meta-programs. This paper provides the first program logics for homogeneous meta-programming – using a variant of MiniML$_e^{\Box}$ by Davies and Pfenning as underlying meta-programming language. We show the applicability of our approach by reasoning about example meta-programs from the literature. We also demonstrate that our logics are relatively complete in the sense of Cook, enable the inductive derivation of characteristic formulae, and exactly capture the observational properties induced by the operational semantics.

1 Introduction

Meta-programming is the generation or manipulation of programs, or parts of programs, by other programs, i.e. in an algorithmic way. Meta-programming is commonplace, as evidenced by the following examples: compilers, compiler generators, domain specific language hosting, extraction of programs from formal specifications, and refactoring tools.

Many programming languages, going back at least as far as Lisp, have explicit meta-programming features. These can be classified in various ways such as: generative (program creation), intensional (program analysis), compile-time (happening while programs are compiled), run-time (taking place as part of program execution), heterogeneous (where the system generating or analysing the program is different from the system being generated or analysed), homogeneous (where the systems involved are the same), and lexical (working on simple strings) or syntactical (working on abstract syntax trees). Arguably the most common form of meta-programming is reflection, supported by mainstream languages such as Java, C#, and Python. Web system languages such as PHP use meta-programming to produce web pages containing JavaScript; JavaScript (in common with some other languages) does meta-programming by dynamically generating strings and then executing them using its `eval` function. In short, meta-programming is a mainstream activity.

An important type of meta-programming is generative meta-programming, specifically homogeneous meta-programming. The first language to support homogeneous meta-programming was Lisp with its S-expression based macros; Scheme's macros improve upon Lisp's by being fully hygienic, but are conceptually similar. Perhaps unfortunately, the power of Lisp-based macros was long seen to rest largely on Lisp's minimalistic syntax; it was not until MetaML [17] that a modern, syntactically rich language

E.M. Clarke and A. Voronkov (Eds.): LPAR-16, LNAI 6355, pp. 64–81, 2010.

was shown to be capable of homogeneous meta-programming. Since then, MetaOCaml [18] (a descendant of MetaML), Template Haskell [16] (a pure compile-time meta-programming language) and Converge [19] (inspired by Template Haskell and adding features for embedding domain specific languages) have shown that a variety of modern programming languages can house homogeneous generative meta-programming, and that this allows powerful, safe programming of a type previously impractical or impossible.

Meta-Programming & Verification. The ubiquity of meta-programming demonstrates its importance; by extension, it means that the correctness of much modern software depends on the correct use of meta-programming. Surprisingly, correctness and meta-programming have received little joint attention. In particular, there seem to be no program logics for meta-programming languages, homogeneous or otherwise. We believe that the following reasons might be partly responsible.

- First, developing logics for non-meta-programming programming languages is already a hard problem, and only recently have satisfactory solutions been found for reasoning about programs with higher-order functions, state, pointers, continuations or concurrency [1, 14, 21]. Since reasoning about meta-programs contains reasoning about normal programs as a special case, program logics for meta-programming are at least as complicated as logics for normal languages.
- Second, it is often possible to side-step the question of meta-programming correctness altogether by considering only the final product of meta-programming. Compilation is an example where the meta-programming machinery is typically much more complex than the end product. Verifying only the output of a meta-programming process is inherently limited, because knowledge garnered from the input to the process cannot be used.
- Finally, static typing of meta-programming is challenging, and still not a fully solved problem. Consequently, most meta-programming languages are at least partly dynamically typed (including MetaOCaml); Template Haskell on the other hand intertwines code generation with type-checking in complicated ways. Logics for such languages are not well understood in the absence of other meta-programming features; moreover, many meta-programming languages have additional features such as capturing substitution, pattern matching of code, and splicing of types, which are largely unexplored theoretically. Heterogeneous meta-programming adds the complication of multi-language verification.

Contributions. The present paper is the first in a series investigating the use of program logics for the specification and verification of meta-programming. The aim of the series is to unify and relate all key meta-programming concepts using program logics. One of our goals is to achieve coherency between existing logics for programming languages and their meta-programming extensions (i.e. the former should be special cases of the latter). The contributions of this paper are as follows (all proofs have been omitted in the interests of brevity; they can be found in [4]):

- We provide the first program logic for a generative, homogeneous meta-programming language (PCF_{DP}, a variant of Davies and Pfenning's $\text{MiniML}^{\Box}_{e}$ [6],

itself an extension of PCF [8]). The logic smoothly generalises previous work on axiomatic semantics for the ML family of languages [1,2,9,11,12,21]. The logic is for total correctness (for partial correctness see [4]). A key feature of our logic is that for PCF_{DP} programs that don't do meta-programming, i.e. programs in the PCF-fragment, reasoning can be done in the simpler logic [9,11] for PCF. Hence reasoning about meta-programming does not impose an additional burden on reasoning about normal programs.
- We show that our logic is relatively complete in the sense of Cook [5] (Section 5).
- We demonstrate that the axiomatic semantics induced by our logic coincides precisely with the contextual semantics given by the reduction rules of PCF_{DP} (Section 5).
- We present an additional inference system for characteristic formulae which enables, for each program M, the inductive derivation of a pair A, B of formulae which describe completely M's behaviour (descriptive completeness [10], Section 5).

2 The Language

This section introduces PCF_{DP}, the homogeneous meta-programming language that is the basis of our study. PCF_{DP} is a meta-programming variant of call-by-value (CBV) PCF [8], extended with the meta-programming features of Mini-$\text{ML}_e^{\Box}$ [6, Section 3]. From now on we will simply speak of PCF to mean CBV PCF. Mini-$\text{ML}_e^{\Box}$ was the first typed homogeneous meta-programming language to provide a facility for executing generated code. Typing the execution of generated code is a difficult problem. Mini-$\text{ML}_e^{\Box}$ achieves type-safety with two substantial restrictions:

- Only closed generated code can be executed.
- Variables free in code cannot be λ-abstracted or be recursion variables.

Mini-$\text{ML}_e^{\Box}$ was one of the first meta-programming languages with a Curry-Howard correspondence, although this paper does not investigate the connection between program logic and the Curry-Howard correspondence. PCF_{DP} is essentially Mini-$\text{ML}_e^{\Box}$, but with a slightly different form of recursion that can be given a moderately simpler logical characterisation. PCF_{DP} is an ideal vehicle for our investigation for two reasons. First, it is designed to be a simple, yet non-trivial meta-programming language, having all key features of homogeneous generative meta-programming while removing much of the complexity of real-world languages (for example, PCF_{DP}'s operational semantics is substantially simpler than that of the MetaML family of languages [17]). Second, it is built on top of PCF, a well-understood idealised programming language with existing program logics [9,11], which means that we can compare reasoning in the PCF-fragment with reasoning in full PCF_{DP}.

PCF_{DP} extends PCF with one new type $\langle\alpha\rangle$ as well as two new term constructors, quasi-quotes $\langle M\rangle$ and an unquote mechanism $\texttt{let}\ \langle x\rangle = M\ \texttt{in}\ N$. *Quasi-quotes* $\langle M\rangle$ represent the code of M, and allow code fragments to be simply expressed in normal concrete syntax; quasi-quotes also provide additional benefits such as ensuring hygiene. The quasi-quotes we use in this paper are subtly different from the abstract syntax trees (ASTs) used in languages like Template Haskell and Converge. In such languages, ASTs are a distinct data type, shadowing the conventional language feature hierarchy.

In this paper, if M has type α, then $\langle M \rangle$ is typed $\langle \alpha \rangle$. For example, $\langle 1+7 \rangle$ is the code of the program $1+7$ of type $\langle \mathsf{Int} \rangle$. $\langle M \rangle$ is a value for all M and hence $\langle 1+7 \rangle$ does not reduce to $\langle 8 \rangle$. Extracting code from a quasi-quote is the purpose of the *unquote* $\mathtt{let}\ \langle x \rangle = M\ \mathtt{in}\ N$. It evaluates M to code $\langle M' \rangle$, extracts M' from the quasi-quote, names it x and makes M' available in N without reducing M'. For example

$$\mathtt{let}\ \langle x \rangle = (\lambda z.z)\langle 1+7 \rangle\ \mathtt{in}\ \langle \lambda n.x^n \rangle$$

first reduces the application to $\langle 1+7 \rangle$, then extracts the code from $\langle 1+7 \rangle$, names it x and makes it available unevaluated to the code $\langle \lambda n.x^n \rangle$:

$$\begin{array}{lcl} \mathtt{let}\ \langle x \rangle = (\lambda z.z)\langle 1+7 \rangle\ \mathtt{in}\ \langle \lambda n.x^n \rangle & \to & \mathtt{let}\ \langle x \rangle = \langle 1+7 \rangle\ \mathtt{in}\ \langle \lambda n.x^n \rangle \\ & \to & \langle \lambda n.x^n \rangle[1+7/x] \\ & = & \langle \lambda n.(1+7)^n \rangle \end{array}$$

$$\frac{(\Gamma \cup \Delta)(x) = \alpha}{\Gamma;\Delta \vdash x : \alpha} \qquad \frac{\Gamma, x : \alpha; \Delta \vdash M : \beta}{\Gamma;\Delta \vdash \lambda x^{\alpha}.M : \alpha \to \beta} \qquad \frac{\Gamma;\Delta \vdash M : \alpha \to \beta \quad \Gamma;\Delta \vdash N : \alpha}{\Gamma;\Delta \vdash MN : \beta}$$

$$\frac{\Gamma, f : (\alpha \to \beta); \Delta \vdash \lambda x^{\alpha}.M : \alpha \to \beta}{\Gamma;\Delta \vdash \mu f^{\alpha \to \beta}.\lambda x^{\alpha}.M : \alpha \to \beta} \qquad \frac{\Gamma;\Delta \vdash M : \mathsf{Bool} \quad \Gamma;\Delta \vdash N : \alpha \quad \Gamma;\Delta \vdash N' : \alpha}{\Gamma;\Delta \vdash \mathtt{if}\ M\ \mathtt{then}\ N\ \mathtt{else}\ N' : \alpha}$$

$$\frac{\Gamma;\Delta \vdash M : \mathsf{Int} \quad \Gamma;\Delta \vdash N : \mathsf{Int}}{\Gamma;\Delta \vdash M + N : \mathsf{Int}} \qquad \frac{\varepsilon;\Delta \vdash M : \alpha}{\Gamma;\Delta \vdash \langle M \rangle : \langle \alpha \rangle} \qquad \frac{\Gamma;\Delta \vdash M : \langle \alpha \rangle \quad \Gamma;\Delta, x : \alpha \vdash N : \beta}{\Gamma;\Delta \vdash \mathtt{let}\ \langle x \rangle = M\ \mathtt{in}\ N : \beta}$$

Fig. 1. Key typing rules for PCF_{DP}

Not evaluating code after extraction from a quasi-quote is fundamental to meta-programming because it enables the construction of code *other than values* under λ-abstractions. This is different from the usual reduction strategies of CBV-calculi — Section 6 discusses briefly how PCF_{DP} might nevertheless be embeddable into PCF. Unquote can also be used to run a meta-program: if M evaluates to a quasi-quote $\langle N \rangle$, the program $\mathtt{let}\ \langle x \rangle = M\ \mathtt{in}\ x$ evaluates M, extracts N, binds N to x and then runs x, i.e. N. In this sense, PCF_{DP}'s unquote mechanism unifies splicing and executing quasi-quoted code, where the MetaML family of languages uses different primitives for these two functions [17].

Syntax and Types. We now formalise PCF_{DP}'s syntax and semantics, assuming a set of variables, ranged over by $x, y, f, u, m, \ldots$ (for more details see [6,8]).

$$\begin{array}{lcl} \alpha & ::= & \mathsf{Unit} \mid \mathsf{Bool} \mid \mathsf{Int} \mid \alpha \to \beta \mid \langle \alpha \rangle \\ V & ::= & \mathtt{c} \mid x \mid \lambda x^{\alpha}.M \mid \mu f^{\alpha \to \beta}.\lambda x^{\alpha}.M \mid \langle M \rangle \\ M & ::= & V \mid \mathtt{op}(\tilde{M}) \mid MN \mid \mathtt{if}\ M\ \mathtt{then}\ N\ \mathtt{else}\ N' \mid \mathtt{let}\ \langle x \rangle = M\ \mathtt{in}\ N \end{array}$$

Here α ranges over *types*, V over *values* and M ranges over *programs*. Constants $\mathtt{c}$ range over the integers $0, 1, -1, \ldots$, booleans $\mathtt{t}, \mathtt{f}$, and $()$ of type Unit, $\mathtt{op}$ ranges over

the usual first-order operators like addition, multiplication, equality, conjunction, negation, comparison, etc., with the restriction that equality is *not* defined on expressions of function type or of type $\langle\alpha\rangle$. The recursion operator is $\mu f.\lambda x.M$. The *free variables* $\mathsf{fv}(M)$ of M are defined as usual with two new clauses: $\mathsf{fv}(\langle M\rangle) \stackrel{def}{=} \mathsf{fv}(M)$ and $\mathsf{fv}(\texttt{let}\ \langle x\rangle = M\ \texttt{in}\ N) \stackrel{def}{=} \mathsf{fv}(M) \cup (\mathsf{fv}(N) \setminus \{x\})$. We write $\lambda().M$ for $\lambda x^{\mathsf{Unit}}.M$ and $\texttt{let}\ x = M\ \texttt{in}\ N$ for $(\lambda x.N)M$, assuming that $x \notin \mathsf{fv}(M)$ in both cases. A *typing environment* $(\Gamma, \Delta, ...)$ is a finite map $x_1 : \alpha_1, ..., x_k : \alpha_k$ from variables to types. The *domain* $\mathsf{dom}(\Gamma)$ of Γ is the set $\{x_1, ..., x_n\}$, assuming that Γ is $x_1 : \alpha_1, ..., x_n : \alpha_n$. We also write ε for the empty environment. The *typing judgement* is written $\Gamma;\Delta \vdash M : \alpha$ where we assume that $\mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Delta) = \emptyset$. We write $\vdash M : \alpha$ for $\varepsilon;\varepsilon \vdash M : \alpha$. We say a term M is *closed* if $\vdash M : \alpha$. We call Δ a *modal context* in $\Gamma;\Delta \vdash M : \alpha$. We say a variable x is *modal* in $\Gamma;\Delta \vdash M : \alpha$ if $x \in \mathsf{dom}(\Delta)$. Modal variables represent code inside other code, and code to be run. The key type-checking rules are given in Figure 1. Typing for constants and first-order operations is standard.

Noteworthy features of the typing system are that modal variables cannot be λ- or μ-abstracted, that all free variables in quasi-quotes must be modal, and that modal variables can only be generated by unquotes. [6] gives detailed explanations of this typing system and its relationship to modal logics.

The *reduction relation* $\rightarrow$ is unchanged from PCF for the PCF-fragment of PCF_{DP}, and adapted to PCF_{DP} as follows. First we define define *reduction contexts*, by extending those for PCF as follows.

$$\mathcal{E}[\cdot] \ ::= \ ... \mid \texttt{let}\ \langle x\rangle = \mathcal{E}[\cdot]\ \texttt{in}\ M$$

Now $\rightarrow$ is defined as usual on *closed* terms with one new rule.

$$\texttt{let}\ \langle x\rangle = \langle M\rangle\ \texttt{in}\ N \rightarrow N[M/x]$$

We write $\twoheadrightarrow$ for $\rightarrow^*$. $M \Downarrow V$ means that $M \twoheadrightarrow V$ for some value V. We write $M \Downarrow$ if $M \Downarrow V$ for some appropriate V.

By $\sqsubseteq_{\Gamma;\Delta;\alpha}$ (usually abbreviated to just $\sqsubseteq$) we denote the usual *typed contextual precongruence*: if $\Gamma;\Delta \vdash M_i : \alpha$ for $i = 1, 2$ then: $M_1 \sqsubseteq_{\Gamma;\Delta;\alpha} M_2$ iff for all closing context $C[\cdot]$ such that $\vdash C[M_i] : \mathsf{Unit}$ $(i = 1, 2)$ we have $C[M_1] \Downarrow$ implies $C[M_2] \Downarrow$. We write $\simeq$ for $\sqsubseteq \cap \sqsubseteq^{-1}$ and call $\simeq$ *contextual congruence*. Other forms of congruence are possible. Our choice means that code can only be observed contextually by running it. Hence for example $\langle M\rangle$ and $\langle \lambda x.Mx\rangle$ are contextually indistinguishable if $x \notin \mathsf{fv}(M)$, as are $\langle 1+2\rangle$ and $\langle 3\rangle$. This coheres well with the notion of equality in PCF, and facilitates a smooth integration of the logics for PCF_{DP} with the logics for PCF. Some meta-programming languages are more discriminating, allowing, e.g. printing of code, which can distinguish α-equivalent programs. It is unclear how to design logics for such languages.

Examples. We now present classic generative meta-programs in PCF_{DP}. We reason about some of these programs in later sections.

The first example is from [6] and shows how generative meta-programming can *stage* computation, which can be used for powerful domain-specific optimisations. As an example, consider staging an exponential function $\lambda n.a^n$. It is generally more efficient to run $\lambda a.a \times a \times a$ than $(\lambda n.a^n)\ 3$, because the recursion required in computing a^n for

arbitrary n can be avoided. Thus if a program contains many applications $(\lambda na.a^n)\ 3$, it makes sense to specialise such applications to $\lambda a.a \times a \times a$. Meta-programming can be used to generate such specialised code as the following example shows.

$$\mathsf{power} \stackrel{def}{=} \mu p.\lambda n.\mathtt{if}\ n \leq 0\ \mathtt{then}\ \langle \lambda x.1 \rangle\ \mathtt{else}\ \mathtt{let}\ \langle q \rangle = p(n-1)\ \mathtt{in}\ \langle \lambda x.x \times (q\ x) \rangle$$

This function has type $\vdash \mathsf{power} : \mathsf{Int} \to \langle \mathsf{Int} \to \mathsf{Int} \rangle$. This type says that power takes an integer and returns code. That code, when run, is a function from integers to integers. More efficient specialisers are possible. This program can be used as follows.

$$\mathsf{power}\ 2 \quad \twoheadrightarrow \quad \langle \lambda a.a \times ((\lambda b.b \times ((\lambda c.1)b))a) \rangle$$

The next example is *lifting* [6], which plays a vital role in meta-programming. Call a type α *basic* if it does not contain the function space constructor, i.e. if it has no subterms of the form $\beta \to \beta'$. The purpose of lifting is to take an arbitrary value V of basic type α, and convert (lift) it to code $\langle V \rangle$ of type $\langle \alpha \rangle$. Note that we cannot simply write $\lambda x.\langle x \rangle$ because modal variables (i.e. variables free in code) cannot be λ-abstracted. For $\alpha = \mathsf{Int}$ the function is defined as follows:

$$\mathsf{lift}_{\mathsf{Int}} \stackrel{def}{=} \mu g.\lambda n^{\mathsf{Int}}.\mathtt{if}\ n \leq 0\ \mathtt{then}\ \langle 0 \rangle\ \mathtt{else}\ \mathtt{let}\ \langle x \rangle = g(n-1)\ \mathtt{in}\ \langle x+1 \rangle.$$

Note that $\mathsf{lift}_{\mathsf{Int}}\ 3$ evaluates to $\langle 0+1+1+1 \rangle$, not $\langle 3 \rangle$. In more expressive meta-programming languages such as Converge the corresponding program would evaluate to $\langle 3 \rangle$, which is more efficient, although $\langle 0+1+1+1 \rangle$ and $\langle 3 \rangle$ are observationally indistinguishable.

Lifting easily extended to Unit and Bool, but not to function types. For basic types $\langle \alpha \rangle$ we can use the following definition:

$$\mathsf{lift}_{\langle \alpha \rangle} \stackrel{def}{=} \lambda x^{\langle \alpha \rangle}.\mathtt{let}\ \langle a \rangle = x\ \mathtt{in}\ \langle \langle a \rangle \rangle$$

PCF_{DP} can implement a function of type $\langle \alpha \rangle \to \alpha$ for running code [6]:

$$\mathsf{eval} \stackrel{def}{=} \lambda x^{\langle \alpha \rangle}.\mathtt{let}\ \langle y \rangle = x\ \mathtt{in}\ y.$$

3 A Logic for Total Correctness

This section defines the syntax and semantics of the logic. Our logic is a Hoare logic with pre- and post-conditions in the tradition of logics for ML-like languages [1, 2, 11, 12]. *Expressions*, ranged over by $e, e', \ldots$ and *formulae* $A, B, \ldots$ of the logic are given by the grammar below, using the types and variables of PCF.

$$\begin{array}{lcl} e & ::= & \mathsf{c} \mid x \mid \mathsf{op}(\tilde{e}) \\ A & ::= & e = e' \mid \neg A \mid A \wedge B \mid \forall x^{\alpha}.A \mid u \bullet e = m\{A\} \mid u = \langle m \rangle \{A\} \end{array}$$

The logical language is based on standard first-order logic with equality. Other quantifiers and propositional connectives like $\supset$ (implication) are defined by de Morgan duality. Quantifiers range over values of appropriate type. Constants c and operations op are those of Section 2.

The proposed logic extends the logic for PCF of [9,10,11] with a new *code evaluation* primitive $u = \langle m \rangle \{A\}$. It says that u, which must be of type $\langle \alpha \rangle$, denotes (up to contextual congruence) a quasi-quoted program $\langle M \rangle$, such that whenever M is executed, it converges to a value; if that value is denoted by m then A makes a true statement about that value. We recall from [9,10,11] that $u \bullet e = m\{A\}$ says that (assuming u has function type) u denotes a function, which, when fed with the value denoted by e, terminates and yields another value. If we name this latter value m, A holds. We call the variable m in $u \bullet e = m\{A\}$ and $u = \langle m \rangle \{A\}$ an *anchor*. The anchor is a bound variable with scope A. The *free variables* of e and A, written $\mathsf{fv}(e)$ and $\mathsf{fv}(A)$, respectively, are defined as usual noting that $\mathsf{fv}(u = \langle m \rangle \{A\}) \stackrel{def}{=} (\mathsf{fv}(A) \setminus \{m\}) \cup \{u\}$. We use the following abbreviations: $A^{\neg x}$ indicates that $x \notin \mathsf{fv}(A)$ while $e \Downarrow$ means $\exists x^{\alpha}.e = x$, assuming that e has type α. We let $m = \langle e \rangle$ be short for $m = \langle x \rangle \{x = e\}$ where x is fresh, $m \bullet e = e'$ abbreviates $m \bullet e = x\{x = e'\}$ where x is fresh. We often omit typing annotations in expressions and formulae.

Judgements (for total correctness) are of the form $\{A\}\ M :_m \{B\}$. The variable m is the *anchor* of the judgement, is a bound variable with scope B, and not modal. The judgement is to be understood as follows: if A holds, then M terminates to a value, and if we denote that value m, then B holds. If a variable x occurs freely in A or in B, but not in M, then x is an *auxiliary variable* of the judgement $\{A\}\ M :_m \{B\}$. With environments as in Section 2, the *typing judgements* for expressions and formulae are $\Gamma; \Delta \vdash e : \alpha$ and $\Gamma; \Delta \vdash A$, respectively. The typing rules are given in Appendix A, together with the typing of judgements. The anchor in $u = \langle m \rangle \{A\}$ is modal, while it is not modal in $u \bullet e = m\{A\}$.

The logic has the following noteworthy features. (1) Quantification is not directly available on modal variables. (2) Equality is possible between modal and non-modal variables. The restriction on quantification makes the logic weaker for modal variables than first-order logic. Note that if x is modal in A we can form $\forall y.(x = y \supset A)$, using an equation between a modal and a non-modal variable. Quantification over all variables is easily defined by extending the grammar with a *modal quantifier* which ranges over arbitrary programs, not just values:

$$A \quad ::= \quad \ldots \mid \forall x^{\Box\alpha}.A$$

For $\forall x^{\Box\alpha}.A$ to be well-formed, x must be modal and of type α in A. The existential modal quantifier is given by de Morgan duality. Modal quantification is used only in the metalogical results of Section 5. We sometimes drop type annotations in quantifiers, e.g. writing $\forall x.A$. This shorthand will *never* be used for modal quantifiers. We abbreviate modal quantification to $\forall x^{\Box}.A$. *From now on, we assume all occurring programs, expressions, formulae and judgements to be well-typed.*

Examples of Assertions & Judgements. We continue with a few simple examples to motivate the use of our logic.

- The assertion $m = \langle 3 \rangle$, which is short for $m = \langle x \rangle \{x = 3\}$ says that m denotes code which, when executed, will evaluate to 3. It can be used to assert on the program $\langle 1 + 2 \rangle$ as follows: $\{\mathsf{T}\}\ \langle 1 + 2 \rangle :_m \{m = \langle 3 \rangle\}$

- Let Ω_α be a non-terminating program of type α (we usually drop the type subscript). When we quasi-quote Ω, the judgement $\{\mathsf{T}\}\ \langle\Omega\rangle :_m \{\mathsf{T}\}$ says (*qua* precondition) that $\langle\Omega\rangle$ is a terminating program. Indeed, that is the strongest statement we can make about $\langle\Omega\rangle$ in a logic for total correctness.
- The assertion $\forall x^{\mathsf{Int}}.m \bullet x = y\{y = \langle x\rangle\}$ says that m denotes a terminating function which receives an integer and returns code which evaluates to that integer. Later we use this assertion when reasoning about $\mathsf{lift}_{\mathsf{Int}}$ which has the following specification.

$$\{\mathsf{T}\}\ \mathsf{lift}_{\mathsf{Int}} :_u \{\forall n.n \geq 0 \supset u \bullet n = m\{m = \langle n\rangle\}\}$$

- The formula $A_u \stackrel{def}{=} \forall n^{\mathsf{Int}} \geq 0.\exists f^{\mathsf{Int}\to\mathsf{Int}}.(u \bullet n = \langle f\rangle \wedge \forall x^{\mathsf{Int}}.f \bullet x = x^n)$ says that u denotes a function which receives an integer n as argument, to return code which when evaluated and fed another integer x, computes the power x^n, provided $n \geq 0$. We can show that $\{\mathsf{T}\}\ \mathsf{power} :_u \{A_u\}$ and $\{A_u\}\ u\ 7 :_r \{r = \langle f\rangle\{\forall x.f \bullet x = x^7\}\}$.
- The formula $\forall x^{\langle\alpha\rangle} y^\alpha.(x = \langle y\rangle \supset u \bullet x = y)$ can be used to specify the evaluation function from Section 2: $\{\mathsf{T}\}\ \mathsf{eval} :_u \{\forall x^{\langle\alpha\rangle} y^\alpha.(x = \langle y\rangle \supset u \bullet x = y)\}$

Models and the Satisfaction Relation. This section formally presents the semantics of our logic. We begin with the notion of model. The key difference from the models of [11] is that modal variables denote possibly non-terminating programs.

$(e1)$	$x \bullet y = z\{A\} \wedge x \bullet y = z\{B\}$	$\equiv$	$x \bullet y = z\{A \wedge B\}$	
$(e2)$	$x \bullet y = z\{\neg A\}$	$\supset$	$\neg x \bullet y = z\{A\}$	
$(e3)$	$x \bullet y = z\{A\} \wedge \neg x \bullet y = z\{B\}$	$\equiv$	$x \bullet y = z\{A \wedge \neg B\}$	
$(e4)$	$x \bullet y = z\{A \wedge B\}$	$\equiv$	$A \wedge x \bullet y = z\{B\}$	$z \notin \mathsf{fv}(A)$
$(e5)$	$x \bullet y = z\{\forall a^\alpha.A\}$	$\equiv$	$\forall a^\alpha.x \bullet y = z\{A\}$	$a \neq x,y,z$
$(e6)$	$(A \supset B) \wedge x \bullet y = z\{A\}$	$\supset$	$x \bullet y = z\{B\}$	$z \notin \mathsf{fv}(A,B)$
(ext)	$x = y$	$\equiv$	$\mathsf{Ext}(xy)$	

Fig. 2. Key total correctness axioms for $\mathrm{PCF}_{\mathrm{DP}}$

Let Γ,Δ be two contexts with disjoint domains (the idea is that Δ is modal while Γ is not). A *model* of type $\Gamma;\Delta$ is a pair (ξ,σ) where ξ is a map from $\mathsf{dom}(\Gamma)$ to closed *values* such that $\vdash \xi(x) : \Gamma(x)$; σ is a map from $\mathsf{dom}(\Delta)$ to closed *programs* $\vdash \sigma(x) : \Delta(x)$. We also write $(\xi,\sigma)^{\Gamma;\Delta}$ to indicate that (ξ,σ) is a model of type $\Gamma;\Delta$. We write $\xi \cdot x : V$ for $\xi \cup \{(x,V)\}$ assuming that $x \notin \mathsf{dom}(\xi)$, and likewise for $\sigma \cdot x : VM$. We can now present the semantics of expressions. Let $\Gamma;\Delta \vdash e : \alpha$ and assume that (ξ,σ) is a $\Gamma;\Delta$-model, we define $[\![e]\!]_{(\xi,\sigma)}$ by the following inductive clauses. $[\![\mathsf{c}]\!]_{(\xi,\sigma)} \stackrel{def}{=} \mathsf{c}$, $[\![\mathsf{op}(\vec{e})]\!]_{(\xi,\sigma)} \stackrel{def}{=} \mathsf{op}([\![\vec{e}]\!]_{(\xi,\sigma)})$, $[\![x]\!]_{(\xi,\sigma)} \stackrel{def}{=} (\xi \cup \sigma)(x)$. The satisfaction relation for formulae has the following shape. Let $\Gamma;\Delta \vdash A$ and assume that (ξ,σ) is a $\Gamma;\Delta$-model. We define $(\xi,\sigma) \models A$ as usual with two extensions.

- $(\xi,\sigma) \models e = e'$ iff $[\![e]\!]_{(\xi,\sigma)} \simeq [\![e']\!]_{(\xi,\sigma)}$.
- $(\xi,\sigma) \models \neg A$ iff $(\xi,\sigma) \not\models A$.

- $(\xi,\sigma) \models A \wedge B$ iff $(\xi,\sigma) \models A$ and $(\xi,\sigma) \models B$.
- $(\xi,\sigma) \models \forall x^{\alpha}.A$ iff for all closed values V of type α: $(\xi \cdot x : V,\sigma) \models A$.
- $(\xi,\sigma) \models u \bullet e = x\{A\}$ iff $([\![u]\!]_{(\xi,\sigma)}[\![e]\!]_{(\xi,\sigma)}) \Downarrow V$ and $(\xi \cdot x : V,\sigma) \models A$.
- $(\xi,\sigma) \models u = \langle m \rangle\{A\}$ iff $[\![u]\!]_{(\xi,\sigma)} \Downarrow \langle M \rangle$, $M \Downarrow V$ and $(\xi,\sigma \cdot m : V) \models A$.

To define the semantics of judgements, we need to say what it means to apply a model (ξ,σ) to a program M, written $M(\xi,\sigma)$. That is defined as usual, e.g. $x(\xi,\sigma) \stackrel{def}{=} (\xi \cup \sigma)(x)$ and $(MN)(\xi,\sigma) \stackrel{def}{=} M(\xi,\sigma)N(\xi,\sigma)$.

The *satisfaction relation* $\models \{A\}\ M :_m \{B\}$ is given next. Let $\Gamma;\Delta;\alpha \vdash \{A\}\ M :_m \{B\}$. Then

$$\models \{A\}\ M :_m \{B\} \quad \text{iff} \quad \forall(\xi,\sigma)^{\Gamma;\Delta}.(\xi,\sigma) \models A \supset \exists V.(M(\xi,\sigma) \Downarrow V \wedge (\xi \cdot m : V,\sigma) \models B).$$

This is the standard notion of total correctness, adapted to the present logic.

Axioms. This section introduces the key axioms of the logic. All axioms of [9,10,11] remain valid. We add axioms for $x = \langle y \rangle\{A\}$. Tacitly, we assume typability of all axioms. Some key axioms are given in Figure 2, more precisely, the axioms are the universal closure of the formulae presented in Figure 2. The presentation of axioms uses the following abbreviation: $\mathsf{Ext}_{\mathsf{q}}(xy)$ stands for $\forall a.(x = \langle z \rangle\{z = a\} \equiv y = \langle z \rangle\{z = a\})$.

Axiom $(q1)$ says that if the quasi-quote denoted by x makes A true (assuming the program in that quasi-quote is denoted by y), and in the same way makes B true, then it also makes $A \wedge B$ true, and vice versa. Axiom $(q2)$ says that if the quasi-quote denoted by x contains a terminating program, denoted by y, and makes $\neg A$ true, then it cannot be the case that under the same conditions A holds. The reverse implication is false, because $\neg x = \langle y \rangle\{A\}$ is also true when x denotes a quasi-quote whose contained program is diverging. Next is $(q3)$: $x = \langle y \rangle\{A\}$ says in particular that x denotes a quasi-quote containing a terminating program, so $\neg x = \langle y \rangle\{B\}$ can only be true because B is false. Axioms $(q4, q5)$ let us move formulae and quantifiers in and out of code-evaluation formulae, as long as the anchor is not inappropriately affected. Axiom $(q6)$ enables us

$$\frac{}{\{A[x/m] \wedge x \Downarrow\}\ x :_m \{A\}}\ \textsc{Var} \qquad \frac{}{\{A[\mathsf{c}/m]\}\ \mathsf{c} :_m \{A\}}\ \textsc{Const} \qquad \frac{\{A^{\neg g}\}\ M :_u \{B\}}{\{A\}\ \mu g.M :_u \{B[u/g]\}}\ \textsc{Rec}$$

$$\frac{\{A^{\neg x} \wedge B\}\ M :_m \{C\}}{\{A\}\ \lambda x^{\alpha}.M :_u \{\forall x.(B \supset u \bullet x = m\{C\})\}}\ \textsc{Abs} \qquad \frac{\{A\}\ M :_m \{B\} \quad \{B\}\ N :_n \{C[m+n/u]\}}{\{A\}\ M+N :_u \{C\}}\ \textsc{Add}$$

$$\frac{\{A\}\ M :_m \{B\} \quad \{B[b_i/m]\}\ N_i :_u \{C\} \quad b_1 = \mathsf{t}, b_2 = \mathsf{f} \quad i = 1,2}{\{A\}\ \mathtt{if}\ M\ \mathtt{then}\ N_1\ \mathtt{else}\ N_2 :_u \{C\}}\ \textsc{If}$$

$$\frac{\{A\}\ M :_m \{B\} \quad \{B\}\ N :_n \{m \bullet n = u\{C\}\}}{\{A\}\ MN :_u \{C\}}\ \textsc{App} \qquad \frac{\{A\}\ M :_m \{B\}}{\{\mathsf{T}\}\ \langle M \rangle :_u \{A \supset u = \langle m \rangle\{B\}\}}\ \textsc{Quote}$$

$$\frac{\{A\}\ M :_m \{B^{\neg mx} \supset m = \langle x \rangle\{C^{\neg m}\}\} \quad \{B \supset C\}\ N :_u \{D^{\neg mx}\}}{\{A\}\ \mathtt{let}\ \langle x \rangle = M\ \mathtt{in}\ N :_u \{D\}}\ \textsc{Unquote}$$

Fig. 3. Key PCF_{DP} inference rules for total correctness

to weaken a code-evaluation formula. The code-extensionality axiom (ext_q) formalises what it means for two quasi-quotes to be equal: they must contain observationally indistinguishable code.

Rules. Key rules of inference can be found in Figure 3. We write $\vdash \{A\}\ M :_m \{B\}$ to indicate that $\{A\}\ M :_m \{B\}$ is derivable using these rules. Rules make use of capture-free syntactic substitutions $e[e'/x]$, $A[e/x]$ which is straightforward, except that in $e[e'/x]$ and $A[e/x]$, x must be non-modal. Structural rules like Hoare's rule of consequence, are unchanged from [9,10,11] and used without further comment. The rules in Figure 3 are standard and also unchanged from [9,10,11] with three significant exceptions, explained next.

[VAR] adds $x \Downarrow$, i.e. $\exists a.x = a$ in the precondition. By construction of our models,$x \Downarrow$ is trivially true if x is non-modal. If x is modal, the situation is different because x may denote a non-terminating program. In this case $x \Downarrow$ constrains x so that it really denotes a value, as is required in a total correctness logic.

[QUOTE] says that $\langle M \rangle$ always terminates (because the conclusion's precondition is simply T). Moreover, if u denotes the result of evaluating $\langle M \rangle$, i.e. $\langle M \rangle$ itself, then, assuming A holds (i.e., given the premise, if M terminates), u contains a terminating program, denoted m, making B true. Clearly, in a logic of total correctness, if M is not a terminating program, A will be equivalent to F, in which case, [QUOTE] does not make a non-trivial assertion about $\langle M \rangle$ beyond stating that $\langle M \rangle$ terminates.

[UNQUOTE] is similar to the usual rule for $\mathtt{let}\ x = M\ \mathtt{in}\ N$ which is easily derivable:

$$\frac{\{A\}\ M :_x \{B\} \quad \{B\}\ N :_u \{C\}}{\{A\}\ \mathtt{let}\ x = M\ \mathtt{in}\ N :_u \{C\}}\ \textsc{Let}$$

Rules for $\mathtt{let}\ \langle x \rangle = M\ \mathtt{in}\ N$ are more difficult because a quasi-quote always terminates, but the code it contains may not. Moreover, even if M evaluates to a quasi-quote containing a divergent program, the overall expression may still terminate, because N uses the destructed quasi-quote in a certain manner. Here is an example:

$$\mathtt{let}\ \langle x \rangle = \langle \Omega \rangle\ \mathtt{in}\ ((\lambda ab.a)\ 7\ (\lambda().x)).$$

[UNQUOTE] deals with this complication in the following way. Assume $\{A\}\ M :_m \{B \supset m = \langle x \rangle \{C\}\}$ holds. If M evaluates to a quasi-quote containing a divergent program, B would be equivalent to F. The rule uses $B \supset C$ in the right premise, where x is now a free variable, hence also constrained by C. If B is equivalent to F, the right precondition is T, i.e. contains no information, and M's termination behaviour cannot depend on x, i.e. N must use whatever x denotes in a way that makes the termination or otherwise of N independent of x. Apart from this complication, the rule is similar to [LET].

4 Reasoning Examples

We now put our logic to use by reasoning about some of the programs introduced in Section 2. The derivations use the abbreviations of Section 3 and omit many steps that

are not to do with meta-programming. Several reduction steps are justified by the following two standard structural rules omitted from Figure 3.

$$\frac{A \supset A' \quad \{A'\}\, M :_u \{B'\} \quad B' \supset B}{\{A\}\, M :_u \{B\}}\ \textsc{Conseq} \qquad \frac{\{A\}\, M :_m \{B \supset C\}}{\{A \wedge B\}\, M :_m \{C\}}\ \supset\text{-}\wedge$$

Example 1. We begin with the simple program $\{\mathsf{T}\}\ \langle 1+2\rangle :_m \{m = \langle 3\rangle\}$. The derivation is straightforward.

1	$\{\mathsf{T}\}\ 1+2 :_a \{a = 3\}$	
2	$\{\mathsf{T}\}\ \langle 1+2\rangle :_m \{\mathsf{T} \supset m = \langle a\rangle\{a = 3\}\}$	QUOTE, *1*
3	$\{\mathsf{T}\}\ \langle 1+2\rangle :_m \{m = \langle 3\rangle\}$	CONSEQ, *2*

Example 2. This example deals with the code of a non-terminating program. We derive $\{\mathsf{T}\}\ \langle\Omega\rangle :_m \{\mathsf{T}\}$. This is the strongest total correctness assertion about $\langle\Omega\rangle$. In the proof, we assume that $\{\mathsf{F}\}\ \Omega :_a \{\mathsf{T}\}$ is derivable, which is easy to show.

1	$\{\mathsf{F}\}\ \Omega :_a \{\mathsf{T}\}$	
2	$\{\mathsf{T}\}\ \langle\Omega\rangle :_m \{\mathsf{F} \supset m = \langle a\rangle\{\mathsf{T}\}\}$	QUOTE, *1*
3	$\{\mathsf{T}\}\ \langle\Omega\rangle :_m \{\mathsf{T}\}$	CONSEQ, *2*

Example 3. The third example destructs a quasi-quote and then injects the resulting program into another quasi-quote.

$$\{\mathsf{T}\}\ \texttt{let}\ \langle x\rangle = \langle 1+2\rangle\ \texttt{in}\ \langle x+3\rangle :_m \{m = \langle 6\rangle\}$$

We derive the assertion in small steps to demonstrate how to apply our logical rules.

1	$\{\mathsf{T}\}\ \langle 1+2\rangle :_m \{m = \langle 3\rangle\}$	*Ex. 1*
2	$\{(a = 3)[x/a] \wedge x \Downarrow\}\ x :_a \{a = 3\}$	VAR
3	$\{x = 3\}\ x :_a \{a = 3\}$	CONSEQ, *2*
4	$\{\mathsf{T}\}\ 3 :_b \{b = 3\}$	CONST, CONSEQ
5	$\{a = 3\}\ 3 :_b \{a = 3 \wedge b = 3\}$	INVAR, *4*
6	$\{a = 3\}\ 3 :_b \{(c = 6)[a+b/c]\}$	CONSEQ, *5*
7	$\{x = 3\}\ x+3 :_c \{c = 6\}$	ADD, *3, 6*
8	$\{\mathsf{T}\}\ \langle x+3\rangle :_u \{x = 3 \supset u = \langle c\rangle\{c = 6\}\}$	QUOTE, *7*
9	$\{x = 3\}\ \langle x+3\rangle :_u \{u = \langle c\rangle\{c = 6\}\}$	$\supset$-$\wedge$, *8*
10	$\{\mathsf{T}\}\ \langle 1+2\rangle :_m \{\mathsf{T} \supset m = \langle x\rangle\{x = 3\}\}$	CONSEQ, *1*
11	$\{\mathsf{T} \supset x = 3\}\ \langle x+3\rangle :_u \{u = \langle 6\rangle\}$	CONSEQ, *9*
12	$\{\mathsf{T}\}\ \texttt{let}\ \langle x\rangle = \langle 1+2\rangle\ \texttt{in}\ \langle x+3\rangle :_u \{u = \langle 6\rangle\}$	UNQUOTE, *10, 11*

Example 4. Now we show that destructing a quasi-quote containing a non-terminating program, and then not using that program still leads to a terminating program. This reflects the operational semantics in Section 2.

$$\{\mathsf{T}\}\ \mathtt{let}\ \langle x\rangle = \langle \Omega\rangle\ \mathtt{in}\ \langle 1+2\rangle :_m \{m = \langle 3\rangle\}$$

The derivation follows.

1	$\{\mathsf{T}\}\ \langle\Omega\rangle :_m \{\mathsf{T}\}$	*Ex. 2*
2	$\{\mathsf{T}\}\ \langle\Omega\rangle :_m \{\mathsf{F} \supset m = \langle a\rangle\{\mathsf{T}\}\}$	CONSEQ, *1*
3	$\{\mathsf{T}\}\ \langle 1+2\rangle :_m \{m = \langle 3\rangle\}$	*Ex. 1*
4	$\{\mathsf{F} \supset \mathsf{T}\}\ \langle 1+2\rangle :_m \{m = \langle 3\rangle\}$	CONSEQ, *3*
5	$\{\mathsf{T}\}\ \mathtt{let}\ \langle x\rangle = \langle\Omega\rangle\ \mathtt{in}\ \langle 1+2\rangle :_m \{m = \langle 3\rangle\}$	UNQUOTE, *2, 4*

The examples below make use of the following convenient forms of the recursion rule and [UNQUOTE]. Both are easily derived.

$$\frac{\{A^{\neg g n} \wedge \forall 0 \le i < n.B[i/n][g/u]\}\ \lambda x.M :_u \{B^{\neg g}\}}{\{A\}\ \mu g.\lambda x.M :_u \{\forall n \ge 0.B\}}\ \textsc{Rec'} \qquad \frac{\{A\}\ M :_m \{\mathsf{T}\} \quad \{\mathsf{T}\}\ N :_u \{B\}}{\{A\}\ \mathtt{let}\ \langle x\rangle = M\ \mathtt{in}\ N :_u \{B\}}\ \textsc{UQ}$$

Example 5. This example extract a non-terminating program from a quasi-quote, and injects it into a new quasi-quote. Our total-correctness logic cannot say anything non-trivial about the resulting quasi-quote (cf. Example 2):

$$\{\mathsf{T}\}\ \mathtt{let}\ \langle x\rangle = \langle\Omega\rangle\ \mathtt{in}\ \langle x\rangle :_u \{\mathsf{T}\}$$

The derivation is straightforward.

1	$\{\mathsf{T}\}\ \langle\Omega\rangle :_m \{\mathsf{T}\}$	*Ex. 2*
2	$\{\mathsf{F}[x/a] \wedge x \Downarrow\}\ x :_a \{\mathsf{F}\}$	VAR
3	$\{\mathsf{F}\}\ x :_a \{\mathsf{T}\}$	CONSEQ, *2*
4	$\{\mathsf{T}\}\ \langle x\rangle :_u \{\mathsf{F} \supset u = \langle a\rangle\{\mathsf{T}\}\}$	QUOTE, *3*
5	$\{\mathsf{T}\}\ \mathtt{let}\ \langle x\rangle = \langle\Omega\rangle\ \mathtt{in}\ \langle x\rangle :_u \{\mathsf{T}\}$	UQ, *1, 4*

Example 6. Now we reason about $\mathsf{lift}_{\mathsf{Int}}$ from Section 3. In the proof we assume that i, n range over non-negative integers. Let $A^u_n \stackrel{def}{=} u \bullet n = m\{m = \langle n\rangle\}$. We are going to establish the following assertion from Section 3: $\{\mathsf{T}\}\ \mathsf{lift}_{\mathsf{Int}} :_u \{\forall n.A^u_n\}$. We set $C \stackrel{def}{=} i \le n \wedge \forall j < n.A^g_j$, $D \stackrel{def}{=} i > 0 \wedge \forall r.(0 \le r < n \supset g \bullet r = m\{m = \langle r\rangle\})$ and $P \stackrel{def}{=} \mathtt{let}\ \langle x\rangle = g(i-1)\ \mathtt{in}\ \langle x+1\rangle$.

1	$\{C\}\ i \leq 0 :_b \{C \wedge (b = \mathsf{t} \equiv i \leq 0)\}$	
2	$\{\mathsf{T}\}\ \langle 0 \rangle :_m \{m = \langle 0 \rangle\}$	*Like Ex. 1*
3	$\{i = 0\}\ \langle 0 \rangle :_m \{m = \langle i \rangle\}$	INVAR, CONSEQ, *2*
4	$\{(C \wedge b = \mathsf{t} \equiv i \leq 0)[\mathsf{t}/b]\}\ \langle 0 \rangle :_m \{m = \langle i \rangle\}$	CONSEQ, *3*
5	$\{x = i - 1\}\ \langle x + 1 \rangle :_m \{m = \langle i \rangle\}$	*Like Ex. 3*
6	$\{\mathsf{T} \supset x = i - 1\}\ \langle x + 1 \rangle :_m \{m = \langle i \rangle\}$	CONSEQ, *5*
7	$\{(C \wedge b = \mathsf{t} \equiv i \leq 0)[\mathsf{f}/b]\}\ g :_s \{D\}$	VAR
8	$\{D\}\ i - 1 :_r \{g \bullet r = t\{t = \langle i - 1 \rangle\}\}$	
9	$\{(C \wedge b = \mathsf{t} \equiv i \leq 0)[\mathsf{f}/b]\}\ g(i - 1) :_t \{t = \langle i - 1 \rangle\}$	APP, *7, 8*
10	$\{(C \wedge b = \mathsf{t} \equiv i \leq 0)[\mathsf{f}/b]\}\ P :_m \{m = \langle i \rangle\}$	UNQUOTE, CONSEQ, *6, 9*
11	$\{C\}\ \mathtt{if}\ i \leq 0\ \mathtt{then}\ \langle 0 \rangle\ \mathtt{else}\ P :_m \{m = \langle i \rangle\}$	IF, *4, 10*
12	$\{\mathsf{T}\}\ \lambda i.\mathtt{if}\ i \leq 0\ \mathtt{then}\ \langle 0 \rangle\ \mathtt{else}\ P :_u \{\forall i.(C \supset A_i^u)\}$	ABS, *11*
13	$\{\forall j < n.A_j^g\}\ \lambda i.\mathtt{if}\ i \leq 0\ \mathtt{then}\ \langle 0 \rangle\ \mathtt{else}\ P :_u \{\forall i \leq n.A_i^u\}$	CONSEQ $\supset$-$\wedge$, *12*
14	$\{\mathsf{T}\}\ \mathsf{lift}_{\mathsf{Int}} :_u \{\forall n.\forall i \leq n.A_n^u\}$	REC', *13*
15	$\{\mathsf{T}\}\ \mathsf{lift}_{\mathsf{Int}} :_u \{\forall n.A_n^u\}$	CONSEQ, *14*

Example 7. We close this section by reasoning about the staged power function from Section 2. Assuming that i, j, k, n range over non-negative integers, we define $B_n^u \stackrel{def}{=} u \bullet n = m\{m = \langle y \rangle\{\forall j.y \bullet j = j^n\}\}$. In the derivation, we provide less detail than in previous proofs for readability.

1	$C \stackrel{def}{=} n \leq k \wedge \forall i < k.B_i^p \qquad D \stackrel{def}{=} C \wedge (b = \mathsf{t} \wedge n \leq 0)$	
2	$P \stackrel{def}{=} \mathtt{let}\ \langle q \rangle = p(n-1)\ \mathtt{in}\ \langle \lambda x.x \times (q\ x) \rangle$	
3	$\{C\}\ n \leq 0 :_b \{D\}$	
4	$\{D[\mathsf{t}/b]\}\ \langle \lambda x.1 \rangle :_m \{m = \langle y \rangle\{\forall j.y \bullet j = j^n\}\}$	*Like prev. examples*
5	$\{D[\mathsf{f}/b]\}\ p(n-1) :_r \{\mathsf{T} \supset r = \langle q \rangle\{\forall j.q \bullet j = j^{n-1}\}\}$	*Like Ex. 6*
6	$\{\mathsf{T} \supset \forall j.q \bullet j = j^{n-1}\}\ \langle \lambda x.x \times (q\ x) \rangle :_m \{m = \langle y \rangle\{\forall j.y \bullet j = j^n\}\}$	*Like Ex. 6*
7	$\{D[\mathsf{f}/b]\}\ P :_m \{m = \langle y \rangle\{\forall j.y \bullet j = j^n\}\}$	UNQUOTE, *5, 6*
8	$\{C\}\ \mathtt{if}\ n \leq 0\ \mathtt{then}\ \langle \lambda x.1 \rangle\ \mathtt{else}\ P :_m \{m = \langle y \rangle\{\forall j.y \bullet j = j^n\}\}$	IF, *7*
9	$\{\mathsf{T}\}\ \lambda n.\mathtt{if}\ n \leq 0\ \mathtt{then}\ \langle \lambda x.1 \rangle\ \mathtt{else}\ P :_u \{\forall n \leq k.((\forall i < k.B_i^p) \supset B_n^u)\}$	ABS, *8*
10	$\{\forall i < k.B_i^p\}\ \lambda n.\mathtt{if}\ n \leq 0\ \mathtt{then}\ \langle \lambda x.1 \rangle\ \mathtt{else}\ P :_u \{\forall n \leq k.B_n^u\}$	CONSEQ, *9*
11	$\{\mathsf{T}\}\ \mathsf{power} :_u \{\forall k.\forall n \leq k.B_n^u\}$	REC', *10*
12	$\{\mathsf{T}\}\ \mathsf{power} :_u \{\forall n.B_n^u\}$	CONSEQ, *11*

5 Completeness

This section answers three important metalogical questions about the logic for total correctness introduced in Section 3.

– Is the logic *relatively complete* in the sense of Cook [5]? This question asks if $\models \{A\}\ M :_m \{B\}$ implies $\vdash \{A\}\ M :_m \{B\}$ for all appropriate A, B. Relative completeness means that the logic can syntactically derive all semantically true assertions, and reasoning about programs does not need to concern itself with models. We can always rely on just syntactic rules to derive an assertion (assuming an oracle for Peano arithmetic).

$$\frac{x \text{ non-modal}}{\{\mathsf{T}\}\ x :_m \{x = m\}}\ \textsc{Var} \qquad \frac{x \text{ modal}}{\{x \Downarrow\}\ x :_m \{x = m\}}\ \textsc{Var}_m \qquad \frac{\{A\}\ M :_m \{B\}}{\{\mathsf{T}\}\ \langle M\rangle :_u \{A \supset u = \langle m\rangle\{B\}\}}\ \textsc{Quote}$$

$$\frac{\{A_1\}\ M :_m \{B_1\} \quad \{A_2\}\ N :_u \{B_2\}}{\begin{array}{c}\{A_1 \wedge ((\forall m x^{\square}.A_2) \vee \forall m.(B_1 \supset m = \langle x\rangle\{A_2\}))\} \\ \texttt{let}\ \langle x\rangle = M\ \texttt{in}\ N :_u \\ \{\exists m x^{\square}.((m = \langle\cdot\rangle \supset m = \langle x\rangle) \wedge B_1 \wedge B_2)\}\end{array}}\ \textsc{Uq}$$

Fig. 4. Key inference system for TCAPs, where $m = \langle\cdot\rangle$ is short for $m = \langle z\rangle\{\mathsf{T}\}$

– Is the logic *observationally complete* [10]? The second question investigates if the program logic makes the same distinctions as the observational congruence. In other words, does $M \simeq N$ hold iff for all suitably typed A, B: $\{A\}\ M :_m \{B\}$ iff $\{A\}\ N :_m \{B\}$? Observational completeness means that the operational semantics (given by the contextual congruence) and the axiomatic semantics given by logic cohere with each other.
– If a logic is observationally complete, we may ask: given a program M, can we find, by induction on the syntax of M, *characteristic formulae* A, B such that (1) $\models \{A\}\ M :_m \{B\}$ and (2) for all programs N: $M \simeq N$ iff $\models \{A\}\ N :_m \{B\}$? If characteristic formulae always exist, the semantics of each program can be obtained and expressed finitely in the logic, and we call the logic *descriptively complete* [10].

Following [3,10,21], we answer all questions in the affirmative.

Characteristic Formulae. Program logics reason about program properties denoted by pairs of formulae. But what are program properties? We cannot simply say program properties are subsets of programs, because there are uncountably many such subsets, yet only countably many pairs of formulae. To obtain a notion of program property that is appropriate for a logic of total correctness, we note that such logics cannot express that a program diverges. More generally, if $\models \{A\}\ M :_m \{B\}$ and $M \sqsubseteq N$ (where $\sqsubseteq$ is the contextual pre-congruence from Section 2), then also $\models \{A\}\ N :_m \{B\}$. Thus pairs A, B talk about *upwards-closed* sets of programs. A set S of programs is upwards-closed if $M \in S$ and $M \sqsubseteq N$ together imply that $N \in S$. It can be shown that each upwards

closed set of PCF_{DP}-terms has a unique least element up-to $\simeq$. Thus each upwards-closed set has a distinguished member, is its least element. Consequently a pair A, B is a characteristic assertion pair for M (at m) if M is the *least* program w.r.t. $\sqsubseteq$ such that $\models \{A\}\ M :_m \{B\}$, leading to the following key definition.

Definition. A pair (A, B) is a *total characteristic assertion pair*, or *TCAP*, *of M at u*, if the following conditions hold (in each clause we assume well-typedness).

1. (soundness) $\models \{A\}\ M :_u \{B\}$.
2. (MTC, minimal terminating condition) For all models (ξ, σ), $M(\xi, \sigma) \Downarrow$ if and only if $(\xi, \sigma) \models A$.
3. (closure) If $\models \{E\}\ N :_u \{B\}$ and $E \supset A$, then for all (ξ, σ): $(\xi, \sigma) \models E$ implies $M(\xi, \sigma) \sqsubseteq N(\xi, \sigma)$.

A TCAP of M denotes a set of programs whose minimum element is M, and in that sense characterises that behaviour uniquely up to $\sqsubseteq$.

Descriptive Completeness. The main tool in answering the three questions posed above is the inference system for TCAPs, of which the key rules are given in Figure 4. The remaining rules are unchanged from [10]. We write $\vdash^{\text{tcap}} \{A\}\ M :_u \{B\}$ to indicate that $\{A\}\ M :_u \{B\}$ is derivable in that new inference system. It is obvious that TCAPs can be derived mechanically from programs – no invariants for recursion have to be supplied manually.

The premise of [UNQUOTE] in Figure 4 uses modal quantification. This is the only use of the modal quantifier. The semantics is: $(\xi, \sigma) \models \forall x^{\Box\alpha}.A$ iff for all closed programs M of type α: $(\xi, \sigma \cdot x : M) \models A$. Syntactic reasoning with modal quantifiers needs a few straightforward quantifier axioms beyond those of first-order logic and those of Figure 2, for example $\neg\forall x^{\Box}.x \Downarrow$, and $\neg\forall x^{\Box}.\neg x \Downarrow$. An interesting open question is whether modal quantification can be avoided altogether in constructing TCAPs.

Theorem 1.

1. *(descriptive completeness for total correctness)* Assume $\Gamma; \Delta \vdash M : \alpha$. Then $\vdash^{\text{tcap}} \{A\}\ M :_u \{B\}$ implies (A, B) is a TCAP of M at u.
2. (observational completeness) $M \simeq N$ if and only if, for each A and B, we have $\models \{A\}\ M :_u \{B\}$ iff $\models \{A\}\ N :_u \{B\}$.
3. (relative completeness) Let B be upward-closed at u. Then $\models \{A\}\ M :_u \{B\}$ implies $\vdash \{A\}\ M :_u \{B\}$.

6 Conclusion

We have proposed the first program logic for a meta-programming language, and established key metalogical properties like completeness and the correspondence between axiomatic and operational semantics. We are not aware of previous work on program logics for meta-programming. Instead, typing systems for statically enforcing program properties have been investigated. We discuss the two systems with the most expressive typing systems, Ωmega [15] and Concoqtion [7]. Both use indexed typed to achieve expressivity. Ωmega is a CBV variant of Haskell with generalised algebraic datatypes (GADTs) and an extensible kind system. In Ωmega, GADTs can

express easily datatypes representing object-programs, whose meta-level types encode the object-level types of the programs represented. Tagless interpreters can directly be expressed and typed for these object programs. Ωmega is expressive enough to encode the MetaML typing system together with a MetaML interpreter in a type-safe manner. Concoqtion is an extension of MetaOCaml and uses the term language of the theorem prover Coq to define index types, specify index operations, represent the properties of indexes and construct proofs. Basing indices on Coq terms opens all mathematical insight available as Coq libraries to use in typing meta-programs. Types in both languages are not as expressive with respect to properties of *meta-programs themselves*, as our logics, which capture exactly the observable properties. Nevertheless, program logic and type-theory are not mutually exclusive; on the contrary, reconciling both in the context of meta-programming is an important open problem.

The construction of our logics as extensions of well-understood logics for PCF indicates that logical treatment of meta-programming is mostly orthogonal to that of other language features. Hence [18] is an interesting target for generalising the techniques proposed here because it forms the basis of MetaOCaml, the most widely studied meta-programming language in the MetaML tradition. PCF_{DP} and [18] are similar as meta-programming languages with the exception that the latter's typing system is substantially more permissive: even limited forms evaluation of *open* code is possible. We believe that a logical account of meta-programming with open code is a key challenge in bringing program logics to realistic meta-programming languages. A different challenge is to add state to PCF_{DP} and extend the corresponding logics. We expect the logical treatment of state given in [2,21] to extend smoothly to a meta-programming setting. The main issue is the question what typing system to use to type stateful meta-programming: the system used in MetaOCaml, based on [18], is unsound in the presence of state due to a form of scope extrusion. This problem is mitigated in MetaOCaml with dynamic type-checking. As an alternative to dynamic typing, the Java-like meta-programming language Mint [20] simply prohibits the *sharing* of state between different meta-programming stages, resulting in a statically sound typing system. We believe that both suggestions can be made to coexist with modern logics for higher-order state [2,21], in the case of [20] easily so.

The relationship between PCF_{DP} and PCF, its non-meta-programming core, is also worth investigating. Pfenning and Davies proposed an embedding $\ulcorner\cdot\urcorner$ from PCF_{DP} into PCF, whose main clauses are given next.

$$\ulcorner\langle\alpha\rangle\urcorner \stackrel{def}{=} \mathsf{Unit} \to \ulcorner\alpha\urcorner$$
$$\ulcorner\langle M\rangle\urcorner \stackrel{def}{=} \lambda().\ulcorner M\urcorner$$
$$\ulcorner\mathtt{let}\ \langle x\rangle = M\ \mathtt{in}\ N\urcorner \stackrel{def}{=} \mathtt{let}\ x = \ulcorner M\urcorner\ \mathtt{in}\ \ulcorner N\urcorner[x()/x]$$

We believe that this embedding is fully-abstract, but proving full abstraction is non-trivial because translated PCF_{DP}-term have PCF-inhabitants which are not translations of PCF_{DP}-terms (e.g. $\lambda x^{\mathsf{Int}\to\mathsf{Int}}.\lambda().x$). A full abstraction proof might be useful in constructing a logically fully abstract embedding of the logic presented here into the simpler logic for PCF from [9,11]. A logical full abstraction result [13] is an important step towards integrating logics for meta-programming with logics for the produced meta-programs.

In the light of this encoding one may ask why meta-programming languages are relevant at all: why not simply work with a non-meta-programming language and encodings?

- We believe that nice (i.e. fully abstract and compositional) encodings might exist for simple meta-programming languages like PCF_{DP} because PCF_{DP} lives at the low end of meta-programming expressivity. For even moderately more expressive languages like [18] no fully abstract encodings into simple λ-calculi are known.
- A second reason is to do with efficiency, one of the key reasons for using meta-programming: encodings are unlikely to be as efficient as proper meta-programming.
- Finally, programs written using powerful meta-programming primitives are more readable and hence more easily evolvable than equivalent programs written using encodings.

References

1. Berger, M.: Program Logics for Sequential Higher-Order Control. In: Arbab, F., Sirjani, M. (eds.) Fundamentals of Software Engineering. LNCS, vol. 5961, pp. 194–211. Springer, Heidelberg (2010)
2. Berger, M., Honda, K., Yoshida, N.: A Logical Analysis of Aliasing in Imperative Higher-Order Functions. J. Funct. Program. 17(4-5), 473–546 (2007)
3. Berger, M., Honda, K., Yoshida, N.: Completeness and logical full abstraction in modal logics for typed mobile processes. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part II. LNCS, vol. 5126, pp. 99–111. Springer, Heidelberg (2008)
4. Berger, M., Tratt, L.: Program Logics for Homogeneous Metaprogramming. Long version of the present paper (to appear)
5. Cook, S.A.: Soundness and completeness of an axiom system for program verification. SIAM J. Comput. 7(1), 70–90 (1978)
6. Davies, R., Pfenning, F.: A modal analysis of staged computation. J. ACM 48(3), 555–604 (2001)
7. Fogarty, S., Pašalić, E., Siek, J., Taha, W.: Concoqtion: Indexed Types Now! In: Proc. PEPM, pp. 112–121 (2007)
8. Gunter, C.A.: Semantics of Programming Languages. MIT Press, Cambridge (1995)
9. Honda, K.: From Process Logic to Program Logic. In: ICFP 2004, pp. 163–174. ACM Press, New York (2004)
10. Honda, K., Berger, M., Yoshida, N.: Descriptive and Relative Completeness of Logics for Higher-Order Functions. In: Bugliesi, M., Preneel, B., Sassone, V., Wegener, I. (eds.) ICALP 2006. LNCS, vol. 4052, pp. 360–371. Springer, Heidelberg (2006)
11. Honda, K., Yoshida, N.: A compositional logic for polymorphic higher-order functions. In: Proc. PPDP, pp. 191–202 (2004)
12. Honda, K., Yoshida, N., Berger, M.: An Observationally Complete Program Logic for Imperative Higher-Order Functions. In: LICS 2005, pp. 270–279 (2005)
13. Longley, J., Plotkin, G.: Logical Full Abstraction and PCF. In: Tbilisi Symposium on Logic, Language and Information, CSLI (1998)
14. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: Proc. LICS 2002, pp. 55–74 (2002)
15. Sheard, T., Linger, N.: Programming in Ωmega. In: Proc. Central European Functional Programming School, pp. 158–227 (2007)

16. Sheard, T., Peyton Jones, S.: Template meta-programming for Haskell. In: Proc. Haskell Workshop, pp. 1–16 (2002)
17. Taha, W.: Multi-Stage Programming: Its Theory and Applications. PhD thesis, Oregon Graduate Institute of Science and Technology (1993)
18. Taha, W., Nielsen, M.F.: Environment classifiers. In: Proc. POPL, pp. 26–37 (2003)
19. Tratt, L.: Compile-time meta-programming in a dynamically typed OO language. In: Proc. DLS, pp. 49–64 (October 2005)
20. Westbrook, E., Ricken, M., Inoue, J., Yao, Y., Abdelatif, T., Taha, W.: Mint: Java multi-stage programming using weak separability. In: Proc. PLDI (2010) (to appear)
21. Yoshida, N., Honda, K., Berger, M.: Logical reasoning for higher-order functions with local state. In: Seidl, H. (ed.) FOSSACS 2007. LNCS, vol. 4423, pp. 361–377. Springer, Heidelberg (2007)

Appendix

A Typing Rules for Expressions, Formulae and Assertions

The key typing rules for expressions, formulae and judgements are summarised in Figure 5 above.

$$\frac{(x,\alpha) \in \Gamma \cup \Delta}{\Gamma;\Delta \vdash x : \alpha} \qquad \frac{\Gamma;\Delta \vdash u : \alpha \to \beta \quad \Gamma;\Delta \vdash e : \alpha \quad \Gamma, m : \beta;\Delta \vdash A}{\Gamma;\Delta \vdash u \bullet e = m\{A\}} \qquad \frac{\Gamma;\Delta \vdash e : \alpha \quad \Gamma;\Delta \vdash e' : \alpha}{\Gamma;\Delta \vdash e = e'}$$

$$\frac{\Gamma;\Delta \vdash A \quad \Gamma;\Delta \vdash B}{\Gamma;\Delta \vdash A \wedge B} \qquad \frac{\Gamma, x : \alpha;\Delta \vdash A}{\Gamma;\Delta \vdash \forall x^{\alpha}.A} \qquad \frac{\Gamma;\Delta, x : \alpha \vdash A}{\Gamma;\Delta \vdash \forall x^{\Box\alpha}.A} \qquad \frac{\Gamma;\Delta \vdash u : \langle\alpha\rangle \quad \Gamma;\Delta, m : \alpha \vdash A}{\Gamma;\Delta \vdash u = \langle m \rangle\{A\}}$$

$$\frac{\Gamma;\Delta \vdash A}{\Gamma;\Delta \vdash \neg A} \qquad \frac{\Gamma;\Delta \vdash A \quad m \notin \mathsf{dom}(\Gamma) \cup \mathsf{dom}(\Delta) \quad \Gamma;\Delta \vdash M : \alpha \quad \Gamma, m : \alpha;\Delta \vdash B}{\Gamma;\Delta;\alpha \vdash \{A\}\, M :_m \{B\}}$$

Fig. 5. Typing rules for expressions, formulae and judgements. Rules for constants and first-order operations omitted.

Verifying Pointer and String Analyses with Region Type Systems

Lennart Beringer[1], Robert Grabowski[2], and Martin Hofmann[2]

[1] Department of Computer Science, Princeton University,
35 Olden Street, Princeton 08540, New Jersey
eberinge@cs.princeton.edu
[2] Institut für Informatik, Ludwig-Maximilians-Universität,
Oettingenstrasse 67, D-80538 München, Germany
{robert.grabowski,martin.hofmann}@ifi.lmu.de

Abstract. Pointer analysis statically approximates the heap pointer structure during a program execution in order to track heap objects or to establish alias relations between references, and usually contributes to other analyses or code optimizations. In recent years, a number of algorithms have been presented that provide an efficient, scalable, and yet precise pointer analysis. However, it is unclear how the results of these algorithms compare to each other semantically.

In this paper, we present a general region type system for a Java-like language and give a formal soundness proof. The system is subsequently specialized to obtain a platform for embedding the results of various existing context-sensitive pointer analysis algorithms, thereby equipping the computed relations with a common interpretation and verification. We illustrate our system by outlining an extension to a string value analysis that builds on pointer information.

1 Introduction

Pointer (or points-to) analysis is a static program analysis technique that determines an over-approximation of possible points-to relations that occur during the execution of a program. More precisely, it chooses an abstraction of the pointers and references, and computes which pointers may possibly point to which data. A conservative approximation of this structure is also an alias analysis, as the computed points-to relation directly includes the information which pointers may point to the same object. A pointer analysis is often used for compiler optimizations, but may also serve as the basis for other analyses, such as the computation of possible string values in order to prevent string-based security holes.

There exists a large number of pointer analysis algorithms [1,2,3] for different languages. Each algorithm faces the trade-off between precision and efficiency of the analysis: it should choose the right abstractions in order to produce as much useful information as possible while at the same time being able to process large code bases in a reasonable time. These analyses have different techniques,

E.M. Clarke and A. Voronkov (Eds.): LPAR-16, LNAI 6355, pp. 82–102, 2010.

implementations, and complexities. Especially BDD-based algorithms [4,5] have been shown to be very efficient and precise at the same time.

While several of these analyses also consider soundness, it appears that there does not yet exist a uniformly agreed-upon formal framework that encompasses the interpretations of at least a substantial subset of the analyses. We argue that such a unifying treatment is important, for theoretical as well as pragmatic and practical reasons. First, it is a basis for fair comparisons regarding the precision, flexibility or expressivity of different analyses, the theoretical complexity of the associated algorithms, and their experimental evaluation on common benchmarks. Second, once the analyses agree on the formal property they guarantee, we can safely replace one analysis by another in a compiler or verification tool. Third, a uniform guarantee provides the basis for the formal verification of security-relevant properties that rely on pointer analysis results, as is required in proof-carrying code scenarios.

The first purpose of the present paper is to provide such a framework for Java-like languages, given by a hierarchy of region-based type systems for a language in the style of Featherweight Java [6]. Uniformity (i.e. semantic agreement) is guaranteed by equipping the bottom-most layer in the hierarchy with a formal interpretation and soundness result, and by deriving the higher levels by specializing this bottom-most layer to move towards calculi representing concrete analyses.

Second, we demonstrate that a number of existing pointer analyses for object-oriented programs are based on abstraction disciplines that arise as specializations of a generic parametrized refinement of our base-level type system. We focus on disciplines that specify the abstraction of references and execution points [7], and therefore enable different forms of field-sensitive and context-sensitive analyses. For example, objects may be abstracted to their allocation sites and their class, and execution points may be abstracted by receiver-object or call-site stack contexts.

As one moves higher in the hierarchy, different choices for these abstractions regarding expressivity arise, corresponding to the above abstraction disciplines, and decidability issues become more prominent, in particular the question of algorithmic type checking. Our third contribution consists of showing how the parametrized type system can be reformulated algorithmically to yield a type checking algorithm. Thanks to the hierarchical structure of our framework, we thus immediately obtain algorithms for automatically validating the correctness of the results of concrete analyses, as long as the results have been interpreted in the framework by instantiating its parameters. As we only consider the results, our verification is also independent of implementation details of concrete analysis algorithms.

Finally, we apply our framework to the analysis of string values, in order to lay the foundations for eliminating SQL injections and cross-site scripting attacks. We extend the language to a simple string model, and outline how data flow analyses for possible string values that build on pointer analyses [8,9] can be verified with the correspondingly extended type system.

Related work. We include concepts of context-sensitivity to distinguish different analyses of the same method implementation. In particular, k-CFA [10] is a higher-order control flow analysis where contexts are call stack abstractions of finite length k. The k-CFA mainly addresses control flows in functional languages, where functions are first-class values. It requires a combination of value (data flow) analysis and control flow analysis to approximate the possible lambda abstractions that an expression may evaluate to. The similar dynamic dispatch problem for methods in object-oriented languages is easier to solve, as the possible implementation targets of a method invocation can be retrieved from the class information. k-CFA has been extended with polyvariant types for functions [11], such that different types can be used for the function at different application sites. In our type system, we borrow this concept of polyvariance.

Hardekopf and Lin [12] transform variables that are not address-taken into SSA form, such that running a flow-insensitive analysis on these converted variables has the effect of running a flow-sensitive analysis on the original variables. Our framework assumes SSA-transformed code input, presented in the form of a functional program. Identifying opportunities for SSA transformation, which is a central concern of [12], can thus be seen as a preprocessing phase for our framework to apply.

This paper uses "regions" in the sense of Lucassen and Gifford [13], i.e. as representations of disjoint sets of memory locations. Equivalently, regions thus partition or color the memory. In literature, this disjointness is used to show that certain memory manipulations do not influence other parts of a program, in order to e.g. show semantic equivalences [14], to enable a safe garbage collection [15], or to infer properties for single objects by tracking unique objects in a region [16,17]. In the pointer analysis setting presented here, regions are simply seen as abstract memory locations that summarize one or more concrete locations, thereby helping to discover may-alias relations. We do not consider uniqueness or must-alias information, and do not aim to justify garbage collection.

Paddle [18] and Alias Analysis Library [19] are implementation frameworks that embed concrete pointer analyzers by factoring out different parameters, as is done here. However, the works do not aim at a formal soundness result. Indeed, they embed the *algorithms* into a common *implementation* framework, while our type system embeds the *results* of such algorithms into a common *semantic* framework.

Synopsis. The next section introduces the FJEU language and its semantics. We introduce the base-level region type system and the soundness proof in section 3. In section 4, we specialize the type system to a parametrized version, such that abstraction principles found in pointer analyses can be modeled explicitly as instantiations of the parameters, and outline a type-checking algorithm. Finally, section 5 extends the system, such that results of a string analysis based on pointer analysis can also be verified.

2 Featherweight Java with Updates

We examine programs of the language FJEU [20], a simplified formal model of the sequential fragment of Java that is relevant for pointer analysis. FJEU extends Featherweight Java (FJ) [6] with attribute updates, such that programs may have side effects on a heap. Object constructors do not take arguments, but initialize all fields with *null*, as they can be updated later. Also, the language adds `let` constructs and conditionals to FJ. For a small example, please refer to the full version of this paper [21], which shows a small list copy program in Java and a corresponding implementation in FJEU.

2.1 Preliminaries

We write $\mathcal{P}(X)$ for the set of all subsets of X. The notations $A \to B$ and $A \rightharpoonup B$ stand for the set of total and partial functions from A to B, respectively. We write $[x \mapsto v]$ for the partial function that maps x to v and is else undefined. The function $f[x \mapsto v]$ is equal to f, except that it returns v for x. Both function notations may be used in an indexed fashion, e.g. $f[x_i \mapsto v_i]_{\{1,\ldots,n\}}$, to define multiple values. In typing rules, we sometimes write $x : v$ and $f, x : v$ corresponding to the above notation. Finally, we write $\overline{a}$ for a sequence of entities a.

2.2 Syntax

The following table summarizes the (infinite) abstract identifier sets in the language, the meta-variables we use to range over them, and the syntax of FJEU expressions:

$$\begin{array}{rrrr} \text{variables: } x, y \in \mathcal{X} & & \text{classes: } C, D, E \in \mathcal{C} & \\ \text{fields: } f \in \mathcal{F} & & \text{methods: } m \in \mathcal{M} & \end{array}$$

$$\begin{aligned} \mathcal{E} \ni e ::= {} & \texttt{null} \mid x \mid \texttt{new}\ C \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid x.f \mid x.f := y \mid x.m(\overline{y}) \mid \\ & \texttt{if}\ x\ \texttt{instanceof}\ E\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{if}\ x = y\ \texttt{then}\ e\ \texttt{else}\ e \end{aligned}$$

To keep our calculus minimal and focused on pointer alias analysis, we omit primitive data types such as integers or booleans. However, such data types and their operations can easily be added to the language. We also omit type casts found in FJ, as they do not provide new insights to pointer analysis. In order to simplify the proofs, we require programs to be in let normal form. The somewhat unusual conditional constructs for dynamic class tests and value equality are included to have reasonable if-then-else expressions in the language while avoiding the introduction of booleans.

An FJEU program is defined by the following relations and functions:

$$\begin{array}{rr} \text{subclass relation:} & \prec \in \mathcal{P}(\mathcal{C} \times \mathcal{C}) \\ \text{field list:} & \mathit{fields} \in \mathcal{C} \to \mathcal{P}(\mathcal{F}) \\ \text{method list:} & \mathit{methods} \in \mathcal{C} \to \mathcal{P}(\mathcal{M}) \\ \text{method table:} & \mathit{mtable} \in \mathcal{C} \times \mathcal{M} \rightharpoonup \mathcal{E} \\ \text{FJEU program:} & P = (\prec, \mathit{fields}, \mathit{methods}, \mathit{mtable}) \end{array}$$

FJEU is a language with nominal subtyping: $D \prec C$ means D is an immediate subclass of C. The relation is well-formed if it is a tree successor relation; multiple inheritance is not allowed. We write $\preceq$ for the reflexive and transitive hull of $\prec$. The functions *fields* and *methods* describe for each class C which fields and method objects of that class have. The functions are well-formed if for all classes C and D such that $D \preceq C$, $\mathit{fields}(C) \subseteq \mathit{fields}(D)$ and $\mathit{methods}(C) \subseteq \mathit{methods}(D)$, i.e. classes inherit fields and methods from their superclasses. A method table *mtable* gives for each class and each method identifier its implementation, i.e. the FJEU expression that forms the method's body. To simplify the presentation, we assume that formal argument variables in the body of a method m are named x_1^m, x_2^m, etc., abbreviated to $\overline{x^m}$, besides the implicit and reserved variable this. All free variables of an implementation of m must be from the set $\{\mathit{this}, x_1^m, x_2^m, \ldots\}$. A method table is well-formed if $\mathit{mtable}(C, m)$ is defined whenever $m \in \mathit{methods}(C)$. In other words, all methods declared by *methods* must be implemented, though the implementation may be overridden in subclasses for the same number of formal parameters. In the following, we assume a fixed FJEU program P whose components are all well-formed.

2.3 Semantics

A state consists of a store (variable environment or stack) and a heap (memory). Stores map variables to values, while heaps map locations to objects. An object consists of a class identifier and a valuation of its fields. The only kind of values in FJEU are locations and *null* references.

$$\begin{array}{rlrl} \text{locations:} & l \in \mathcal{L} & \text{stores:} & s \in \mathcal{X} \rightharpoonup \mathcal{V} \\ \text{values:} & v \in \mathcal{V} = \mathcal{L} \cup \{\mathit{null}\} & \text{heaps:} & h, k \in \mathcal{L} \rightharpoonup \mathcal{O} \\ \text{objects:} & \mathcal{O} = \mathcal{C} \times (\mathcal{F} \rightharpoonup \mathcal{V}) & & \end{array}$$

The semantics of FJEU is defined as a standard big-step relation $(s, h) \vdash e \Downarrow v, h'$, which means that an FJEU expression e evaluates in store s and heap h to the value v and modifies the heap to h'. Due to reasons of limited space, figure 1 only shows the defining rules for some of the syntactic forms. A premise involving a partial function, like $s(x) = l$, implies the side condition $x \in \mathit{dom}(s)$.

2.4 Class Tables

A class table $\mathfrak{C}_0 = (A_0, M_0)$ models FJ's standard type system, where types are simply classes. The *field typing* $A_0 : (\mathcal{C} \times \mathcal{F}) \rightharpoonup \mathcal{C}$ assigns to each class C and each field $f \in \mathit{fields}(C)$ the class of the field. The field class is required to be invariant with respect to subclasses of C. The *method typing* $M_0 : (\mathcal{C} \times \mathcal{M}) \rightharpoonup \overline{\mathcal{C}} \times \mathcal{C}$ assigns to each class C and each method $m \in \mathit{methods}(C)$ a *method type*, which specifies the classes of the formal argument variables and of the result value. It is required to be contravariant in the argument classes and covariant in the result class with respect to subclasses of C.

$$\frac{l \notin dom(h) \quad F = [f \mapsto null]_{f \in fields(C)}}{(s,h) \vdash \texttt{new}\ C \Downarrow l, h[l \mapsto (C,F)]} \qquad \frac{(s,h) \vdash e_1 \Downarrow v_1, h_1 \quad (s[x \mapsto v_1], h_1) \vdash e_2 \Downarrow v_2, h_2}{(s,h) \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \Downarrow v_2, h_2}$$

$$\frac{s(x) = l \qquad h(l) = (C,F) \quad h' = h[l \mapsto (C, F[f \mapsto s(y)])]}{(s,h) \vdash x.f := y \Downarrow s(y), h'} \qquad \frac{\begin{array}{c} s(x) = l \qquad h(l) = (C, \_) \qquad |\overline{x^m}| = |\overline{y}| = n \\ s' = [this \mapsto s(x)] \cup [x_i^m \mapsto s(y_i)]_{i \in \{1,\ldots,n\}} \\ (s',h) \vdash mtable(C,m) \Downarrow v, h' \end{array}}{(s,h) \vdash x.m(\overline{y}) \Downarrow v, h'}$$

Fig. 1. Operational semantics of FJEU (extract)

3 Region Type System

In this section we define the base region type system, which serves as a main unifying calculus for pointer analysis and is given an interpretation and soundness proof. We assume an infinite set $\mathcal{R}$ of *regions* r, which are abstract memory locations. Each region stands for zero or more concrete locations. Different regions represent disjoint sets of concrete locations, hence they partition or *color* the memory. Two pointers to different regions can therefore never alias.

3.1 Refined Types and Subtyping

The region type system is a refinement of the plain type system: we equip classes with (possibly infinite) subsets from $\mathcal{R}$. For example, a location l is typed with the *refined type* $C_{\{r,s\}}$ if it points to an object of class C (or a subclass of C), and if l is abstracted to either r or s, but no other region. The *null* value can be given any type, while the type of locations must have a non-empty region set. The following table summarizes the definitions:

$$\begin{array}{rc} \text{Regions:} & r, s, t \in \mathcal{R} \\ \text{Region sets:} & R, S, T \in \mathcal{P}(\mathcal{R}) \\ \text{Refined types:} & \sigma, \tau \in \mathcal{T} = \mathcal{C} \times \mathcal{P}(\mathcal{R}) \end{array}$$

In the following, we use the notation C_R instead of (C, R) for types. Though the region identifier s is already used for variable stores, the difference should be clear from the context. Since region sets are an over-approximation of the possible locations where an object resides, we can easily define a subtyping relation $<:$ based on set inclusion:

$$C_R <: D_S \iff R \subseteq S \ \wedge\ C \preceq D$$

We also extend subtyping to method types:

$$\begin{array}{r} \overline{\sigma} <: \overline{\tau} \iff |\overline{\sigma}| = |\overline{\tau}| \ \wedge\ \forall i \in 1, \ldots, |\overline{\sigma}|.\ \sigma_i <: \tau_i \\ (\overline{\sigma}, \tau) <: (\overline{\sigma'}, \tau') \iff \overline{\sigma'} <: \overline{\sigma} \ \wedge\ \tau <: \tau' \end{array}$$

3.2 Annotated Class Tables

We extend (plain) class tables $\mathfrak{C}_0$ to *annotated class tables* $\mathfrak{C} = (A^{get}, A^{set}, M)$.

- The *annotated field typings* $A^{get}, A^{set} : (\mathcal{C} \times \mathcal{R} \times \mathcal{F}) \rightharpoonup \mathcal{T}$ assign to each class C, region r and field $f \in \mathit{fields}(C)$ the refined type of the field for all objects of class C in region r. The field type is split into a covariant get-type A^{get} for the data read from the field, and a contravariant set-type A^{set} that is needed for data to be written to the field. This technique improves precision and is borrowed from Hofmann and Jost [20]. More formally, annotated field typings are *well-formed* if for all classes C, subclasses $D \preceq C$, regions r and fields $f \in \mathit{fields}(C)$,
  - $A^{set}(C, r, f) <: A^{get}(C, r, f)$, and
  - $A^{get}(D, r, f) <: A^{get}(C, r, f)$ and $A^{set}(C, r, f) <: A^{set}(D, r, f)$.

  Also, the class component of $A^{get}(C, r, f)$ and $A^{set}(C, r, f)$ must be $A_0(C, f)$, i.e. invariant.
- The *annotated method typing* $M : (\mathcal{C} \times \mathcal{R} \times \mathcal{M}) \rightharpoonup \mathcal{P}(\overline{\mathcal{T}} \times \mathcal{T})$ assigns to each class C, region r and method $m \in \mathit{methods}(C)$ an unbounded number of refined method types for objects of class C in region r, enabling infinite polymorphic method types. This makes it possible to use a different type at different invocation sites (program points) of the same method. Even more importantly, the same invocation site can be checked in different type derivations with different method types. For every *well-formed* annotated method type, there must be an improved method type in each subclass: for all classes C, subclasses $D \preceq C$, regions r, and methods $m \in \mathit{methods}(C)$, we require
  - $\forall(\overline{\sigma}, \tau) \in M(C, r, m).\ \exists(\overline{\sigma'}, \tau') \in M(D, r, m).\ (\overline{\sigma}', \tau') <: (\overline{\sigma}, \tau)$.

  Again, the class components of the refined types $M(C, r, m)$ have to match the classes of the underlying unannotated method type $M_0(C, m)$.

In the following, we assume a fixed annotated class table $\mathfrak{C}$ with well-formed field and method typings.

3.3 Region Type System

The type system (see figure 2) derives judgements $\Gamma \vdash e : \tau$, meaning FJEU expression e has type τ with respect to a *variable context* (store typing) $\Gamma : \mathcal{X} \rightharpoonup \mathcal{T}$ that maps variables to types.

The rule T-Sub is used to obtain weaker types for the expression. The rules T-Let, T-Var, and T-IfInst are standard. The rule T-IfEq exploits the fact that the two variables must point to the same object (or be *null*) in the `then` branch, therefore the intersection of the region sets can be assumed. In T-Null, the *null* value may have any type (any class and any region set). In the rule T-New, we may choose any region r, which is an abstract location that includes (possibly among others) the concrete location of the object allocated by this expression.

$$\text{T-Sub}\ \frac{\Gamma \vdash e : \sigma \qquad \sigma <: \tau}{\Gamma \vdash e : \tau} \qquad \text{T-Let}\ \frac{\Gamma \vdash e_1 : \sigma \qquad \Gamma, x : \sigma \vdash e_2 : \tau}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau}$$

$$\text{T-Var}\ \frac{}{\Gamma, x : \tau \vdash x : \tau} \qquad \text{T-Null}\ \frac{}{\Gamma \vdash \texttt{null} : \tau}$$

$$\text{T-IfInst}\ \frac{x \in dom(\Gamma) \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{if}\ x\ \texttt{instanceof}\ E\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau}$$

$$\text{T-IfEq}\ \frac{\Gamma, x : C_{R \cap S}, y : D_{R \cap S} \vdash e_1 : \tau \qquad \Gamma, x : C_R, y : D_S \vdash e_2 : \tau}{\Gamma, x : C_R, y : D_S \vdash \texttt{if}\ x = y\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau}$$

$$\text{T-New}\ \frac{}{\Gamma \vdash \texttt{new}\ C : C_{\{r\}}}$$

$$\text{T-Invoke}\ \frac{\forall r \in R.\ \exists (\overline{\sigma'}, \tau') \in M(C, r, m).\ (\overline{\sigma'}, \tau') <: (\overline{\sigma}, \tau)}{\Gamma, x : C_R, \overline{y} : \overline{\sigma} \vdash x.m(\overline{y}) : \tau}$$

$$\text{T-GetF}\ \frac{\forall r \in R.\ A^{get}(C, r, f) <: \tau}{\Gamma, x : C_R \vdash x.f : \tau} \qquad \text{T-SetF}\ \frac{\forall r \in R.\ \tau <: A^{set}(C, r, f)}{\Gamma, x : C_R, y : \tau \vdash x.f := y : \tau}$$

Fig. 2. Region type system

When a field is read (T-GetF), we look up the type of the field in the A^{get} table. As the variable x may point to a number of regions, we need to ensure that τ is an upper bound of the get-types of f over all $r \in R$. In contrast, when a field is written (T-SetF), the written value must have a subtype of the types allowed for that field by the A^{set} table with respect to each possible region $r \in R$. Finally, the rule T-Invoke requires that for all regions $r \in R$ where the receiver object x may reside, there must exist a method typing that is suitable for the argument and result types.

An FJEU program $P = (\prec, \mathit{fields}, \mathit{methods}, \mathit{mtable})$ is *well-typed* if for all classes C, regions r, methods m and method types $(\overline{\sigma}, \tau)$ such that $(\overline{\sigma}, \tau) \in M(C, r, m)$, the following judgement is derivable:

$$[\mathit{this} \mapsto C] \cup [x_i^m \mapsto \sigma_i]_{i \in \{1, \ldots, |\overline{x^m}|\}} \vdash \mathit{mtable}(C, m) : \tau$$

The polymorphic method types make the region type system very expressive in terms of possible analyses of a given program. Each method may have many types, each corresponding to a derivation of the respective typing judgment. In the different derivations, different regions may be chosen for new objects, and different types may be chosen for called methods. This flexibility provides the basis for our embedding of external pointer analyses. Moreover, since there may be infinitely many method types for each method, this system is equivalent to one which allows infinite unfoldings of method calls.

3.4 Interpretation

We now give a formal interpretation of the typing judgement in form of a soundness theorem. Afterwards, we continue by restricting the expressivity of the system and reformulating the rules in order to move towards an actual type checking algorithm. The underlying idea is that we only have to prove soundness once for the general system; for later systems, it suffices to show that they are special cases of the general system, such that the soundness theorem applies to these systems as well.

A *heap typing* $\Sigma, \Pi : \mathcal{L} \rightharpoonup (\mathcal{C} \times \mathcal{R})$ assigns to heap locations a static class (an upper bound of the actual class found at that location) and a region. Heap typings, a standard practice in type systems for languages with dynamic memory allocations [22], separate the well-typedness definitions of locations in stores and objects from the actual heap, thereby avoiding the need for a co-inductive definition for well-typed heaps in the presence of cyclic structures. Heap typings map locations to very specific types, namely those where the region set is a singleton. A heap typing thus partitions a heap into (disjoint) regions.

We define a typing judgment for values $\Sigma \vdash v : \tau$, which means that according to heap typing Σ, the value v may be typed with τ. In particular, the information in $\Sigma(l)$ specifies the type of l. Also, the typing judgment of locations is lifted to stores and variable contexts.

$$\frac{}{\Sigma \vdash \texttt{null} : \tau} \qquad \frac{\Sigma(l) = (C, r)}{\Sigma \vdash l : C_{\{r\}}} \qquad \frac{\Sigma \vdash v : \sigma \qquad \sigma <: \tau}{\Sigma \vdash v : \tau}$$

$$\Sigma \vdash s : \Gamma \iff \forall x \in dom(\Gamma).\ \Sigma \vdash s(x) : \Gamma(x)$$

A heap h is *well-typed* with respect to a heap typing Σ and implicitly a field typing A^{get}, written $h \models \Sigma$, if the type for all locations given by Σ are actually "valid" with respect to the classes of the objects, and if the field values are well-typed with respect to A^{get} and Σ:

$$h \models \Sigma \iff \forall l \in dom(\Sigma).\ l \in dom(h) \ \wedge\ \Sigma \models h(l) : \Sigma(l)$$

where

$$\Sigma \models (C, F) : (D, r) \iff C \preceq D \ \wedge\ dom(F) = \mathit{fields}(C) \ \wedge \\ \forall f \in \mathit{fields}(C).\ \Sigma \vdash F(f) : A^{get}(C, r, f)$$

As the memory locations are determined at runtime, the heap typings cannot be derived statically. Instead, our interpretation of the typing judgement $\Gamma \vdash e : \tau$ states that whenever a well-typed program is executed on a heap that is well-typed with respect to some typing Σ, then the final heap after the execution is well-typed with respect to some possibly larger heap typing Π. The typing Π may be larger to account for new objects that may have been allocated during execution, but the type of locations that already existed in Σ may not change. More formally, a heap typing Π *extends* a heap typing Σ, written $\Pi \sqsupseteq \Sigma$, if $dom(\Sigma) \subseteq dom(\Pi)$ and $\forall l \in dom(\Sigma).\ \Sigma(l) = \Pi(l)$.

Theorem 1 (Soundness Theorem). *Fix a well-typed program P. For all $\Sigma, \Gamma, \tau, s, h, e, v, k$ with*

$$\Gamma \vdash e : \tau \quad \textit{and} \quad \Sigma \vdash s : \Gamma \quad \textit{and} \quad (s,h) \vdash e \Downarrow v, k \quad \textit{and} \quad h \models \Sigma$$

there exists some $\Pi \sqsupseteq \Sigma$ such that

$$\Pi \vdash v : \tau \quad \textit{and} \quad k \models \Pi.$$

Proof. By induction over the derivations of the operational semantics and the typing judgement. Details can be found on the authors' homepage [21]. We have also developed a formalization in Isabelle/HOL.

4 Parametrized Region Type System

Our next goal is to use the type system for the automatic algorithmic verification of results of external analyses. In this section, we focus on the first step in this direction by showing how to *interpret* given analysis results in the type system. We then outline how to implement an algorithmic type checking algorithm for the automatic verification.

The interpretation of given results requires the reformulation of the above type system to explicitly model abstraction principles that are fundamental for a number of different pointer analysis techniques [7]. We focus on two general classes of abstractions: the abstraction of the call graph using contexts, and the abstraction of objects on the heap. The region type system is equipped with parameters that can be instantiated to specific abstraction principles. We show that the parametrized version of the type system arises as a specialization of the general region type system.

In the following, the notion of program points is made explicit by annotating expressions with *expression labels* $i \in \mathcal{I}$: we write $[e]^i$ for FJEU expressions, where e is defined as before. An FJEU program is well-formed if each expression label i appears at most once in it. In the following, we only consider well-formed programs, and simply write e instead $[e]^i$ if the expression label i is not important.

4.1 Abstraction Principles

Context-insensitive pointer analyses examine each method exactly once. Their result provides for each method a single pointer structure specification which is used for every call of that method. *Context-sensitive* algorithms improve precision: each method may be analyzed multiple times under different *contexts*, so that different specifications can be used for different calls to the same method. A context-sensitive algorithm settles on a specific (finite) set of contexts, and produces for each method one pointer structure specification per context. Pointer

structure specifications correspond to our method types. We therefore model the concept of contexts by introducing a finite abstract set of contexts $\mathcal{Z}$, and by parametrizing the method typing function M to associate one method type per context in $\mathcal{Z}$.

The choice of contexts and specifications for each method call depends on the analysis in question. For example, call-site sensitive algorithms [23] process each method once for each program point where the method is called. Receiver-object sensitive analyses [24] differentiate pointer structure specifications of a method according to the abstraction of the invoking object. More powerful analyses use *call stacks* as contexts to differentiate method calls. For example, a method m may be analyzed for each possible call-site stack that may occur at the time when m is called. Similarly, receiver-object stacks can be used. In other words, one considers the *call graph* of the program. A method m is then represented by a node in this graph, and a context corresponds to a possible path that leads to the node. As there may be infinitely many paths in recursive programs, the number of paths needs to be restricted by some mechanism. A common way is to consider only the last k entries on the stack (k-CFA [10]), or to collapse each strongly connected component into a node, thereby eliminating recursive cycles from the set of possible paths [1,25]. Following this observation, we employ a general *context transfer function* ϕ which represents the edges in the abstract call graph. The function selects a context for the callee based on the caller's context, the class of the receiver object, its region, the method name, and the call site.

Another abstraction principle is the object abstraction, i.e. the abstract location assigned to allocated objects. This corresponds to our concepts of regions. As pointer analysis algorithms differentiate only finitely many abstract objects, we can restrict the set of regions $\mathcal{R}$ to a finite size.

A common abstraction is to distinguish objects according to their allocation site and/or their class. More precise analyses also take into account the context under which allocation takes place. For example, in object-sensitive analysis by Milanova et al. [24], objects are distinguished by their own allocation site and the allocation site of the `this` object. Objects may also be distinguished by the call site of the invoked method, a technique called *heap specialization* [26]. We model these concepts by an *object abstraction function* ψ that assigns the region for the new object, given the allocation site and the current method context. Our system is by design class-sensitive, as the class information is part of the type.

Figure 3 summarizes the four parameters of our system: contexts, context transfer function, regions, and object abstraction function. Also, it shows how to instantiate the parameters to obtain various standard abstraction principles.

The parametrized method typing $\hat{M} : (\mathcal{C} \times \mathcal{R} \times \mathcal{Z} \times \mathcal{M}) \rightharpoonup \overline{\mathcal{T}} \times \mathcal{T}$ replaces the annotated polymorphic method typing M in the annotated class table. It is well-formed if for all classes C and subclasses $D \preceq C$, regions $r \in \mathcal{R}$, methods $m \in \mathit{methods}(C)$, and contexts $z \in \mathcal{Z}$, it holds $\hat{M}(D, r, z, m) <: \hat{M}(C, r, z, m)$.

$$\begin{aligned}
\text{Regions (finite):}\quad & r, s, t \in \mathcal{R} \\
\text{Contexts (finite):}\quad & z \in \mathcal{Z} \\
\text{Context transfer function:}\quad & \phi \in \mathcal{Z} \times \mathcal{C} \times \mathcal{R} \times \mathcal{M} \times \mathcal{I} \to \mathcal{Z} \\
\text{Object abstraction function:}\quad & \psi \in \mathcal{Z} \times \mathcal{I} \to \mathcal{R}
\end{aligned}$$

set of regions	*object abstraction function*	*principle*
$\mathcal{R} = \mathcal{I}$	$\psi(z, i) = i$	allocation site abstraction
$\mathcal{R} = \mathcal{Z} = \mathcal{I} \times \mathcal{I}$	$\psi((i_1, i_2), i_0) = (i_0, i_1)$	object-sensitive allocation site abstraction
$\mathcal{R} = \mathcal{Z} = \mathcal{I}$	$\psi(i_c, i) = i_c$	heap specialization

set of contexts	*context transfer function*	*principle*
$\mathcal{Z} = \{z_0\}$	$\phi(z, C, r, m, i) = z_0$	context-insensitivity
$\mathcal{Z} = \mathcal{R}$	$\phi(z, C, r, m, i) = r$	object-sensitive 1-CFA
$\mathcal{Z} = \bigcup_{n \in \{1,\ldots,k\}} \mathcal{M}^n$	$\phi(z, C, r, m, i) = (m :: z)\|_k$	method identifier k-CFA
$\mathcal{Z} = \{\overline{i} \in \sigma(\mathcal{I}') \mid \mathcal{I}' \subseteq \mathcal{I}\}$	$\phi(z_1, C, r, m, i) = z_2$ s.th. $IE_c(z_1, i, z_2, m)$	CFA with eliminated recursive cycles

where

- $\sigma(X)$ is the set of all permutations of X
- $L|_k$ is the truncation of list L after the first k elements
- IE_c is the call graph relation of [25] where recursive cycles have been replaced by single nodes

Fig. 3. The four type system parameters, and possible instantiations to common abstraction principles

4.2 The Parametrized Type System

The parametrized type system extends the typing judgment of the general region type system by a context component z. With the exception of the following two rules, all rules remain as presented in section 3 (with the addition of the context z in each judgement):

$$\text{TP-New}\ \frac{r = \psi(z, i)}{\Gamma\ ;\ z\ \vdash\ [\texttt{new}\ C]^i\ :\ C_{\{r\}}}$$

$$\text{TP-Invoke}\ \frac{\forall r \in R.\ \hat{M}(C, r, \phi(z, C, r, m, i), m) <: (\overline{\sigma}, \tau)}{\Gamma, x : C_R, \overline{y} : \overline{\sigma}\ ;\ z\ \vdash\ [x.m(\overline{y})]^i\ :\ \tau}$$

While in the previous system any region r could be chosen for new objects, we have restricted this flexibility in the TP-New rule to the region specified by ψ. Moreover, instead of allowing arbitrarily many types per method, from which any type could be selected for invocations, we now have one type determined by the context z that is selected by the ϕ function in the TP-Invoke rule.

For a given parametrized method typing $\hat{M}$, we define a the corresponding polymorphic method typing $M(C,r,m) := \bigcup_{z\in\mathcal{Z}}\{\hat{M}(C,r,z,m)\}$. The rules of the parametrized system are derivable in the previous system: The only changed rules TP-New and TP-Invoke have more restrictive premises than their counterparts T-New and T-Invoke. Hence if $\Gamma\ ;\ z\ \vdash\ e\ :\ \tau$ can be derived from some $\hat{M}$ in the parametrized system, then $\Gamma\ \vdash\ e\ :\ \tau$ can be derived in the previous system with respect to the corresponding method typing M.

A method table is *well-typed* for $\hat{M}$ if for all classes C, contexts z, regions r, and methods m such that $\hat{M}(C,r,z,m) = (\overline{\sigma},\tau)$, the judgement $\Gamma\ ;\ z\ \vdash\ mtable(C,m)\ :\ \tau$ can be derived with $\Gamma = [this \mapsto C] \cup [x_i^m \mapsto \sigma_i]_{i\in\{1,\ldots,|\overline{x^m}|\}}$. It is easy to see that if a method table is well-typed with respect to $\hat{M}$ in the parametrized system, then it is also well-typed with respect to the corresponding method typing M in the general system. Therefore, the soundness theorem is applicable to the parametrized region type system.

4.3 Algorithmic Type Checking

The parametrized type system can be rewritten into a syntax directed form, from which one can directly read off an algorithm $A(\Gamma,e) = \tau$ that computes "from left to right" the type τ of an expression e based on a store typing Γ. For this, we eliminate the subtyping rule, and instead specify the most precise resulting type τ for each expression, similarly to the approach taken by Pierce [22]. The soundness proof shows that the internalisation of the subtyping rule is correct, i.e. that the judgements derived with the algorithmic type system can also be derived with the parametrized type system. To demonstrate the verification capabilities of the algorithmic type system, we have also developed a context-sensitive pointer analysis algorithm for FJEU, described declaratively as a set of recursive Datalog rules in the style of Whaley and Lam [25]. The full algorithmic type system, its soundness proof and the sample pointer analysis algorithm can be found on the authors' homepage [21].

5 String Analysis

String analysis is a dataflow analysis technique to determine possible string values (character sequences) that may occur during the execution of a program. Since strings appear as objects in Java, it is natural to implement a string analysis by building on pointer analysis: string objects are identified and tracked by the pointer analysis, while their possible values are determined by the string analysis.

We now equip the FJEU language with special string objects and operations to give a simplified formalization of Java's `String` class, and extend the region type system to enable the verification of pointer analyses of string objects. Afterwards, we show how to use the region information to interpret the results of a specific string analysis.

5.1 FJEU with Strings

The language FJEUS is an extension of FJEU with operations to create and concatenate strings. While the `String` class in Java is just another class in the Java class hierarchy, it is regarded as a separate type in FJEUS. This allows us to treat string objects differently: An object of class *String* in FJEUS is simply a string value (character sequence) on the heap. The meta-variable w ranges over character sequences $\mathcal{W}$, and W ranges over sets of string values. We rely on a given sequence concatenation function $+$.

$$\begin{array}{rr} \text{character sequences:} & w \in \mathcal{W} \\ \text{sets of character sequences:} & W \in \mathcal{P}(\mathcal{W}) \\ \text{extended expressions:} & \mathcal{E} \ni e ::= \ldots \mid \texttt{new String}(w) \mid x.concat(y) \end{array}$$

$$\begin{array}{rr} \text{character sequence concatentation:} & + \in \mathcal{W} \times \mathcal{W} \to \mathcal{W} \\ \text{heaps:} & h, k \in \mathcal{L} \rightharpoonup \mathcal{O} \cup \mathcal{W} \end{array}$$

The expression $\texttt{new String}(w)$ allocates a new string object with the character sequence w on the heap. The string operation $x.concat(y)$ has its own special semantics and is implemented with the $+$ operator. As we only model strings with non-mutable values in the language, a string concatenation always creates a new string object on the heap. The operational semantics is extended as follows:

$$\frac{l \notin dom(h)}{(s,h) \vdash \texttt{new String}(w) \Downarrow l, h[l \mapsto w]}$$

$$\frac{\begin{array}{ccccc} s(x) = l_1 & s(y) = l_2 & h(l_1) = w_1 & & h(l_2) = w_2 \\ & l \notin dom(h) & w = w_1 + w_2 & & \end{array}}{(s,h) \vdash x.concat(y) \Downarrow l, h[l \mapsto w]}$$

Note that this rather modest extension is intended to keep the formalization simple. Other extensions could include more string operations, or mutable string objects that model Java's `StringBuffer` class.

5.2 Pointer Analysis for String Objects

We extend the region type system from section 3 to accommodate the new string objects. We distinguish references to "proper" objects and to string objects: a type is either a class with a region set (C_R), or the special `String` class with a region set ($\texttt{String}_R$). The `String` class is independent from other classes in the class hierarchy.

$$\text{types: } \sigma, \tau \in \mathcal{T} = (\mathcal{C} \cup \{\texttt{String}\}) \times \mathcal{P}(\mathcal{R})$$

$$\frac{C \preceq D \qquad R \subseteq S}{C_R <: D_S} \qquad\qquad \frac{R \subseteq S}{\texttt{String}_R <: \texttt{String}_S}$$

Fields and method typings may now include the $\texttt{String}_R$ type. Also, all existing typing rules from section 3 remain unchanged. In particular, the *null* value may be assigned a `String` type. A field $x.f$ may only be accessed if x is a (non-`String`) class C; similarly for method calls $x.m$. These are the two additional typing rules for $\texttt{new String}(str)$ and $x.concat(y)$:

$$\frac{}{\Gamma \vdash \texttt{new String}(w) \ : \ \texttt{String}_{\{r\}}}$$

$$\frac{}{\Gamma, x : \texttt{String}_R, y : \texttt{String}_S \vdash \ x.concat(y) \ : \ \texttt{String}_{\{t\}}}$$

The heap typings $\Sigma, \Pi : \mathcal{L} \rightharpoonup (\mathcal{C} \cup \{\texttt{String}\}) \times \mathcal{R}$ may now also map locations to the `String` class. We extend the well-typed value relation $\Sigma \vdash v : \tau$ accordingly:

$$\frac{}{\Sigma \vdash \texttt{null} : \tau} \qquad \frac{\Sigma(l) = (C, r)}{\Sigma \vdash l : C_{\{r\}}} \qquad \frac{\Sigma(l) = (\texttt{String}, r)}{\Sigma \vdash l : \texttt{String}_{\{r\}}}$$

$$\frac{\Sigma \vdash v : \sigma \qquad \sigma <: \tau}{\Sigma \vdash v : \tau}$$

The definition of well-typed heaps additionally requires well-typed character sequences (last line):

$$\begin{aligned} h \models \Sigma &\iff \forall l \in dom(\Sigma).\ l \in dom(h) \ \wedge \ \Sigma \models h(l) : \Sigma(l) \\ \Sigma \models (C, F) : (D, r) &\iff C \preceq D \ \wedge \ \ldots \ \text{(as before)} \\ \Sigma \models w : (\texttt{String}, r) &\iff \mathsf{PROP}(w, r) \end{aligned}$$

In other words, the property that a heap h is well-typed with respect to Σ now includes the condition that for all locations l such that $\Sigma(l) = (\texttt{String}, r)$, $h(l)$ contains a string value w that satisfies a certain property PROP with respect to r. For the moment, assume PROP is simply *True*. The proof of the soundness theorem is extended in a straight-forward way for the extensions described above. Moreover, the type system can be parametrized in the same fashion as described in section 4: as both string operations create new objects, we use the ψ function to determine the region of these objects.

In the following subsection, we present an analysis that can help to prevent cross-site scripting attacks, and give a semantic formalization by instantiating the string property PROP.

5.3 String Analysis with Operation Contexts

In a typical cross-site scripting scenario, a user input is embedded into a string that is executed or interpreted. To prevent the injection of malicious code, one wants to track how a string is constructed, and ensure that its executable parts originate from trusted sources, like string literals in the program code.

We therefore add a string operation context Ω, which expresses the possible string operations that objects in specific regions may be the result of. The typing rules are extended with constraints on Ω.

$$\begin{array}{ll} \text{string operations:} & \mathcal{O} \ni \omega ::= \texttt{newstring}\ w\ r \mid \texttt{concat}\ r\ s\ t \mid \texttt{unknown}\ W\ r \\ \text{operation context:} & \Omega\ \in\ \mathcal{P}(\mathcal{O}) \end{array}$$

$$\frac{\texttt{newstring}\ w\ r \in \Omega}{\Gamma\ \vdash\ \texttt{new String}(w)\ :\ \texttt{String}_{\{r\}}}$$

$$\frac{\forall r \in R, s \in S.\ \texttt{concat}\ r\ s\ t \in \Omega}{\Gamma, x : \texttt{String}_R, y : \texttt{String}_S\ \vdash\ x.concat(y)\ :\ \texttt{String}_{\{t\}}}$$

Informally, an operation $\texttt{newstring}\ w\ r \in \Omega$ means that region r may include a string w. $\texttt{concat}\ r\ s\ t$ means that region t may include a string object that is obtained by concatenating some strings from regions r and s. The operation $\texttt{unknown}\ W\ r$ means that strings in region r may be from the set W. This is useful for external methods whose types are given to but not verified by the type system. Whenever more primitive string operations are added to the language, the set $\mathcal{O}$ may be extended accordingly.

Formally, we define a semantic interpretation $\Omega[\![r]\!]$ that gives the possible values for string objects in region r. It is defined as the smallest set satisfying the following conditions:

$$\begin{array}{rcl} \texttt{newstring}\ w\ r \in \Omega & \Rightarrow & w \in \Omega[\![r]\!] \\ \texttt{concat}\ r\ s\ t \in \Omega & \Rightarrow & \forall w_1 \in \Omega[\![r]\!], w_2 \in \Omega[\![s]\!].\ w_1 + w_2 \in \Omega[\![t]\!] \\ \texttt{unknown}\ W\ r \in \Omega & \Rightarrow & W \subseteq \Omega[\![r]\!] \end{array}$$

After instantiating the string property as $\mathsf{PROP}(w, r) \equiv w \in \Omega[\![r]\!]$, the relation $h \models \Sigma$ ensures that string values on the heap are indeed in the interpretation of the string operation context.

Apart from this extensional interpretation, the string operation context and the typing of external methods also contain intensional information about the origin and the possible constructions of strings in a specific region, which enables the verification of more complex string policies.

For example, consider the following string-manipulating program that relies on external functions *getUserInput()* to retrieve data from the user, *escapeHTML(s)* that quotes all HTML tags in string s and returns the result as a new string, and *output(s)* that outputs the string s.

```
let firstPart = new String("<sometag>")
 in let contents = getUserInput()
     in let escContents = escapeHTML(contents)
         in output(firstPart.concat(sanContents))
```

The security policy is that the output may only be the result of a concatenation of a string literal with a string that does not contain HTML tags. The

policy is expressed using the following types for external methods and the string operation context:

$$\begin{array}{rl} \mathit{getUserInput} : \texttt{unit} \longrightarrow \texttt{String}_{\{q\}} & \Omega = \{\ \texttt{newstring "<sometag>"}\ l, \\ \mathit{escapeHTML} : \texttt{String}_{\{q\}} \longrightarrow \texttt{String}_{\{s\}} & \quad \texttt{unknown}\ \mathcal{W}\ q,\ \ \texttt{unknown}\ \hat{\mathcal{W}}\ s, \\ \mathit{output} : \texttt{String}_{\{t\}} \longrightarrow \texttt{unit} & \quad \texttt{concat}\ s\ l\ t,\ \ \texttt{concat}\ l\ s\ t\ \ \} \end{array}$$

($\hat{\mathcal{W}}$ is the set of all strings that do not contain HTML tags, and `unit` is a unit type, which could be modeled in FJEUS as a null type like $\texttt{String}_{\emptyset}$.) The external function *getUserInput* returns strings with arbitrary values of the set $\mathcal{W}$ in region q ("questionable"), which are converted by *escapeHTML* into strings of region s ("sanitized"), which are assumed to not contain any HTML tags (set $\hat{\mathcal{W}}$). For the literal `firstPart`, the type checker can assign the region l ("literals"), as the literal value in Ω matches. The *output* function only accepts strings from region t ("trusted"), which must be, according to Ω, a concatenation of strings from region l and (HTML tag-free) region s. Therefore, the typability of the program proves that the security policy is indeed fulfilled. The example demonstrates that handling trusted sanitizing functions is actually a strength of type-based presentations: simply assign an appropriate type to a function if you believe the associated semantic property of the function.

The approach is related to the work by Christensen et al. [8] on the analysis of string values using context-free grammars with operation productions. The symbolic string operations in the context correspond to nodes in their annotated flow graph, and their semantics of the flow graph resembles our interpretation of $\Omega[\![r]\!]$. Similarly, Crégut and Alvarado [9] have presented an algorithm that tracks string objects with pointer analysis, and collects intensional information about the string operations applied to them. We thus expect that aspects of the results of these algorithms are verifiable in our system. Our approach is also related to taint analysis [27], as the region identifiers can convey information about the trustworthiness of strings, which is preserved throughout assignments and method invocations.

6 Discussion

We presented a framework for classifying alias analyses for Java-like languages, given by a hierarchy of region-based type systems. We demonstrated how existing disciplines arise as instantiations of our framework and may be given a uniform interpretation by embedding their results in a single type system. We also gave an algorithmic variant of the type system, thus enabling syntax-directed type-checking and hence validation of analyses results. Finally, we showed how our framework may be extended to string analyses. In the following, we briefly discuss specific design decisions, and outline future work.

To our knowledge, most existing pointer analyses express their results at a coarse-grain level of syntactic structure such as methods. In accordance with this, we employed a phrase-based formulation of type systems and interpreted

judgements with respect to a big-step evaluation semantics. An extension of the interpretation to include non-terminating executions appears possible, using techniques as in [28].

Our framework does not aim to be flow- or path-sensitive. We see these concepts as orthogonal to the central idea of our paper, namely interpreting context-sensitivity using polyvariant types. Nevertheless, we acknowledge the increasing relevance of flow and path sensitivity in recent work on pointer analysis. The T-IfEq rule illustrates a possible extension of the type system with path-sensitive capabilities: the information from the branching expression is used to refine the analysis for the "then" branch of the conditional. Moreover, subderivations of a judgement contain implicitly more fine-grained (non-)alias relationships applicable at intermediate program points, and include aspects of flow-sensitivity as any variable may be associated with different types in a derivation. Arguably, local alias assertions could be made more explicit by moving to a small-step operational regime and/or formulations of the type systems that are inspired by abstract interpretation and yield a global specification table with entries for all program points [29]. However, the use of evaluation-style judgements greatly simplifies soundness proofs at least at the level of methods, as recursive calls follow the type derivation structure.

The Doop framework [5] enables the definition of highly precise and efficient pointer analyses declaratively using Datalog rules. While the authors do not seem to aim at a fully formal correctness proof that interprets the Datalog rules and relations with respect to the semantics of the Java language, they take great care to separate the essential aspects of analysis techniques from implementation details of the algorithm. We intend to look at the ideas in their work in order to find ways to adapt our type system to more language features and pointer analyses.

In addition to type systems for pointer alias analysis, the literature contains the terminology "type-based alias analysis" [30]. The latter term appears to mean that datatype or static class information is used to *improve the precision* of an alias analysis, while region type systems directly *include* the points-to relations in the types. However, as our system extends the ordinary type system of Java, it arguably also encompasses type-based alias analyses.

Having been developed in order to embed static analysis results, it is not surprising that the type systems over-approximate their semantic guarantee. Thus, failure to validate the *legitimate* result of a specific analysis may be rooted in either an incorrect interpretation of the analysis into our framework or the fact that the analysis is more precise than the host type system. A particular direction in which our type system (and some analyses) might be generalized is shape analysis [31]. Another interesting recent development that our work does not accommodate is the analysis of programs that use reflection [27]; this would require a fundamental understanding of the semantic analysis of reflection.

Although we have only examined the *results* of pointer analysis algorithms, the algorithms can be seen as an external means of type inference. It seems promising to further investigate the *implementations* of these algorithms, and to recreate

their logic in the type system in order to obtain a parametric type system with a fully automatic (internal) type inference. Alternatively, the identification of abstraction principles could also propose a way to parametrize existing pointer analysis implementations.

Regarding the string analysis, we have concentrated on the previously-noted observation that the precision of the analysis benefits from the availability of (non-)aliasing information [9]. In principle, the benefits may be mutual. For example, the method call $x.concat(y)$ on a `String` in Java actually returns the reference x if y has a length of zero. If the length of y can be obtained from a string analysis, this information helps to improve the region set for the result type in the rule for concatenation. Mutual dependencies between aliasing and string analyses may thus be an interesting topic for future work.

Also, we have only outlined the use of a verified string analysis for security policies. In collaboration with SAP's Sophia-Antipolis-based research lab on security we plan to extend the type system with string operation contexts to verify the absence of cross-site scripting attacks in concrete scenarios.

Acknowledgements. We would like to thank Pierre Crégut for making us aware of the interplay between pointer and string analyses and providing us with several pointers to the literature, as well as Keqin Li (SAP research) for giving us insights to concrete cross-site scripting scenarios and the corresponding string policies. This work was supported by the Information Society Technologies programme of the European Commision, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project.

References

1. Emami, M., Ghiya, R., Hendren, L.J.: Context-sensitive interprocedural points-to analysis in the presence of function pointers. In: 1994 Conference on Programming Language Design and Implementation (PLDI 1994), pp. 242–256. ACM, New York (1994)
2. Andersen, L.O.: Program Analysis and Specialization for the C Programming Language. PhD thesis, DIKU, University of Copenhagen (1994)
3. Steensgaard, B.: Points-to analysis in almost linear time. In: 23rd Symposium on Principles of Programming Languages (POPL 1996), pp. 32–41. ACM, New York (1996)
4. Berndl, M., Lhoták, O., Qian, F., Hendren, L., Umanee, N.: Points-to analysis using BDDs. In: 2003 Conference on Programming Language Design and Implementation (PLDI 2003). ACM, New York (2003)
5. Bravenboer, M., Smaragdakis, Y.: Strictly declarative specification of sophisticated points-to analyses. In: 24th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2009), pp. 243–262. ACM, New York (2009)
6. Igarashi, A., Pierce, B., Wadler, P.: Featherweight Java: A minimal core calculus for Java and GJ. In: 1999 Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 1999). ACM, New York (1999)

7. Ryder, B.G.: Dimensions of precision in reference analysis of object-oriented programming languages. In: Hedin, G. (ed.) CC 2003. LNCS, vol. 2622, pp. 126–137. Springer, Heidelberg (2003)
8. Christensen, A.S., Møller, A., Schwartzbach, M.I.: Precise analysis of string expressions. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694. Springer, Heidelberg (2003)
9. Crégut, P., Alvarado, C.: Improving the Security of Downloadable Java Applications With Static Analysis. Electronic Notes in Theoretical Computer Science 141(1), 129–144 (2005)
10. Shivers, O.: Control-Flow Analysis of Higher-Order Languages, or Taming Lambda. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, Technical Report CMU-CS-91-145 (1991)
11. Banerjee, A., Jensen, T.: Modular control-flow analysis with rank 2 intersection types. Mathematical Structures in Computer Science 13(1), 87–124 (2003)
12. Hardekopf, B., Lin, C.: Semi-sparse flow-sensitive pointer analysis. SIGPLAN Not. 44(1), 226–238 (2009)
13. Lucassen, J.M., Gifford, D.K.: Polymorphic effect systems. In: 15th Symposium on Principles of Programming Languages (POPL 1988), pp. 47–57. ACM, New York (1988)
14. Benton, N., Kennedy, A., Beringer, L., Hofmann, M.: Relational semantics for effect-based program transformations with dynamic allocation. In: Principles and Practice of Decl. Prog. (PPDP 2007). ACM, New York (2007)
15. Tofte, M., Talpin, J.P.: Region-based memory management. Information and Computation 132(2), 109–176 (1997)
16. Crary, K., Walker, D., Morrisett, G.: Typed memory management in a calculus of capabilities. In: 26th Symposium on Principles of Programming Languages (POPL 1999), pp. 262–275. ACM, New York (1999)
17. Foster, J.S., Terauchi, T., Aiken, A.: Flow-sensitive type qualifiers. In: 2002 Conference on Programming Language Design and Implementation (PLDI 2002), pp. 1–12. ACM, New York (2002)
18. Lhoták, O.: Program Analysis using Binary Decision Diagrams. PhD thesis, McGill University (January 2006)
19. Lenherr, T.: Taxonomy and Applications of Alias Analysis. Master's thesis, ETH Zürich (2008)
20. Hofmann, M., Jost, S.: Type-based amortised heap-space analysis. In: Sestoft, P. (ed.) ESOP 2006. LNCS, vol. 3924, pp. 22–37. Springer, Heidelberg (2006)
21. Beringer, L., Grabowski, R., Hofmann, M.: Verifying Pointer and String Analyses with Region Type Systems: Soundness proofs and other material (2010), `http://www.tcs.ifi.lmu.de/ grabow/regions`
22. Pierce, B.C.: Types and Programming Languages. MIT Press, Cambridge (2002)
23. Sharir, M., Pnueli, A.: Two approaches to interprocedural data flow analysis. In: Muchnick, S.S., Jones, N.D. (eds.) Program Flow Analysis: Theory and Applications. Prentice Hall International, Englewood Cliffs (1981)
24. Milanova, A., Rountev, A., Ryder, B.G.: Parameterized object sensitivity for points-to analysis for Java. ACM Trans. Softw. Eng. Methodol. 14(1), 1–41 (2005)
25. Whaley, J., Lam, M.S.: Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. SIGPLAN Not. 39(6), 131–144 (2004)
26. Nystrom, E.M., Kim, H.S., Hwu, W.W.: Importance of heap specialization in pointer analysis. In: 5th Workshop on Program Analysis for Software Tools and Engineering (PASTE 2004). ACM, New York (2004)

27. Tripp, O., Pistoia, M., Fink, S.J., Sridharan, M., Weisman, O.: Taj: effective taint analysis of web applications. In: 2009 Conference on Programming Language Design and Implementation (PLDI 2009), pp. 87–97. ACM, New York (2009)
28. Beringer, L., Hofmann, M., Pavlova, M.: Certification Using the Mobius Base Logic. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2007. LNCS, vol. 5382, pp. 25–51. Springer, Heidelberg (2008)
29. Barthe, G., Pichardie, D., Rezk, T.: A certified lightweight non-interference Java bytecode verifier. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 125–140. Springer, Heidelberg (2007)
30. Diwan, A., McKinley, K.S., Moss, J.E.B.: Type-based alias analysis. In: 1998 Conference on Programming Language Design and Implementation (PLDI 1998), pp. 106–117. ACM, New York (1998)
31. Loncaric, S.: A survey of shape analysis techniques. Pattern Recognition 31, 983–1001 (1998)

ABC: Algebraic Bound Computation for Loops*

Régis Blanc[1], Thomas A. Henzinger[2], Thibaud Hottelier[3], and Laura Kovács[4]

[1] EPFL
[2] IST Austria
[3] UC Berkeley
[4] TU Vienna

Abstract. We present ABC, a software tool for automatically computing symbolic upper bounds on the number of iterations of nested program loops. The system combines static analysis of programs with symbolic summation techniques to derive loop invariant relations between program variables. Iteration bounds are obtained from the inferred invariants, by replacing variables with bounds on their greatest values. We have successfully applied ABC to a large number of examples. The derived symbolic bounds express non-trivial polynomial relations over loop variables. We also report on results to automatically infer symbolic expressions over harmonic numbers as upper bounds on loop iteration counts.

1 Introduction

Establishing tight upper bounds on the execution times of programs is both difficult and interesting, see e.g. [10,5,9,8]. We present ABC, a new software tool for automatically computing tight symbolic upper bounds on the number of iterations of nested program loops. The derived bounds express polynomial relations over loop variables. ABC is fully automatic, combines static analysis of programs with symbolic summation techniques, and requires no user-guidance in providing additional set of predicates, templates and assertions. ABC is also able to derive symbolic expressions over harmonic numbers as upper bounds on loop iteration counts, which, to the best of our knowledge, is not yet possible by other works.

In our approach to bound computation, we have identified a special class of nested loop programs, called the *ABC-loops* (Section 3.1). Further, we have built a *loop converter* to transform, whenever possible, arbitrary loops into their equivalent ABC-loop format (Section 3.2). Informally, an ABC-loop is a nested for-loop such that each loop from the nested loop contains exactly one iteration variable with only one condition and one (non-initializing) update on the iteration variable. For such loops, our method derives precise bounds on the number of loop iterations.

In our work, we assume that each program statement is annotated with the time units it needs to be executed. For simplicity, we assume that an iteration of an unnested loop takes one unit time, and all other instructions of the unnested loop need zero time.

The key steps of our approach to *bound computation* are as follows (Section 3.3). (i) First, we instrument the innermost loop body of an ABC-loop with a new variable

* This work was supported in part by the Swiss NSF. The fourth author is supported by an FWF Hertha Firnberg Research grant (T425-N23).

E.M. Clarke and A. Voronkov (Eds.): LPAR-16, LNAI 6355, pp. 103–118, 2010.

that increases at every iteration of the program. We denote this variable by z. Upper bounds on the value of z thus express upper bounds on the number of loop iterations. (ii) Next, the value of z is computed as a polynomial function over the nested loop's iteration variables. We call the relation between z and the loop's iteration variables the *z-relation*. To this end, for each loop of the ABC-loop, recurrence equations of z and the loop iteration variables are first constructed. Closed forms of variables are then derived using our *symbolic solver* which integrates special techniques from symbolic summation (Section 3.4). The derived closed forms express valid relations between z and the loop iteration variables, and thus the z-relations are loop invariant properties. (iii) Further, by replacing loop iteration variables by bounds on their greatest values in the computed z-relation, bounds on the value of z are obtained. These bounds give us tight symbolic upper bounds on the number of iterations of the program. Our method can be generalized for the timing analysis of loops whose iteration bounds involve harmonic expressions over the loop variables (Section 3.5).

Implementation. ABC was implemented in the the Scala programming language [18], contains altogether 5437 lines of Scala code, and is available at:

`http://mtc.epfl.ch/software-tools/ABC/`

Inputs to ABC are loops written in the Scala syntax. ABC first rewrites the input loop into an equivalent ABC-loop by using its *loop converter*, and then computes bounds on loop iteration counts using its *bound computer*. The *bound computer* relies on the *symbolic solver* in order to derive closed forms of symbolic sums and simplify mathematical expressions. The overall workflow of ABC is given in Figure 1.

Note that *ABC does not rely on an external computer algebra package for symbolic summation.*

Experiments. We successfully applied ABC on examples from [10,9], as well as on 90 nested loops extracted from the JAMA package [13] – see Section 4 and the mentioned URL[1]. Altogether, we ran ABC on 558 lines of JAMA. ABC computed precise upper bounds on iteration counts for all loops, and inferred the z-relation for 87 loops, all in less than one second on a machine with a 2.8 GHz Intel Core 2 Duo processor and 2GB of RAM. The 3 loops for which ABC was not able to derive the z-relation were actually sequences of loops.

We believe that similar experimental results as the ones resulting from JAMA could be obtained by running ABC on the Jampack library [20], or on various numerical packages of computer algebra packages such as Mathematica [22], Matlab [4], or Mathcad [2].

Related work. We only discuss some of the many methods that are related to ABC.

Paper [15] infers polynomial loop invariants among program variables by using polynomial invariant templates of bounded degree. Unlike [15], where no restrictions on the considered loops were made, we require no user guidance in providing invariant templates but automatically derive invariants (z-relations) for a restricted class of loops. Our method has thus advantage in automation, but it is restricted to ABC-loops.

[1] There are 167 loops in JAMA amongst which there are 90 nested for-loops. ABC successfully inferred the exact bound for all but three for-loops.

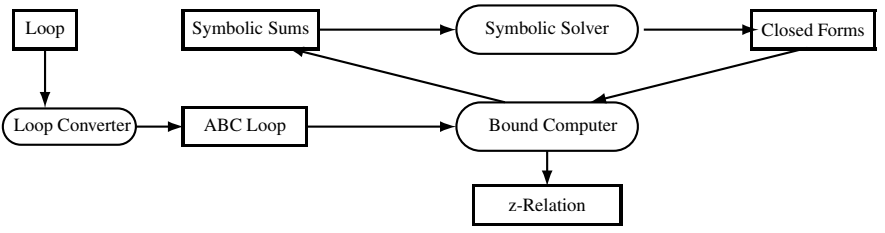


Fig. 1. The ABC tool

The approach presented in [11] infers invariants and bound assertions for loops with nested conditionals and assignments, where the assignments statements describe non-trivial recurrence relations over program variables (i.e. variable initializations are not allowed). To this end, loops are first represented by a collection of loop-free program paths, where each path corresponds to one conditional branch. Further, recurrence solving over variables is applied on each program path separately. Bounds on iteration counters can be finally inferred if the iteration counters are changed by each path in the same manner. Due to these restrictions, nested loops cannot be handled in [11]. Contrarily to [11], we infer bound assertions as z-relations for nested loops, but, unlike [11], our invariant assertions are only over loop iteration variables and not arbitrary program variables.

Paper [10] derives iteration bounds of nested loops by pattern matching simple recurrence equations. Contrarily to [10], we solve more general recurrence equations using the Gosper algorithm [6] and identities over harmonic numbers [7].

Solving recurrence relations is also the key ingredient in [1] for computing bounds. Unlike our method, evaluation trees for the unfoldings of the recurrence relations are first built in [1], and closed forms of recurrences are then derived from the maximum size of the trees. Contrarily to [1], we can handle more general recurrences by means of symbolic computation, but [1] has the advantage of solving non-deterministic recurrences that may result from the presence of guards in the loop body.

Symbolic upper bounds on iteration counts of multi-path loops are automatically derived in [8]. The approach deploys control-flow refinement methods to eliminate infeasible loop paths and rewrites multi-path loops into a collection of simpler loops for which bound assertions are inferred using abstract interpretation techniques [3]. The programs handled by [8] are more general than the ABC-loops. Unlike [8], we do not rely on abstract interpretation, and are able to infer harmonic expressions as upper bounds on loop iterations counts. Abstract interpretation is also used in [12,16] for automatically inferring upper and lower bounds on the number of execution steps of logic programs.

Paper [14] describes an automated approach for inferring linear upper bounds for functional programs, by solving constraints that are generated using linear programming. In our work we derive polynomial, and not just linear, upper bounds.

There has been a great deal of research on estimating the worst case execution time (WCET) of real-time systems, see e.g. [5,9,19]. Papers [5,9] automatically infer loop

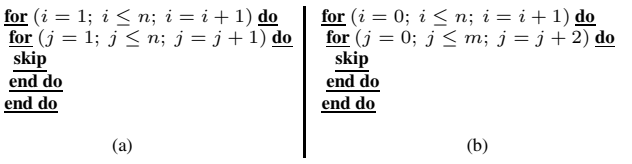


Fig. 2. Examples illustrating the power of ABC to (i) compute z-relations as loop invariants, and (ii) infer tight upper bounds on the number of iterations of loops

bounds only for simple loops; bounds for the iteration numbers of multi-path loops must be provided as user annotations. The aiT tool [5] determines the number of loop iterations by relying on a combination of interval-based abstract interpretation with pattern matching on typical loop patterns. The SWEET tool [9] determines upper bounds on loop iterations by unrolling loops dynamically and analyzing each loop iteration separately using abstract interpretation. In contrast, our method is fully automatic and path-insensitive, but it is restricted to ABC-loops. The TuBound tool [19] implements a constraint logic based approach for loop analysis to compute tight bounds for nested-loops. The constraints being solved in [19] are directly obtained from the loop conditions and express bounds on the loop iteration variables. Unlike [19], we infer loop bounds by computing closed forms of iteration variables.

2 Motivating Examples

We first give some examples illustrating what kind of iteration bounds ABC can automatically generate.

Consider Figure 2(a) taken from the JAMA library [13]. ABC first instruments the innermost loop of Figure 2(a) with a new variable z, initialized to 1, for counting the number of iterations of Figure 2(a). The thus obtained loop is presented in Figure 3(a). Further, by applying ABC on Figure 3(a), we derive the z-relation[2]:

$$z = (i-1)n + j$$

as an invariant property of the loop. By replacing i and j with bounds on their greatest values (i.e. n) in the z-relation, the number of iterations of Figure 2(a) is bounded by:

$$n^2.$$

Consider next Figure 2(b) with a non-unit increment, and its "instrumented" version in Figure 3(b). We obtain the z-relation:

$$z = 1 + \left\lfloor \frac{j}{2} \right\rfloor + i\left(\left\lfloor \frac{m}{2} \right\rfloor + 1\right),$$

[2] Actually, the loops of Figure 2 are first translated into their equivalent ABC-format, and then the z variable is introduced in their innermost loop body. For simplicity, in Figure 3 we present the "instrumentation" step directly on the loops of Figure 2 and not on their ABC-loop formats.

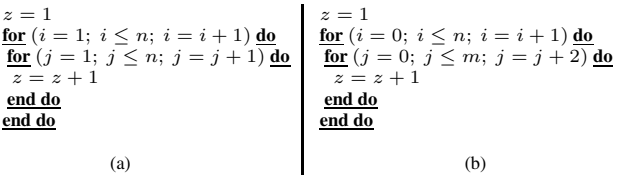


Fig. 3. Figure 2 instrumented by ABC

yielding:

$$1 + (1 + n)\left\lfloor \frac{m}{2} \right\rfloor + n$$

as a tight upper bound on loop iteration counts [3], where $\lfloor \frac{m}{2} \rfloor$ denotes the largest integer not greater than $\frac{m}{2}$.

In the sequel, we illustrate the main steps of ABC on Figure 2(b).

3 ABC: System Description

We have identified a special class of loops, called the *ABC-loops* (Section 3.1), and designed a loop converter for translating programs into their equivalent ABC-loop shape (Section 3.2). Algorithmic methods from symbolic summation, implemented in our symbolic solver (Section 3.4), are further deployed in ABC to automatically derive upper bounds on loop iterations of ABC-loops (Section 3.3).

3.1 ABC-Loops

We denote by $\mathbb{Z}$ the set of integer numbers, and by $\mathbb{Z}[x]$ the ring of polynomial functions in indeterminate x over $\mathbb{Z}$.

We consider programs of the following form:

$$\begin{array}{l}
\textbf{for}\ (i_1 = 1;\ i_1 \leq c;\ i_1 = i_1 + 1)\ \textbf{do} \\
\quad \textbf{for}\ (i_2 = 1;\ i_2 \leq f_1(i_1);\ i_2 = i_2 + 1)\ \textbf{do} \\
\qquad \ldots \\
\qquad \textbf{for}\ (i_d = 1;\ i_d \leq f_{d-1}(i_1, \ldots, i_{d-1});\ i_d = i_d + 1)\ \textbf{do} \\
\qquad\quad \textbf{skip} \\
\qquad \textbf{end do} \\
\qquad \ldots \\
\quad \textbf{end do} \\
\textbf{end do}
\end{array} \tag{1}$$

where $i_1, \ldots, i_d$ are pairwise disjoint *scalar variables* (called *loop iteration variables*) with values from $\mathbb{Z}$, c is an integer-valued symbolic constants, and $f_k \in \mathbb{Z}[i_1, \ldots, i_k]$ are polynomial functions ($k = 1, \ldots, d-1$).

[3] In our work we did not consider analyzing the relations between the smallest and greatest symbolic values of the loop iteration variables. It may however be the case that these symbolic values are such that the loops are never executed (e.g. $n < 0$).

Algorithm 1. Loop Converter

Input: For-loop F and $conversion\_list = \{\}$
Output: ABC-loop F' and $conversion\_list$
1: $\langle ovar,\ oincr\rangle$: = $\langle$outer_iteration_variable(F), outer_iteration_increment(F)$\rangle$
2: $\langle olbound,\ oubound\rangle$: = $\langle$outer_iteration_lowerbound(F), outer_iteration_upperbound(F)$\rangle$
3: $nvar$: = fresh_variable()
4: F_0 : = $loop\_body(F)\big[ovar \mapsto oincr \cdot \big(nvar + olbound - 1\big)\big]$
5: $conversion\_list$: =$conversion\_list \cup \{ovar = oincr \cdot \big(nvar + olbound - 1\big)\}$
6: **if** isloop(F_0) **then**
7: F' : = for-loop($nvar$, 1, $\lfloor \frac{oubound - olbound}{oincr} \rfloor + 1$, 1, Loop Converter($F_0$))
8: **else**
9: F' : = for-loop($nvar$, 1, $\lfloor \frac{oubound - olbound}{oincr} \rfloor + 1$, 1, F_0)
10: **end if**

In what follows, loops satisfying (1) will be called *ABC-loops*.

3.2 The Loop Converter

Converting loops into ABC-loops is done as presented in Algorithm 1. The algorithm (i) converts loops into equivalent ones such that the smallest values of the loop iteration variables are 1, and (ii) converts loops with arbitrary increments over the iteration variables into equivalent loops with increments of 1. The for-loop(v,e_1,e_2,e_3,body) notation used in Algorithm 1 is a short-hand notation for the loop:

$$\underline{\textbf{for}}\ (v = e_1; v \leq e_2; v = v + e_3)\ \underline{\textbf{do}}\ body\ \underline{\textbf{end do}}.$$

In more detail, Algorithm 1 takes as input a nested for-loop F and an empty list $conversion\_list$, and returns, whenever possible, an ABC-loop F' that is equivalent to F. The $conversion\_list$ is used to store the list of changes made by Algorithm 1 on the iteration variables of F.

Lines 4-9 of Algorithm 1 are required to convert F into an equivalent loop whose outermost loop has the following properties: it iterates over a new variable $nvar$ instead of the iteration variable $ovar$ of the outermost loop of F, where $nvar$ and $ovar$ are polynomially related; the smallest value of $nvar$ is 1 (instead of the smallest value $olbound$ of $ovar$); $nvar$ is increment by 1 (instead of the $oincr$ increment value of $ovar$); the greatest value of $nvar$ is given by the largest integer not greater than the rational expression $\frac{oubound - olbound}{oincr} + 1$, where $oubound$ is the greatest value of $ovar$. The appropriately modified[4] loop body F_0 of F is processed in the similar manner, yielding finally the ABC-loop F' that is equivalent to F.

Example 1. Consider Figure 2(b). By applying Algorithm 1, the loop iteration variables i_1 and j_1 are introduced with $i = i_1 - 1$ and $j = 2(j_1 - 1)$ (lines 3-5 of Algorithm 1). The smallest values of i_1 and j_1 are 1, their greatest values are respectively $n + 1$ and $\lfloor \frac{m}{2} \rfloor + 1$, and i_1 and j_1 are incremented by 1 (lines 6-9 of Algorithm 1). The ABC-loop format of Figure 2(b) is given in Figure 4(a).

[4] The expression $s[x \mapsto e]$ denotes the expression obtained from s by substituting each occurrence of the variable x by the expression e.

```
for (i1 = 1; i1 ≤ n + 1; i1 = i1 + 1) do
  for(j1 = 1; j1 ≤ ⌊m/2⌋ + 1; j1 = j1 + 1) do
    skip
  end do
end do
```
(a)

```
z = 1
for (i1 = 1; i1 ≤ n + 1; i1 = i1 + 1) do
  for (j1 = 1; j1 ≤ ⌊m/2⌋ + 1; j1 = j1 + 1) do
    z := z + 1
  end do
end do
```
(b)

Fig. 4. ABC-loop format of Figure 2(b) and its instrumented version, where $i = i_1 - 1$ and $j = 2(j_1 - 1)$. Note that $\lfloor m/2 \rfloor \in \mathbb{Z}$.

Based on Algorithm 1 and keeping the notations of (1), we conclude that the general shape of loops that can be converted into ABC-loops is:

```
for (i1 = l; i1 ≤ c; i1 = i1 + inc1) do
  for (i2 = g1(i1); i2 ≤ f1(i1); i2 = i2 + inc2) do
    ...
    for (id = gd−1(i1, ..., id−1); id ≤ fd−1(i1, ..., id−1); id = id + incd) do
      skip
    end do
    ...
  end do
end do
```
(2)

where $l, inc_1, \ldots, inc_d$ are integer-valued symbolic constants, and $g_k \in \mathbb{Z}[i_1, \ldots, i_k]$.

3.3 The Bound Computer

We assume that each program statement is annotated with the time units it needs to be executed. For simplicity, we assume that an iteration of an unnested ABC-loop takes one time unit, and all other instructions of the unnested loop need zero time (e.g. assignment statements take zero time to be executed). That is we compute a bound on the total number of loop iterations of an ABC-loop (1).

In our approach to bound computation, we instrument the innermost loop body of (1) with a new variable that increases at every iteration of the program, and is initialized to 1 before entering the ABC-loop. We denote this variable by z. From (1), we thus obtain:

```
z = 1
for (i1 = 1; i1 ≤ c; i1 = i1 + 1) do
  ...
  for (id = 1; id ≤ fd−1(i1, ..., id−1); id = id + 1) do
    z := z + 1
  end do
  ...
end do
```
(3)

Example 2. The instrumented loop of Figure 4(a) is given in Figure 4(b).

Upper bounds on the value of z give upper bounds on the number of iterations of (3). We are hence left with computing the value of z as a function, called the *z-relation*,

Algorithm 2. Bound Computer

Input: ABC-loop F, initial value z_0 of z
Output: z-relation $zrel$

```
1: inner := loop_body(F)
2: incr := z_reduce_loop(inner)
3: ⟨ovar, oubound⟩ := ⟨outer_iteration_variable(F),outer_iteration_upperbound(F)⟩
4: nvar := fresh_variable()
5: z_i := z_0 + solve_sum(nvar, 1, ovar − 1, incr[ovar ↦ nvar])
6: if isloop(inner) then
7:     zrel := z = Bound Computer(inner, z_i)
8: else
9:     zrel := z = z_i
10: end if
```

over $i_1, \ldots, i_d$. To this end, the value of z at an arbitrary iteration of the outermost loop of (3) is first computed.

Computing the value of z after an arbitrary iteration of the outermost loop of (3). Let us consider a more general loop than (3):

$$
\begin{array}{l}
\underline{\textbf{for}}\ (i_1 = 1;\ i_1 \le c;\ i_1 = i_1 + 1)\ \underline{\textbf{do}} \\
\quad \underline{\textbf{for}}\ (i_2 = 1;\ i_2 \le f_1(i_1);\ i_2 = i_2 + 1)\ \underline{\textbf{do}} \\
\qquad \cdots \\
\qquad \underline{\textbf{for}}\ (i_d = 1;\ i_d \le f_{d-1}(i_1, \ldots, i_{d-1});\ i_d = i_d + 1)\ \underline{\textbf{do}} \\
\qquad\quad z := z + g(i_d) \\
\qquad \underline{\textbf{end do}} \\
\qquad \cdots \\
\quad \underline{\textbf{end do}} \\
\underline{\textbf{end do}}
\end{array}
\tag{4}
$$

where $i_1, \ldots, i_d$, c, $f_1, \ldots, f_{d-1}$ are as in (1), and $g \in \mathbb{Z}[i_d]$. In particular, if $g = 1$ then (4) becomes (3).

Let $s_1, \ldots, s_l$ be nonnegative integers ($l = 1, \ldots, d$) such that $1 \le s_1 \le c$, $1 \le s_2 \le f_1(i_1)$, …, and $1 \le s_l \le f_{l-1}(i_1, \ldots, i_{l-1})$. In the sequel we consider $s_1, \ldots, s_l$ arbitrary but fixed. We write $x^{(l,\|s_1,\ldots,s_l\|)}$ to mean the value of a variable $x \in \{i_1, \ldots, i_d, z\}$ in (4) such that the kth loop of (4) is at its s_kth iteration ($k = 1, \ldots, l$),

We are thus interested in deriving $z^{(1,\|s_1\|)}$ for $s_1 \in \{1, \ldots, c\}$. We proceed as follows. For each loop of (4), starting from the innermost one, we (i) model the assignment over z as a recurrence equation, (ii) deploy symbolic summation algorithms to compute the closed form of z, and (iii) replace the loop by a single assignment over z expressing the relation between the values of z before the first and after the last execution of the loop. Steps (i)-(iii) are recursively applied until all loops of (4) are eliminated.

In more detail, $z^{(1,\|s_1\|)}$ is derived as follows. We start with the innermost loop of (4). The assignment over z is modeled by the recurrence equation:

$$
z^{(d,\|s_1,\ldots,s_d+1\|)} = z^{(d,\|s_1,\ldots,s_d\|)} + g(i_d^{(d,\|s_1,\ldots,s_d-1\|)}),
\tag{5}
$$

yielding:

$$z^{(d,\|s_1,\ldots,s_d\|)} = ini_z + \sum_{k=1}^{s_d} g(i_d^{(d,\|s_1,\ldots,k-1\|)}),$$

where $ini_z = z^{(d,\|s_1,\ldots,0\|)}$ denotes the value of z before entering the innermost loop of (4). The value of $i_d^{(d,\|s_1,\ldots,s_d\|)}$ is computed from the recurrence equation:

$$i_d^{(d,\|s_1,\ldots,s_d+1\|)} = i_d^{(d,\|s_1,\ldots,s_d\|)} + 1.$$

Namely, we have $i_d^{(d,\|s_1,\ldots,s_d\|)} = ini_d + \sum_{k=1}^{s_d} 1$, where $ini_d = 1$ denotes the initial value of i_d (i.e. before the first iteration of the innermost loop of (4)). Hence,

$$i_d^{(d,\|s_1,\ldots,s_d\|)} = s_d + 1. \tag{6}$$

Note that (6) holds for each iteration variable, that is:

$$i_l^{(l,\|s_1,\ldots,s_l\|)} = s_l + 1$$

for every $l \in \{1,\ldots,d\}$. For this reason, in what follows we write i_l instead of $i_l^{(l,\|s_1,\ldots,s_l\|)}$ and use the relation $i_l = s_l + 1$ to speak about the value of i_l at iteration s_l of the lth loop. We thus obtain:

$$z^{(d,\|s_1,\ldots,s_d\|)} = ini_z + \sum_{k=1}^{s_d} g(i_d^{(d,\|s_1,\ldots,k-1\|)}) = ini_z + \sum_{k=1}^{i_d-1} g(k).$$

Since $g \in \mathbb{Z}[i_d]$, the closed form of $\sum_{k=1}^{i_d-1} g(k)$ always exists [6] and can be computed as a polynomial function over i_d (see Section 3.4).

Finally, we consider the last iteration $s_d = i_d - 1 = f_{d-1}(i_1,\ldots,i_{d-1})$ of the innermost loop of (4), and write $incr_d = \sum_{k=1}^{f_{d-1}(i_1,\ldots,i_{d-1})} g(k)$. We make use of $incr_d \in \mathbb{Z}[i_1,\ldots,i_{d-1}]$ to "eliminate" the innermost loop of (4) and obtain:

```
for (i_1 = 1; i_1 ≤ c; i_1 = i_1 + 1) do
  ...
  for (i_{d-1} = 1; i_{d-1} ≤ f_{d-1}(i_1,...,i_{d-2}); i_{d-1} = i_{d-1} + 1) do
    z := z + incr_d
  end do
  ...
end do
```
(7)

Inner loops of (7) can be further eliminated by applying recursively the steps described above, since closed forms of polynomial expressions over $i_1,\ldots,i_d$ yield polynomial expressions over $i_1,\ldots,i_d$ whenever the summation variables are bounded by polynomial expressions. As a result, the total number of increments over z in the s_1th iteration of the outermost loop of (4) is derived. Let us denote this number by $incr_1$. Then:

$$z^{(1,\|s_1\|)} = z_0 + incr_1, \text{ where } z_0 = 1 \text{ is the value of } z \text{ before (4)}.$$

Example 3. Consider Figure 4(b). We aim at deriving $z^{(1,\|s_1\|)}$, where $1 \leq s_1 \leq n+1$ is arbitrary but fixed such that $i_1 = s_1 + 1$.

From the innermost loop of Figure 4(b), we have $z^{(2,\|s_1,s_2+1\|)} = z^{(2,\|s_1,s_2\|)} + 1$ for an arbitrary but fixed s_2, where $1 \leq s_2 \leq \lfloor \frac{m}{2} \rfloor + 1$ and $j_1 = s_2 + 1$. Hence,

$$z^{(2,\|s_1,s_2\|)} = ini_2 + j_1 - 1,$$

where ini_2 is the initial value of z before entering the innermost loop. Further, after $s_2 = j_1 - 1 = \lfloor \frac{m}{2} \rfloor + 1$ iterations of the innermost loop, the total number of increments over z is:

$$incr_2 = \sum_{k=1}^{\lfloor \frac{m}{2} \rfloor + 1} 1 = \lfloor \frac{m}{2} \rfloor + 1.$$

The innermost loop of Figure 4(b) is next eliminated, yielding:

$$\underline{\textbf{for}}\ (i_1 = 1; i_1 \leq n+1; i_1 = i_1 + 1)\ \underline{\textbf{do}}\quad z = z + \lfloor \frac{m}{2} \rfloor + 1\ \underline{\textbf{end do}}$$

with the recurrence equation of z as:

$$z^{(1,\|s_1\|+1)} = z^{(1,\|s_1\|)} + \lfloor \frac{m}{2} \rfloor + 1.$$

Solving this recurrence and using that $z_0 = 1$ is the initial value of z before the outermost loop of Figure 4(b), we obtain:

$$z^{(1,\|s_1\|)} = 1 + \sum_{k=1}^{i_1 - 1} \big(\lfloor \frac{m}{2} \rfloor + 1 \big) = 1 + (i_1 - 1)\big(\lfloor \frac{m}{2} \rfloor + 1 \big).$$

Computing the z-relation among arbitrary values of $z, i_1, \ldots, i_d$. We are interested in deriving the value of $z^{(d,\|s_1,\ldots,s_d\|)}$, where $i_k = s_k + 1$ $(k = 1, \ldots, d)$, from which *the z-relation can be immediately constructed as* $z = z^{(d,\|s_1,\ldots,s_d\|)}$.

The value $z^{(d,\|s_1,\ldots,s_d\|)}$ (and hence the z-relation) is inferred by Algorithm 2 as follows.

(a) The value $incr$ is computed such that $z^{(1,\|s_1\|)} = z_0 + incr$ (line 2 of Algorithm 2);
(b) The outermost loop of (4) is omitted (line 1 of Algorithm 2), yielding:

$$\begin{array}{l} \underline{\textbf{for}}\ (i_2 = 1;\ i_2 \leq f_1(i_1);\ i_2 = i_2 + 1)\ \underline{\textbf{do}} \\ \quad \ldots \\ \quad \underline{\textbf{for}}\ (i_d = 1;\ i_d \leq f_{d-1}(i_1, \ldots, i_{d-1});\ i_d = i_d + 1)\ \underline{\textbf{do}} \\ \quad\quad z := z + g(i_d) \\ \quad \underline{\textbf{end do}} \\ \quad \ldots \\ \underline{\textbf{end do}} \end{array} \tag{8}$$

(c) The value of z at the s_2th iteration of the outermost loop (8) is next computed, where the initial value of z before (8) is considered to be $z^{(1,\|s_1\|)}$ (line 7 of Algorithm 2). As a result, $z^{(2,\|s_1,s_2\|)}$ in the loop (4) is obtained.

(d) Steps (b)-(c) are recursively applied on (8) until no more loops are left and $z^{(d,\|s_1,\ldots,s_d\|)}$ is derived (lines 6-9 of Algorithm 2).

Example 4. Consider Figure 4(b). The outermost loop of Figure 4(b) is omitted (line 1 of Algorithm 2), yielding:

$$\underline{\textbf{for}}\ (j_1 = 1; j_1 \leq \lfloor \frac{m}{2} \rfloor + 1; j_1 = j_1 + 1)\ \underline{\textbf{do}}\ \ z = z + 1\ \underline{\textbf{end do}} \tag{9}$$

The total number of increments $incr_2 = \lfloor \frac{m}{2} \rfloor + 1$ over z made by (9) is computed, as presented in Example 3 (line 2 of Algorithm 2). The value $z_i = z^{(1,\|s_1\|)}$ of z at an iteration $s_1 = i_1 - 1$ of the outermost loop of Figure 4(b) is further obtained (lines 3-5 of Algorithm 2), as:

$$z_i = z_0 + \sum_{nvar=1}^{i_1-1} \big(\lfloor \frac{m}{2} \rfloor + 1\big) = 1 + (i_1 - 1)\big(\lfloor \frac{m}{2} \rfloor + 1\big).$$

Next, Algorithm 2 is called on (9) with the initial value z_i to compute the value of z at an iteration $s_2 = j_1 - 1$ of (9) (line 7 of Algorithm 2). As (9) has no inner loops, we have $incr = 1$ and $z'_i = z_i + \sum_{nvar=1}^{j_1-1} 1$, yielding (lines 2-5 of Algorithm 2):

$$z^{(2,\|s_1,s_2\|)} = z'_i = (i_1 - 1)\big(\lfloor \frac{m}{2} \rfloor + 1\big) + j_1.$$

The z-relation of Figure 4(b) is finally derived (line 9 of Algorithm 2), as:

$$z = (i_1 - 1)\big(\lfloor \frac{m}{2} \rfloor + 1\big) + j_1.$$

To obtain the z-relation of Figure 2(b), we make use of $i = i_1 - 1$ and $j = 2(j_1 - 1)$ and have:

$$z = i\left(\left\lfloor \frac{m}{2} \right\rfloor + 1\right) + \left\lfloor \frac{j}{2} \right\rfloor + 1.$$

Replacing i and j respectively with n and m in the z-relation, the upper bound on loop iteration counts of Figure 2(b) is:

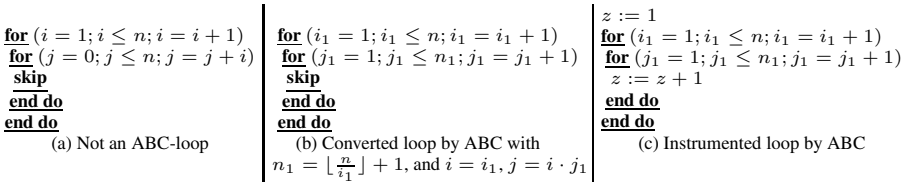
$$(n+1)\big(\left\lfloor \frac{m}{2} \right\rfloor + 1\big).$$

3.4 Symbolic Solver

Simplifying arithmetical expressions and computing closed forms of symbolic sums is performed by the symbolic solver engine of ABC. Our symbolic solver supports the closed form computation of the following sums:

$$\sum_{x=e_1}^{e_2} c_1 \cdot n_1^x \cdot x^{d_1} + \cdots + c_r \cdot n_r^x \cdot x^{d_r}$$

where e_1, e_2 are integer-valued symbolic constants, n_i, d_i are natural numbers, and c_i are rational numbers. Closed forms of such sums always exists [6]. For computing the closed forms of these sums we rely on a simplified version of the Gosper algorithm [6].



(a) Not an ABC-loop

(b) Converted loop by ABC with $n_1 = \lfloor \frac{n}{i_1} \rfloor + 1$, and $i = i_1, j = i \cdot j_1$

(c) Instrumented loop by ABC

Fig. 5. ABC on a non-ABC-loop

We have also instrumented our symbolic solver to handle symbolic sums whose closed forms involve harmonic numbers [7], as discussed in Section 3.5.

3.5 Beyond ABC-Loops

Our approach to bound computation implemented in ABC is complete for ABC-loops and for loops satisfying (2). That is, it *always* returns the z-relation and loop iteration bound of an ABC-loop.

It is worth to be mentioned that some loops violating (2) can still be successfully analyzed by ABC.

Consider Figure 5(a) violating (2), as updates over j depend on i. However, using Algorithm 1 we derive the loop given in Figure 5(b), yielding finally the "instrumented" loop from Figure 5(c). Further, by applying Algorithm 2, we are left with finding the closed form of $\sum_{k=1}^{i_1-1} \left\lfloor \frac{n}{i_1} \right\rfloor$. This sum cannot be further simplified [7]. However, we can compute an upper-bound on its closed form using the relation:

$$\sum_{k=1}^{i_1-1} \left\lfloor \frac{n}{i_1} \right\rfloor \leq \left\lfloor \sum_{k=1}^{i_1-1} \frac{n}{i_1} \right\rfloor = \left\lfloor n \sum_{k=1}^{i_1-1} \frac{1}{i_1} \right\rfloor .$$

Note that $\sum_{k=1}^{i_1-1} \frac{1}{i_1}$ is the harmonic number $H(i_1 - 1)$ arising from the truncation of the harmonic series [7]. We make use of $H(i_1 - 1)$ and derive an upper bound on the loop iteration count of Figure 5(a) as being a harmonic expression. To this end, we have extended our symbolic solver with some simple identities over harmonic numbers. To the best of our knowledge, inferring a tight loop bound for Figure 5(a) is not yet possible by other works.

ABC can thus be successfully applied to loops for which symbolic computation methods can be deployed to compute or approximate closed forms of loop variables. Such cases may arise from nested loops whose inner loop counters are updated by non-constant polynomial functions in the outer loop counters (i.e. yielding iteration bounds as harmonic numbers).

4 Experiments

We applied ABC to a large number of examples, including benchmark programs from recent work on timing analysis [21] as well as from the JAMA package [13].

Table 1. Experimental results obtained by ABC on benchmark examples

Loop	z-relation	Iteration bound	Time [s]
for $(i = a; i \leq b; i = i + 1)$ **skip** **end do**	$z = 1 + i - a$	$1 + b - a$	0.172
for $(i = 0; i \leq n - 1; i = i + 1)$ **for** $(j = 0; j \leq i; j = j + 1)$ **skip** **end do** **end do**	$z = 1 + j + \frac{i+i^2}{2}$	$\frac{n+n^2}{2}$	0.219
for $(i = 1; i \leq m; i = i + 1)$ **for** $(j = 1; j \leq i; j = j + 1)$ **for** $(k = i + 1; k \leq m; k = k + 1)$ **for** $(l = 1; l \leq k; l = l + 1)$ **skip** **end do** **end do** **end do** **end do**	$z = \frac{i^2m^2-im^2+i^2m-im}{4} + \frac{i^2-i^4}{8} + \frac{i^3-i}{12} + \frac{jm+jm^2+k^2}{2} - \frac{m^2+ji^2+ji+m+k}{2} + 1$	$\frac{3m^4+2m^3-3m^2-2m}{24}$	1.281
for $(i = 0; i \leq (\frac{n*n*n}{2} - 1); i = i + 1)$ **for** $(j = 0; j \leq n - 1; j = j + 1)$ **for** $(k = 0; k \leq j - 1; k = k + 1)$ **skip** **end do** **end do** **end do**	$z = 1 + k + \frac{in^2-in+j^2-j}{2}$	$\frac{n^5-n^4}{4}$	0.234
for $(i = 1; i \leq n; i = i + 1)$ **for** $(j = 1; j \leq i; j = j + 1)$ **skip** **end do** **end do**	$z = \frac{i^2-i}{2} + j$	$\frac{n^2+n}{2}$	0.203
for $(i = 1; i \leq n; i = i + 1)$ **for** $(j = i; j \leq n; j = j + 1)$ **skip** **end do** **end do**	$z = (i - 1)n + j + \frac{i-i^2}{2}$	$\frac{n^2+n}{2}$	0.203
for $(j = 1; i \leq m; j = j + 1)$ **for** $(i = 1; i \leq n; i = j + 1)$ **skip** **end do** **end do**	$z = i + (j - 1)n$	nm	0.187
for $(i = n; i \geq 1; i = i - 1)$ **for** $(j = m; j \geq 1; j = j - 1)$ **skip** **end do** **end do**	$z = (n-i+1)m-j+1$	nm	0.188

Tables 1 and 2 summarize some of our results obtained on a machine with a 2.8 GHz Intel Core 2 Duo processor and 2GB of RAM.

The first four programs of Table 1 are examples taken from [21], whereas the last four programs of Table 1 are loops taken from the JAMA package [13]. The examples of Table 2 are our own benchmark examples, and illustrate the power of ABC in handling nested loops whose inner loop counters polynomially depend on its outer loop counters. The difference between the first four programs of Table 2 is given by the mixed incremental/decremental updates and smallest/greatest values of the loop counters. Note that the last three programs of Table 2 yield polynomial loop bounds as sums of multivariate monomials.

Table 2. Further experimental results obtained by ABC

Loop	z-relation	Iteration bound	Time [s]
for $(i = 1; i \leq n; i = i + 1)$ **for** $(j = 1; j \leq m; j = j + 1)$ **skip** **end do** **end do**	$z = j + (i - 1)m$	nm	0.187
for $(i = n; i \geq 1; i = i - 1)$ **for** $(j = 1; j \leq m; j = j + 1)$ **skip** **end do** **end do**	$z = j + (n - i)m$	nm	0.187
for $(i = n; i \geq 1; i = i - 1)$ **for** $(j = m; j \geq 1; j = j - 1)$ **skip** **end do** **end do**	$z = 1 - j + (n - i + 1)m$	nm	0.187
for $(i = n; i \geq 1; i = i - 1)$ **for** $(j = m; j \geq m; j = j - 1)$ **skip** **end do** **end do**	$z = 1 - i - j + m + n$	n	0.203
for $(i = a; i \leq b; i = i + 1)$ **for** $(j = c; j \leq d; j = j + 1)$ **for** $(k = i - j; k \leq i + j; k = k + 1)$ **skip** **end do** **end do** **end do**	$z = 1 - 2ad + 2id - ad^2 + id^2 + ac^2 - ic^2 + j^2 - c^2 + j - a + k$	$(c^2 - (d + 1)^2)(a - b - 1)$	0.328
for $(i = n; i \geq 1; i = i - 1)$ **for** $(j = 1; j \leq m; j = j + 1)$ **for** $(k = i; k \leq i + j; k = k + 1)$ **for** $(l = 1; l \leq k; l = l + 1)$ **skip** **end do** **end do** **end do** **end do**	$z = -\frac{m^2+3m+2}{4} i^2 + (\frac{j^2+j-1}{2} - \frac{2m^3+9m^2+13}{12})i + \frac{k^2-k}{2} + \frac{j^3-j}{6} + 1 + \frac{2m^2+3mn+9m+9n+13}{12} mn$	$\frac{2m^2+3mn+9m+9n+13}{12} mn$	0.625
for $(i = n; i \geq 1; i = i - 1)$ **for** $(j = 1; j \leq m; j = j + 1)$ **for** $(k = i; k \leq p; k = k + 1)$ **for** $(l = q; l \leq j; l = l - 1)$ **skip** **end do** **end do** **end do** **end do**	$z = \frac{3j-ij-j^2+ij^2}{2} + k - l - p + \frac{i^2m+im^2-i^2m^2-im}{4} + jq - jk + kq - pq + \frac{mn-m^2n-mn^2+m^2n^2}{4} + \frac{3jp-j^2p-imp+im^2p}{2} - ijq + jpq + \frac{mnp-m^2np-imq}{2} - impq + \frac{im^2q+mnq-mn^2q}{2} + mnpq$	$\frac{m^2n^2-mn^2-m^2n+mn}{4} + \frac{mnp-m^2np+mnq-mn^2q}{2} + mnpq$	0.375

The first column of Table 1 (respectively of Table 2) presents the loop being fed into ABC, the second column shows the z-relation derived by ABC, whereas the third one presents the number of loop iterations computed by ABC. The forth column gives the time (in seconds) needed by ABC to infer bounds on loop iteration counts [5]. Note that iteration bounds are *integer-valued* polynomial expressions (e.g. $n^2 + n$ is divisible by 2).

[5] Note, that the timings given in Tables 1 and 2 include also the required time to launch the JVM.

5 Conclusions

We describe the software tool ABC for automatically deriving tight symbolic upper bounds on loop iteration counts of a special class of loops, called the ABC-loops. The system was successfully tried on a large number of examples. The derived symbolic bounds express non-trivial (polynomial and harmonic) relations over loop variables.

Future work includes extending ABC to handle more complex sums, such as e.g. fractions of polynomials [17], including more sophisticated control-flow refinement techniques, such as [8], into ABC, and improving the loop converter of ABC for handling more complex loops.

References

1. Albert, E., Arenas, P., Genaim, S., Puebla, G.: Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis. In: Alpuente, M., Vidal, G. (eds.) SAS 2008. LNCS, vol. 5079, pp. 221–237. Springer, Heidelberg (2008)
2. Birkeland, B.: Calculus and Algebra with MathCad 2000. Haeftad. Studentlitteratur (2000)
3. Cousot, P., Cousot, R.: Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: Proc. of POPL, pp. 238–252 (1977)
4. Danaila, I., Joly, P., Kaber, S.M., Postel, M.: An Introduction to Scientific Computing: Twelve Computational Projects Solved with MATLAB. Springer, Heidelberg (2007)
5. Ferdinand, C., Heckmann, R.: aiT: Worst Case Execution Time Prediction by Static Program Analysis. In: Proc. of IFIP Congress Topical Sessions, pp. 377–384 (2004)
6. Gosper, R.W.: Decision Procedures for Indefinite Hypergeometric Summation. PNAS 75, 40–42 (1978)
7. Graham, R.L., Knuth, D.E., Patashnik, O.: Concrete Mathematics, 2nd edn. Addison-Wesley Publishing Company, Reading (1989)
8. Gulwani, S., Jain, S., Koskinen, E.: Control-flow Refinement and Progress Invariants for Bound Analysis. In: Proc. of PLDI, pp. 375–385 (2009)
9. Gustafsson, J., Ermedahl, A., Sandberg, C., Lisper, B.: Automatic Derivation of Loop Bounds and Infeasible Paths for WCET Analysis Using Abstract Execution. In: Proc. of RTSS, pp. 57–66 (2006)
10. Healy, C.A., Sjödin, M., Rustagi, V., Whalley, D.B., van Engelen, R.: Supporting Timing Analysis by Automatic Bounding of Loop Iterations. Real-Time Systems 18(2/3), 129–156 (2000)
11. Henzinger, T.A., Hottelier, T., Kovacs, L.: Valigator: A Verification Tool with Bound and Invariant Generation. In: Cervesato, I., Veith, H., Voronkov, A. (eds.) LPAR 2008. LNCS (LNAI), vol. 5330, pp. 333–342. Springer, Heidelberg (2008)
12. Hermenegildo, M.V., Puebla, G., Bueno, F., Lopez-Garcia, P.: Integrated Program Debugging, Verification, and Optimization using Abstract Interpretation (and the Ciao System Preprocessor). Sci. Comput. Program. 58(1-2), 115–140 (2005)
13. Hicklin, J., Moler, C., Webb, P., Boisvert, R.F., Miller, B., Pozo, R., Remington, K.: JAMA: A Java Matrix Package (2005), `http://math.nist.gov/javanumerics/jama/`
14. Jost, S., Loidl, H., Hammond, K., Scaife, N., Hofmann, M.: "Carbon Credits" for Resource-Bounded Computations Using Amortised Analysis. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 354–369. Springer, Heidelberg (2009)
15. Müller-Olm, M., Petter, M., Seidl, H.: Interprocedurally Analyzing Polynomial Identities. In: Durand, B., Thomas, W. (eds.) STACS 2006. LNCS, vol. 3884, pp. 50–67. Springer, Heidelberg (2006)

16. Navas, J., Mera, E., Lopez-Garcia, P., Hermenegildo, M.V.: User-Definable Resource Bounds Analysis for Logic Programs. In: Dahl, V., Niemelä, I. (eds.) ICLP 2007. LNCS, vol. 4670, pp. 348–363. Springer, Heidelberg (2007)
17. Nemes, I., Petkovsek, M.: RComp: A Mathematica Package for Computing with Recursive Sequences. Journal of Symbolic Computation 20(5-6), 745–753 (1995)
18. Odersky, M.: The Scala Language Specification (2008), http://www.scala-lang.org
19. Prantl, A., Knoop, J., Schordan, M., Triska, M.: Constraint Solving for High-Level WCET Analysis. CoRR, abs/0903.2251 (2009)
20. Stewart, G.W.: JAMPACK: A Java Package For Matrix Computations, http://www.mathematik.hu-berlin.de/~lamour/software/JAVA/Jampack/
21. van Engelen, R.A., Birch, J., Gallivan, K.A.: Array Data Dependence Testing with the Chains of Recurrences Algebra. In: Proc. of IWIA, pp. 70–81 (2004)
22. Wolfram, S.: The Mathematica Book. Version 5.0. Wolfram Media, Champaign (2003)

Hardness of Preorder Checking for Basic Formalisms

Laura Bozzelli, Axel Legay, and Sophie Pinchinat

IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France

Abstract. We investigate the complexity of preorder checking when the specification is a flat finite-state system whereas the implementation is either a non-flat finite-state system or a standard timed automaton. In both cases, we show that simulation checking is EXPTIME-hard, and for the case of a non-flat implementation, the result holds even if there is no synchronization between the parallel components and their alphabets of actions are pairwise disjoint. Moreover, we show that the considered problems become PSPACE-complete when the specification is assumed to be deterministic. Additionally, we establish that comparing a synchronous non-flat system with no hiding and a flat system is PSPACE-hard for any relation between trace containment and bisimulation equivalence.

1 Introduction

One popular approach to formal verification of reactive systems is equivalence/preorder checking between a specification and an implementation which formally describe at a different level of abstraction a given system. This scenario may arise either because the design is being carried out in an incremental fashion, or because the system is too complex and an abstraction needs to be used to verify its properties. In this context, the verification problem is mathematically formulated as a question whether a behavioral equivalence or preorder holds between the labeled state-transition graphs of the specification and the implementation. Decidability and complexity issues for equivalence/preorder checking have been addressed for various computational models of reactive systems (see [10] for a survey of the existing results for infinite-state systems). Moreover, many notions of equivalences or preorders have been investigated, which turn out to be useful for specific aims. Van Glabbeek [21] classified such equivalences/preorders in a hierarchy, where bisimulation equivalence is the finest and trace containment is the coarsest.

Non-flat finite-state systems. Finite-state systems are a natural and a primary target in formal verification. When the systems are given as explicit (flat) state-transition graphs, then many relevant verification problems are tractable. For example, simulation-preorder checking and bisimulation-equivalence checking are PTIME-complete [2,15]. However, in a concurrent setting, the system under consideration is typically the parallel composition of many components (we call such systems *non-flat systems*). As a consequence, the size of the global state-transition graph is usually exponential in the size of the system presentation. This phenomenon is known as 'state explosion', and coping with this problem is one of the most important issues in computer-aided verification. From a theoretical point of view, the goal is to understand better which verification problems have to face state explosion in an intrinsic way and which special manner of combining subsystems could avoid state explosion.

E.M. Clarke and A. Voronkov (Eds.): LPAR-16, LNAI 6355, pp. 119–135, 2010.

Different models of non-flat systems have been investigated. The simplest one is the fully asynchronous model (synchronization-free non-flat systems), where at each step exactly one component performs a transition [20,13]. For more complex models, the components can communicate by shared actions (or by access to shared variables), and, additionally, actions may be 'hidden' (i.e., replaced by some special action) after the parallel composition [7,12,14]. For synchronization-free non-flat systems, some problems in equivalence/preorder checking are still tractable. In particular, Groote and Moller [5] have shown that if the alphabets of actions of the components are assumed to be pairwise disjoint, then checking bisimulation equivalence (and other equivalences which satisfy a certain set of axioms) is PTIME-complete. Moreover, for these systems and also for basic synchronous non-flat systems with no hiding (where the components are forced to synchronize on shared actions), checking trace containment/equivalence has the same complexity as for flat systems, i.e. it is PSPACE-complete [17,20]. Simulation-preoder checking and bisimulation-equivalence checking for synchronous non-flat systems (with or without hiding) are hard, since they are EXPTIME-complete [9,4,11,16]. Checking whether a *synchronous* non-flat system (where the communication is allowed by access to shared variables) is simulated by a *flat system* remains EXPTIME-hard [4]. Moreover, Rabinovich [14] has shown that preorder/equivalence checking between a synchronous non-flat system *with hiding* and a flat system is PSPACE-hard for any relation between trace containment and bisimulation. More recently, Muscholl and Walukiewicz [13] have obtained a surprising result: checking whether a deterministic *flat system* is simulated by a *synchronization-free* non-flat system whose components are deterministic remains EXPTIME-hard. The exact complexity for the converse direction, i.e., whether a synchronization-free non-flat system is simulated by a flat system is open.

Timed automata. Timed automata (TA) introduced by Alur and Dill [1] are a widely accepted formalism to model the behavior of real-time systems. Equivalence/preorder checking for this infinite-state computational model has been addressed in many papers. Timed language containment/equivalence is undecidable [1]. Timed bisimulation and timed simulation have been shown to be decidable and in EXPTIME in [19] and [18], respectively; matching lower bounds have been given in [11]. Time-abstract simulation and time-abstract bisimulation have been considered in [6] and are in EXPTIME.

Our contribution. We investigate the complexity of preorder checking when the specification is a flat system and the implementation is either a timed automaton or a non-flat system. Note that the considered setting is relevant since the specification is more abstract than the implementation, and, thus, it is usually described by a simple formalism. The results obtained are as follows:

- Checking whether a timed automaton is time-abstract simulated by a flat system is EXPTIME-hard.
- Checking whether a synchronization-free non-flat system is simulated by a flat system is EXPTIME-hard even if the components of the implementation are assumed to be deterministic and with pairwise disjoint alphabets of actions.
- The two problems above are PSPACE-hard if the specification is deterministic.

– Comparing a synchronous non-flat system *with no hiding* and a flat system is PSPACE-hard for any relation lying in between trace containment and bisimilarity.

Our first result in a sense improves the EXPTIME-hardness result for timed simulation between timed automata, by showing that checking timed-abstract simulation remains EXPTIME-hard even if one of the compared TA is replaced by a flat (finite-state) system. Regarding our second and third results, they imply that refinement checking of non-flat systems is intractable even for the simplest model (action-based parallel compositions of deterministic components with pairwise disjoint alphabets) and even if the specification is flat (note that for deterministic specifications, simulation preorder and trace containment are equivalent notions). Finally, our fourth result significantly strengthens the PSPACE-hardness result of Rabinovich [14] in which hiding is involved.

It is interesting to observe that our second result is surprising for the following reasons: *if the alphabets of the components are assumed to be pairwise disjoint*, then bisimulation checking between non-flat systems is in PTIME [5], and simulation checking between a flat implementation and a non-flat specification (i.e., whether a flat system is simulated by a non-flat system) is in PTIME as well [13].[1]

Note that the lower bounds for the first three problems are optimal since they match well-known upper bounds in the literature (see Section 2).

Due to lack of space, for the omitted details we refer the interested reader to a forthcoming extended version of this paper.

2 Preliminaries

Labeled transition systems and simulation preorder. A *labeled transition system* (LTS) over a (possibly infinite) set of actions Act is a tuple $\mathcal{G} = \langle Act, S, s_0, \Delta \rangle$, where S is a (possibly infinite) set of states, $s_0 \in S$ is a designated initial state, and $\Delta \subseteq S \times Act \times S$ is the transition relation. A transition $(s,a,s') \in \Delta$ is denoted by $s \xrightarrow{a} s'$. We say that $\mathcal{G}$ is *deterministic* if for all $s \in S$ and $a \in Act$, there is at most one transition of the form $s \xrightarrow{a} s'$ for some state s'. The set of traces of $\mathcal{G}$, $Tr(\mathcal{G})$, is the set of finite words $a_1, \ldots, a_n$ over Act such that there is a path in $\mathcal{G}$ from the initial state of the form $s_0 \xrightarrow{a_1} s_1 \ldots s_{n-1} \xrightarrow{a_n} s_n$.

An Act-labeled tree is an unordered finite or infinite tree whose edges are labeled by actions in Act. Note that an Act-labeled tree is a particular LTS over Act whose initial state is the root. For a LTS $\mathcal{G}$ over Act and a state s of $\mathcal{G}$, the *unwinding of $\mathcal{G}$ from s*, written $Unw(\mathcal{G},s)$, is the Act-labeled tree defined in the usual way.

Given two LTS $\mathcal{G}_1 = \langle Act_1, S_1, s_1^0, \Delta_1 \rangle$ and $\mathcal{G}_2 = \langle Act_2, S_2, s_2^0, \Delta_2 \rangle$, a *simulation from $\mathcal{G}_1$ to $\mathcal{G}_2$* is a relation $\mathcal{R} \subseteq S_1 \times S_2$ satisfying the following for all $(s_1,s_2) \in \mathcal{R}$: if $s_1 \xrightarrow{a} s_1' \in \Delta_1$ for some state $s_1' \in S_1$ and $a \in Act_1$, then there is some state $s_2' \in S_2$ so that $s_2 \xrightarrow{a} s_2' \in \Delta_2$ and $(s_1', s_2') \in \mathcal{R}$. If, additionally, the inverse of $\mathcal{R}$ is a simulation from $\mathcal{G}_2$ to $\mathcal{G}_1$, then we say that $\mathcal{R}$ is a *bisimulation* from $\mathcal{G}_1$ to $\mathcal{G}_2$. Given states $s_1 \in S_1$ and $s_2 \in S_2$, we say that s_1 *is simulated by* s_2 (resp., s_1 *and* s_2 *are bisimilar*) if there is a simulation

[1] In [13], membership in PTIME is shown for the case in which the components of the specification and the implementations are assumed to be deterministic. However, the proof can be easily extended to the case in which this requirement is relaxed.

(resp., bisimulation) $\mathcal{R}$ from $\mathcal{G}_1$ to $\mathcal{G}_2$ such that $(s_1,s_2)\in\mathcal{R}$. The *simulation preorder* $\preceq$ (resp., the *bisimulation equivalence* $\sim_{bis}$) is the binary relation over LTS defined as: $\mathcal{G}_1\preceq\mathcal{G}_2$ (resp., $\mathcal{G}_1\sim_{bis}\mathcal{G}_2$) iff the initial state s_1^0 of $\mathcal{G}_1$ is simulated by the initial state s_2^0 of $\mathcal{G}_2$ (resp., s_1^0 and s_2^0 are bisimilar). Moreover, the *trace containment preorder* $\sqsubseteq_{tr}$ is defined as: $\mathcal{G}_1\sqsubseteq_{tr}\mathcal{G}_2$ iff $Tr(\mathcal{G}_1)\subseteq Tr(\mathcal{G}_2)$. Note that for each $\trianglelefteq\in\{\sim_{bis},\preceq,\sqsubseteq_{tr}\}$, $\mathcal{G}_1\trianglelefteq\mathcal{G}_2$ iff $Unw(\mathcal{G}_1,s_1^0)\trianglelefteq Unw(\mathcal{G}_2,s_2^0)$.

Flat and Non-flat systems. A *flat system* (FS) is an LTS $\mathcal{A}=\langle Act,Q,q_0,\Delta\rangle$ such that Act and the set of states Q are both finite. The size of $\mathcal{A}$ is $|\mathcal{A}|=|Q|+|\Delta|$.

A *synchronization-free Non-Flat System* (NFS) is a tuple $\mathcal{S}=\langle\mathcal{A}_1,\ldots,\mathcal{A}_k\rangle_{SF}$ such that each component $\mathcal{A}_i=\langle Act_i,Q_i,q_i^0,\Delta_i\rangle$ is a FS. $\mathcal{S}$ induces the FS $[\![\mathcal{S}]\!]$ given by

$$[\![\mathcal{S}]\!]=\langle\bigcup_{i=1}^{i=k}Act_i,Q_1\times\ldots\times Q_k,(q_1^0,\ldots,q_k^0),\Delta_{SF}\rangle$$

where Δ_{SF} is defined as follows: $((q_1,\ldots,q_k),a,(q_1',\ldots,q_k'))\in\Delta_{SF}$ iff for some i, $(q_i,a,q_i')\in\Delta_i$ and for all $j\neq i$, we have $q_j'=q_j$.

We also consider *synchronous* NFS $\mathcal{S}=\langle\mathcal{A}_1,\ldots,\mathcal{A}_k\rangle$, where the components $\mathcal{A}_i=\langle Act_i,Q_i,q_i^0,\Delta_i\rangle$ communicate by synchronization on common actions. Formally, $\mathcal{S}$ induces the FS given by

$$[\![\mathcal{S}]\!]=\langle\bigcup_{i=1}^{i=k}Act_i,Q_1\times\ldots\times Q_k,(q_1^0,\ldots,q_k^0),\Delta\rangle$$

where Δ is defined as: $((q_1,\ldots,q_k),a,(q_1',\ldots,q_k'))\in\Delta$ iff for each i, $(q_i,a,q_i')\in\Delta_i$ if $a\in Act_i$, and $q_i'=q_i$ otherwise. Note that all the components $\mathcal{A}_i$ with $a\in Act_i$ must perform a transition labeled by a. Moreover, note that if distinct components have no actions in common, then $[\![\langle\mathcal{A}_1,\ldots,\mathcal{A}_k\rangle]\!]=[\![\langle\mathcal{A}_1,\ldots,\mathcal{A}_k\rangle_{SF}]\!]$. The size of $\mathcal{S}$ is $|\mathcal{S}|=\sum_{i=1}^{i=n}|\mathcal{A}_i|$.

Timed automata. Let $\mathbb{R}_{\geq 0}$ be the set of non-negative reals. Fix a finite set of *clock* variables X. The set $C(X)$ of *clock constraints* (*over* X) is the set of boolean combinations of formulas of the form $x\leq c$ or $x<c$, where $x\in X$, and c is a natural number. A (*clock*) *valuation* (*over* X) is a function $v:X\to\mathbb{R}_{\geq 0}$ that maps every clock to a non-negative real number. Whether a valuation v *satisfies* a clock constraint $g\in C(X)$, denoted $v\models g$, is defined in a natural way. For $t\in\mathbb{R}_{\geq 0}$, the valuation $v+t$ is defined as $(v+t)(x)=v(x)+t$ for all $x\in X$. For $Y\subseteq X$, the valuation $v[Y:=0]$ is defined as $(v[Y:=0])(x)=0$ if $x\in Y$ and $(v[Y:=0])(x)=v(x)$ otherwise.

Definition 1. [1] *A* timed automaton (TA) *over a finite set of actions Act is a tuple* $\mathcal{T}=\langle Act,X,Q,q_0,\rho\rangle$*, where* Q *is a finite set of* locations*,* $q_0\in Q$ *is the* initial location*, and* $\rho\subseteq Q\times Act\times C(X)\times 2^X\times Q$ *is a finite* transition relation*.*

The TA $\mathcal{T}$ induces an infinite-state LTS $[\![\mathcal{T}]\!]=\langle\mathbb{R}_{\geq 0}\times Act,S,s_0,\Delta\rangle$ over $\mathbb{R}_{\geq 0}\times Act$, where S is the set of pairs (q,v) such that $q\in Q$ and v is a clock valuation, $s_0=(q_0,\overrightarrow{0})$ ($\overrightarrow{0}$ assigns to each clock value 0), and Δ is defined as follows: $(q,v)\xrightarrow{(t,a)}(q',v')\in\Delta$ *iff* there is a transition $(q,a,g,Y,q')\in\rho$ such that $v+t\models g$ and $v'=(v+t)[Y:=0]$.

The *abstract* LTS associated with $\mathcal{T}$ is $[\![\mathcal{T}]\!]_{abs} = \langle Act, S, s_0, \Delta_{abs}\rangle$, where $(q,v) \xrightarrow{a} (q',v') \in \Delta_{abs}$ iff $(q,v) \xrightarrow{(t,a)} (q',v') \in \Delta$ for some $t \geq 0$. We say that $\mathcal{T}$ is *strongly timed-deterministic* if $[\![\mathcal{T}]\!]_{abs}$ is deterministic and for each $(q,v) \xrightarrow{a} (q',v') \in \Delta_{abs}$, there is exactly one timestamp t such that $(q,v) \xrightarrow{(t,a)} (q',v') \in \Delta$.

Investigated problems. We consider the following decision problems:

Problem 1: given a TA $\mathcal{T}$ and a FS $\mathcal{B}$, does $[\![\mathcal{T}]\!]_{abs} \preceq \mathcal{B}$ hold?
Problem 2: given a *synchronization-free* NFS $\mathcal{S}$ and a FS $\mathcal{B}$, does $[\![\mathcal{S}]\!] \preceq \mathcal{B}$ hold?
Problem 3($\trianglelefteq$): given a *synchronous* NFS $\mathcal{S}$ and a FS $\mathcal{B}$, does $[\![\mathcal{S}]\!] \trianglelefteq \mathcal{B}$ hold?

where $\trianglelefteq$ is a fixed binary relation on LTS. We also consider the *deterministic versions* of Problems 1 and 2, where the FS $\mathcal{B}$ above is assumed to be deterministic.

Theorem 1. *Problems 1 and 2 are in* EXPTIME*, while their deterministic versions are in* PSPACE.

Proof. Membership in EXPTIME for Problems 1 and 2 directly follows from the following:

Fact 1 [2]: given two FS $\mathcal{A}_1$ and $\mathcal{A}_2$, checking whether $\mathcal{A}_1 \preceq \mathcal{A}_2$ is in PTIME.
Fact 2: for a NFS $\mathcal{S}$, the size of the FS $[\![\mathcal{S}]\!]$ is singly exponential in the size of $\mathcal{S}$.
Fact 3 [6]: given two TA $\mathcal{T}_1$ and $\mathcal{T}_2$, checking whether $[\![\mathcal{T}_1]\!]_{abs} \preceq [\![\mathcal{T}_2]\!]_{abs}$ is in EXPTIME.

Membership in PSPACE for the deterministic versions of Problems 1 and 2 directly follows from Fact 2 and the following:
Fact 4: for two LTS $\mathcal{G}_1$ and $\mathcal{G}_2$ such that $\mathcal{G}_2$ is deterministic, $\mathcal{G}_1 \preceq \mathcal{G}_2$ iff $\mathcal{G}_1 \sqsubseteq_{tr} \mathcal{G}_2$.
Fact 5 [1]: given a TA $\mathcal{T}$, one can construct a FS $\mathcal{A}_{\mathcal{T}}$ (*region automaton*) of size singly exponential in the size of $\mathcal{T}$ such that $Tr(\mathcal{A}_{\mathcal{T}}) = Tr([\![\mathcal{T}]\!]_{abs})$.
Fact 6: given a deterministic FS $\mathcal{A}$ over Act, one can trivially construct in linear-time a standard finite-state automaton which accepts all and only the words in $Act^* \setminus Tr(\mathcal{A})$.
Fact 7 [8]: checking emptiness of the intersection of the languages accepted by two (nondeterministic) finite-state automata is in NLOGSPACE.

In the rest of this paper, we provide lower bounds for Problems 1 and 2 (and their deterministic versions) which match the upper bounds of Theorem 1. Moreover, we show that Problem 3($\trianglelefteq$) is PSPACE-hard for any binary relation $\trianglelefteq$ lying in between trace containment and bisimulation equivalence.

3 EXPTIME-Hardness of Problems 1 and 2

In this section, we show that Problems 1 and 2 are both EXPTIME-hard by polynomial-time reductions from the acceptance problem for *linearly-bounded alternating* Turing Machines (TM) with a binary branching degree, which is EXPTIME-complete [3].

In the rest of this section, we fix such a TM machine $\mathcal{M} = \langle A, Q = Q_\forall \cup Q_\exists \cup \{q_{acc}, q_{rej}\}, q_0, \delta, \{q_{acc}\}\rangle$, where A is the input alphabet, $Q_\exists$ (resp., $Q_\forall$) is the set of existential (resp., universal) states, q_0 is the initial state, $q_{acc} \notin Q_\forall \cup Q_\exists$ is the (terminal) accepting state, $q_{rej} \notin Q_\forall \cup Q_\exists$ is the (terminal) rejecting state, and $\delta : (Q_\forall \cup Q_\exists) \times A \rightarrow$

$(Q \times A \times \{+1,-1\}) \times (Q \times A \times \{+1,-1\})$ is the transition function. In each non-terminal step (i.e., the current state is in $Q_\forall \cup Q_\exists$), $\mathcal{M}$ overwrites the tape cell being scanned, and the tape head moves one position to the left (-1) or right ($+1$). Moreover, we fix an input $\alpha \in A^*$ and consider the parameter $n = |\alpha|$.

Since $\mathcal{M}$ is linearly bounded, w.l.o.g. we assume that $\mathcal{M}$ uses exactly n tape cells when started on the input α. Hence, a TM configuration (of $\mathcal{M}$ over α) is a word $C = \beta_1,(a,q),\beta_2 \in A^* \cdot (A \times Q) \cdot A^*$ of length exactly n denoting that the tape content is β_1,a,β_2, the current state is q, and the tape head is at position $|\beta_1|+1$. The initial configuration C_α is given by $(\alpha(1),q_0),\alpha(2),\ldots,\alpha(n)$. Moreover, w.l.o.g. we assume that when started on C_α, no matter what are the universal and existential choices, $\mathcal{M}$ always *halts* by reaching a terminal configuration C, i.e. such that the associated state, denoted by $q(C)$, is in $\{q_{acc},q_{rej}\}$ (this assumption is standard, see [3]).

It is convenient to define the notion of acceptance of $\mathcal{M}$ as follows. For each $q \in Q$, define $Val(q) = 1$ if $q = q_{acc}$, and $Val(q) = 0$ otherwise. A (*full*) *pseudo-computation tree T of $\mathcal{M}$* from a TM configuration C is a binary tree whose nodes are labeled by TM configurations and such that the root is labeled by C, the internal nodes have two children, and the leaves are labeled by terminal configurations. If T is finite, then its *boolean value* $Val(T) \in \{0,1\}$ is defined as follows. If T consists just of the root, then $Val(T) = Val(q(C))$. Otherwise, let T_L and T_R be the trees rooted at the children of the root of T. Then, $Val(T)$ is $Val(T_L) \vee Val(T_R)$ if $q(C) \in Q_\exists$, and $Val(T_L) \wedge Val(T_R)$ otherwise. The tree T *leads to acceptance* if $Val(T) = 1$. A (*full*) *computation tree* is a pseudo-computation tree which is faithful to the evolution of $\mathcal{M}$. $\mathcal{M}$ accepts α iff the computation tree of $\mathcal{M}$ over C_α, which by our assumption is finite, leads to acceptance.

In the rest of this section, we show that it is possible to construct in *polynomial time* (in the sizes of the fixed TM $\mathcal{M}$ and input α) an instance I of Problem 1 (resp., Problem 2) such that $\mathcal{M}$ accepts α iff the instance I has a positive answer.

Preliminary step: encoding of acceptance by simulation. Before illustrating the polynomial reductions to Problems 1 and 2 (in Subsections 3.1 and 3.2, respectively), we consider a preliminary step in which we define for a given finite set of actions *Act* and a given encoding of the TM configurations by words over *Act*, two *Act*-labeled trees ET_{C_α} and $VT_{q(C_\alpha),1}$ such that $\mathcal{M}$ accepts the fixed input α iff the root of ET_{C_α} is simulated by the root of $VT_{q(C_\alpha),1}$. The tree ET_{C_α} is finite and deterministic, and it is a natural encoding of the computation tree of $\mathcal{M}$ over C_α, while the tree $VT_{q(C_\alpha),1}$ is *infinite* and *non-deterministic*, and encodes in a suitable way all the possible *finite pseudo-computation trees* of $\mathcal{M}$ which *lead to acceptance*. The two encodings ensure that the root of ET_{C_α} is *simulated* by the root of $VT_{q(C_\alpha),1}$ iff ET_{C_α} is 'contained' in $VT_{q(C_\alpha),1}$, i.e. the full computation tree of $\mathcal{M}$ over α leads to acceptance. Now, we define these trees.

Assumptions: we assume that $Act \supseteq Q \cup (Q \times \{0,1\}) \cup \{L,R\}$ and each TM configuration C is encoded by a finite word over *Act*, denoted by $code(C)$. We denote by *Codes* the finite set of these codes, which are assumed to have the same length. The precise definition of *Act* and *Codes* will depend on the specific problem we consider (either Problem 1 or Problem 2).

For each $code \in Codes$, let T_{code} be the finite *Act*-labeled tree, which is a chain and whose unique maximal path from the root is labeled by *code*. For a *non-terminal*

configuration $C = \beta_1, (a,q), \beta_2$ (i.e., such that $q \in Q_\forall \cup Q_\exists$), $succ_L(C)$ (resp., $succ_R(C)$) denotes the TM successor of C obtained by choosing the left (resp., right) triple in $\delta(q,a)$. A *good* configuration is a configuration reachable from C_α.

Definition 2. [*Emulation trees*] *For each* good *TM configuration C, the finite deterministic Act-labeled* emulation tree of $\mathcal{M}$ from C, *denoted by* ET_C, *is inductively defined as follows. The tree* ET_C *is obtained from* $T_{code(C)}$ *by adding an edge from the leaf* x_m *of* $T_{code(C)}$ (main node of level 0) *to a new node* x_c (choice node of level 0) *such that:*

- C is terminal*: x_c is a leaf and the edge from x_m to x_c is labeled by $(q(C), Val(q(C)))$.*
- C is not terminal*: let $C_L = succ_L(C)$ and $C_R = succ_R(C)$. Then, x_c has two children x_L and x_R so that the subtree rooted at x_L (resp., x_R) is isomorphic to ET_{C_L} (resp., ET_{C_R}). The edge from the main node x_m to the choice node x_c is labeled by $q(C)$, and the edge from x_c to x_L (resp., to x_R) is labeled by L (resp., R).*

The structure of the emulation trees ET_C is depicted in Figure 1. The boolean value $Val(ET_C)$ of ET_C is the boolean value of the computation tree of $\mathcal{M}$ from C.

Remark 1. $\mathcal{M}$ accepts α iff $Val(ET_{C_\alpha}) = 1$.

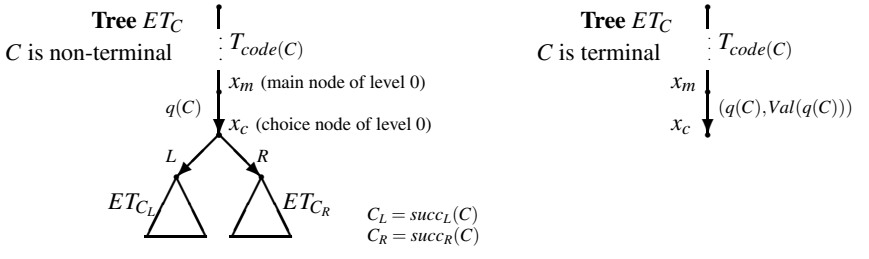


Fig. 1. Structure of the *Act*-labeled emulation tree ET_C for a good TM configuration C

Let T_{Codes} be the tree encoding of *Codes*, i.e. the unique *deterministic* finite *Act*-labeled tree such that the set of sequences of edge-labels associated with all maximal paths from the root is exactly *Codes*. For all $q \in Q$ and boolean value $b \in \{0,1\}$, let $Choices_{q,b}$ be the non-empty finite set of pairs $\langle (q_1,b_1),(q_2,b_2)\rangle$ such that $q_1, q_2 \in Q$, $b_1, b_2 \in \{0,1\}$, and $b_1 \vee b_2 = b$ if q is an existential state, and $b_1 \wedge b_2 = b$ otherwise.

Definition 3. [*Valuation trees*] *For all $q \in Q$ and $b \in \{0,1\}$, the Act-labeled* (q,b)-valuation tree of $\mathcal{M}$, *denoted by* $VT_{q,b}$, *is the Act-labeled* infinite *tree satisfying the following.* $VT_{q,b}$ *is obtained from* T_{Codes} *by adding for each leaf* y_m (main node of level 0) *of* T_{Codes} *exactly* $|Choices_{q,b}|$ *edges from* y_m *labeled by q and one edge from* y_m *to a* leaf *node labeled by* (q,b). *Moreover, for each* $\langle (q_L,b_L),(q_R,b_R)\rangle \in Choices_{q,b}$, *one of these new edges labeled by q leads to a node* y_c (choice node of level 0) *so that: (1)* y_c *has two children* y_L *and* y_R, *(2) the edge from* y_c *to* y_L (resp., *to* y_R) *is labeled by L* (resp., *R*), *and (3) the subtree rooted at* y_L (resp., y_R) *is isomorphic to* VT_{q_L,b_L} (resp., VT_{q_R,b_R}). *Note that the subtrees rooted at the main nodes of level* 0 *are isomorphic. The structure of* $VT_{q,b}$ *is depicted in Figure* 2.

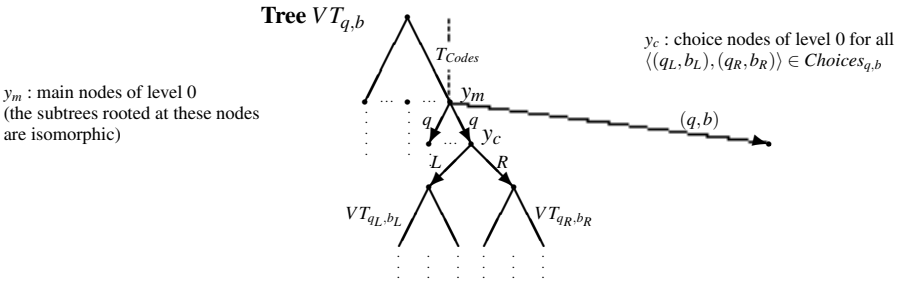


Fig. 2. Structure of the infinite Act-labeled valuation tree $VT_{q,b}$

Lemma 1. *Fix a good TM configuration C, a state $q \in Q$, and $b \in \{0,1\}$. Then, the root of ET_C is simulated by the root of $VT_{q,b}$* iff *$q = q(C)$ and $b = Val(ET_C)$.*

Lemma 1 can be easily proved by structural induction over the (finite) tree ET_C. By Remark 1 and Lemma 1, we obtain the following result.

Lemma 2. *$\mathcal{M}$ accepts α iff the root of ET_{C_α} is simulated by the root of $VT_{q(C_\alpha),1}$.*

3.1 EXPTIME-Hardness of Problem 1

In the case of Problem 1, the set Act of actions is given by

$$Act = Q \cup (Q \times \{0,1\}) \cup \{L,R\} \cup A \cup (A \times Q) \cup \{\lambda_0, \lambda_1\}$$

The code of each TM configuration $C = C(1), \ldots, C(n)$ is the word over Act of length $3n$ defined as:

$$code(C) = \lambda_0, C(1), \lambda_1, \ldots, \lambda_0, C(n), \lambda_1$$

Note that each symbol $u \in A \cup (A \times Q)$ is encoded by the word λ_0, u, λ_1. Let K be the size of $A \cup (A \times Q)$. We fix an ordering $\{a_1, \ldots, a_K\}$ of the elements in $A \cup (A \times Q)$ and we associate to each a_i the timestamp $\tau(a_i)$ given by i.

We construct a strongly timed-deterministic TA $\mathcal{T}_{em}$ over Act and a nondeterministic FS $\mathcal{A}_{val}$ over Act of sizes polynomial in the sizes of the fixed TM $\mathcal{M}$ and input α so that: the unwinding of $[\![\mathcal{T}_{em}]\!]_{abs}$ from its initial state is the emulation tree ET_{C_α} of Definition 2, and the unwinding of $\mathcal{A}_{val}$ from its initial state is the valuation tree $VT_{q(C_\alpha),1}$ of Definition 3. By Lemma 2 it follows that $\mathcal{M}$ accepts α iff $[\![\mathcal{T}_{em}]\!]_{abs} \preceq \mathcal{A}_{val}$. Hence, EXPTIME-hardness of Problem 1 follows.

Theorem 2. *Given a TA $\mathcal{T}$ and a nondeterministic FS $\mathcal{A}$, checking whether $[\![\mathcal{T}]\!]_{abs} \preceq \mathcal{A}$ is EXPTIME-hard, even if the TA $\mathcal{T}$ is assumed to be strongly timed-deterministic.*

In the rest of this subsection, we illustrate the construction of $\mathcal{T}_{em}$ (the construction of $\mathcal{A}_{val}$ is an easy task). Note that for a TM configuration $C = C(1), \ldots, C(n)$, the 'value' u_i of the i-th symbol of the left (resp., right) successor of C is completely determined by the values $C(i-1)$, $C(i)$ and $C(i+1)$ (taking $C(i+1)$ for $i = n$ and $C(i-1)$ for $i = 1$

to be some special symbol, say $\perp$). We denote by $next_L(C(i-1),C(i),C(i+1)$ (resp., $next_R(C(i-1),C(i),C(i+1))$ our expectation for u_i (these functions can be trivially obtained from the transition function δ of the fixed TM $\mathcal{M}$).

$\mathcal{T}_{em}$ uses $n+1$ clocks $x_0,x_1,\ldots,x_n$ in order to ensure a correct emulation of the evolution of $\mathcal{M}$. Clock x_0 is reset on generating the special action λ_1 (and only in this circumstance). On generating the special action λ_0, we require $(x_0 = 0)$ to hold, on generating $u \in A \cup (A \times Q)$, we require $(x_0 = \tau(u))$ to hold, and on generating λ_1, we require $(x_0 = K)$ to hold. This ensures that the durations of the consecutive three steps at which the code λ_0,u,λ_1 of $u \in A \cup (A \times Q)$ is generated are 0, $\tau(u)$, and $K - \tau(u)$, respectively. Hence, the overall duration of these steps is exactly K (independent on the specific action $u \in A \cup (A \times Q)$). Moreover, the overall duration of the sequence of steps at which a code $code(C)$ is generated is exactly nK. Furthermore, whenever an action $u \in A \cup (A \times Q)$ is generated from a location associated with the i-th symbol of a TM configuration C, clock x_i is reset (and only in this circumstance). This ensures that when the special action λ_0, associated with the i-th symbol of the next generated TM configuration C_{dir} with $dir \in \{L,R\}$, has to be generated, the following holds: the value of clock x_i is exactly $nK - \tau(C(i))$ and (assuming $i < n$) the value of clock x_{i+1} is exactly $(n-1)K - \tau(C(i+1))$. Thus, $\mathcal{T}_{em}$ will time-deterministically move (by taking a transition whose action is λ_0 and whose clock constraint is $x_0 = 0 \wedge x_i = nK - \tau(C(i)) \wedge x_{i+1} = (n-1)K - \tau(C(i+1))$ to a location of the form $p = (i,C(i-1),C(i),C(i+1),dir,\ldots)$, where the 'value' $C(i-1)$ is 'transmitted' from the previous location. At this point, $\mathcal{T}_{em}$ has all the information $(C(i-1),C(i),C(i+1)$, and $dir \in \{L,R\})$ to determine the i-th symbol of C_{dir}. Thus, there is exactly one transition from location p of the form $(p,u,x_0 = \tau(u),\{x_i\},p')$, where $u = next_{dir}(C(i-1),C(i),C(i+1))$.

3.2 EXPTIME-Hardness of Problem 2

In the case of Problem 2, the set Act of actions is given by

$$Act = Q \cup (Q \times \{0,1\}) \cup \{L,R\} \cup \big(\{1,\widetilde{1},\ldots,n,\widetilde{n}\} \times (A \cup (A \times Q))\big)$$

where for each $i \in \{1,\ldots,n\}$, $\widetilde{i}$ denotes a fresh copy of i. The meaning of these symbols will be explained later. The code of a TM configuration $C = C(1),\ldots,C(n)$ is now a word of length $2n$ given by

$$code(C) = (\widetilde{1},C(1)),(1,C(1)),\ldots,(\widetilde{n},C(n)),(n,C(n))$$

We construct a nondeterministic FS $\mathcal{B}_{val}$ over Act and a *synchronization-free* NFS $\mathcal{S}_{SF}$ over Act (whose components are deterministic and with pairwise disjoint alphabets of actions) of sizes polynomial in the size of the fixed TM $\mathcal{M}$ and input α such that $\mathcal{M}$ accepts α iff $[\![\mathcal{S}_{SF}]\!] \preceq \mathcal{B}_{val}$. Hence, EXPTIME-hardness of Problem 2 follows.

We note that a synchronization-free NFS of size polynomial in the size of $\mathcal{M}$ and α cannot faithfully emulate the evolution of $\mathcal{M}$ over the input α. In order to cope with this problem, we first define suitable extensions $Ext\_ET_{C_\alpha}$ and $Ext\_VT_{q(C_\alpha),1}$ of the emulation tree ET_{C_α} (Definition 2) and valuation tree $VT_{q(C_\alpha),1}$ (Definition 3), respectively, in such a way that the result of Lemma 2 holds even for these extensions. Then, we show that we can construct $\mathcal{B}_{val}$ and $\mathcal{S}_{SF}$ in such a way to ensure that the unwinding of $[\![\mathcal{S}_{SF}]\!]$

from its initial state is the extended emulation tree $Ext\_ET_{C_\alpha}$, and the unwinding of $\mathcal{B}_{val}$ from its initial state is the extended valuation tree $Ext\_VT_{q(C_\alpha),1}$.

Extended emulation trees and Extended valuation trees.

Definition 4. [*Extended emulation trees*] *For each* good *TM configuration C, an* extended emulation tree of $\mathcal{M}$ from C *is a* (*possibly infinite*) *Act-labeled tree Ext_ET_C which* extends *the emulation tree ET_C* (*Definition 2*) *in such a way that the following conditions are inductively satisfied. There is* exactly one partial path from the root which is labeled by some code in *Codes (note that by Definition 2, this path coincides with the partial path from the root to the main node of level* 0 *of ET_C). Moreover, each* new *edge from the main node of level 0 is labeled by an action in $Act \setminus (Q \cup (Q \times \{0,1\}))$ and:*

- C is terminal*: each* new *edge from the unique leaf of ET_C is labeled by an action in $Act \setminus \{L,R\}$.*
- C is not terminal*: each* new *edge from the choice node x_c of level* 0 *of ET_C is labeled by an action in $Act \setminus \{L,R\}$. Moreover, let x_L and x_R be the children of x_c in ET_C (recall that the subtrees rooted at x_L and x_R in ET_C correspond to $ET_{succ_L(C)}$ and $ET_{succ_R(C)}$, respectively). Then, we require that the subtrees rooted at x_L and x_R in Ext_ET_C are extended emulation trees of $\mathcal{M}$ from $succ_L(C)$ and $succ_R(C)$, respectively.*

Note that for a given (good) TM configuration C, there can be many extended emulation trees of $\mathcal{M}$ from C, and the definition above specifies only some properties that must be satisfied by them. We denote by $Ext(ET_C)$ the nonempty set of extended emulation trees of $\mathcal{M}$ from C. An *extended code* is a word over Act of the form $code \cdot u \cdot dir$, where $code \in Codes$, $u \in Q \cup (Q \times \{0,1\})$, and $dir \in \{L,R\}$. Let us consider the infinite Act-labeled valuation trees $VT_{q,b}$ of Definition 3. A node y of $VT_{q,b}$ is called *starting node* iff either y is the root or y is the child of some choice node (note that the subtree rooted at a child of a choice node corresponds to some valuation tree $VT_{q',b'}$). We extend $VT_{q,b}$ as follows, where T_{full} denotes the Act-labeled infinite tree obtained as the unwinding of the deterministic FS having a unique state and for each $u \in Act$, a self-loop labeled by u.

Definition 5. *[Extended valuation trees] For all $q \in Q$ and $b \in \{0,1\}$, the infinite Act-labeled* extended valuation tree *$Ext\_VT_{q,b}$ is obtained from $VT_{q,b}$ as follows. For each node y of $VT_{q,b}$, let y_0 be the first ancestor of y which is a starting node (note that y_0 may be y), and let $w_{y_0,y}$ be the word over Act labeling the partial path from y_0 to y. Note that $w_{y_0,y}$ has length at most $2n+1$ and is the* proper *prefix of some extended code. Then, for each $u \in Act$ such that the word $w_{y,y_0} \cdot u$ is* not *the prefix of any extended code, we add an edge labeled by u from y to the root of a tree isomorphic to T_{full}.*

The following lemma is the variant of Lemma 1 for extended emulation trees and extended valuation trees.

Lemma 3. *Fix a good TM configuration C, a state $q \in Q$, and $b \in \{0,1\}$. Then, for each extended emulation tree $Ext\_ET_C \in Ext(ET_C)$, the root of Ext_ET_C is simulated by the root of the extended valuation tree $Ext\_VT_{q,b}$* iff *$q = q(C)$ and $b = Val(ET_C)$.*

***Constructions of the* FS $\mathcal{B}_{val}$ *and the synchronization-free* NFS $\mathcal{S}_{SF}$.**

Lemma 4. *One can construct a nondeterministic* FS $\mathcal{B}_{val}$ *of size* polynomial *in the sizes of* $\mathcal{M}$ *and* α *such that the unwinding of* $\mathcal{B}_{val}$ *from its initial state is* $Ext\_VT_{q(C_\alpha),1}$.

Lemma 5. *One can construct a synchronization-free* NFS $\mathcal{S}_{SF}$ *(whose components are deterministic and with pairwise disjoint alphabets of actions) of size* polynomial *in the sizes of* $\mathcal{M}$ *and* α *such that the unwinding of* $[\![\mathcal{S}_{SF}]\!]$ *from its initial state is in* $Ext(ET_{C_\alpha})$.

Below, we prove Lemmata 4 and 5. By Remark 1 and Lemmata 3, 4, and 5, it follows that $\mathcal{M}$ accepts α iff $[\![\mathcal{S}_{SF}]\!] \preceq \mathcal{B}_{val}$, where $\mathcal{B}_{val}$ is the FS of Lemma 4 and $\mathcal{S}_{SF}$ is the NFS of Lemma 5. Hence, we obtain the desired result.

Theorem 3. *Given a* FS $\mathcal{A}$ *and a synchronization-free non-flat system* $\mathcal{S}$*, checking whether* $[\![\mathcal{S}]\!] \preceq \mathcal{A}$ *is* EXPTIME*-hard, even if the components of* $\mathcal{S}$ *are assumed to be deterministic and their alphabets are assumed to be pairwise disjoint.*

Proof of Lemma 4. We need some additional definition. Let $0 \leq i < 2n$, $last_\perp \in \{\perp\} \cup (\{1, \widetilde{1}, \ldots, n, \widetilde{n}\} \times (A \cup (A \times Q)))$ ($\perp$ is for undefined), and $f \in \{yes, no\}$ such that $last_\perp = \perp$ iff $i = 0$. A $(i, last_\perp, f)$-word is a *proper* prefix w_p of a code in *Codes* such that $|w_p| = i$, $last_\perp$ is the last symbol of w_p if $i > 0$, and $f = yes$ iff w_p contains some occurrence of a symbol in $\{\widetilde{1}, \ldots, \widetilde{n}\} \times (A \times Q)$. For each $u \in Act$, we consider the predicate $Prefix(i, last_\perp, f, u)$ which holds iff there exists a $(i, last_\perp, f)$-word w_p such that $w_p \cdot u$ is the prefix of some code in *Codes*. Note that by definition of *Codes*, the satisfaction of $Prefix(i, last_\perp, f, u)$ is independent on what representative is chosen in the set of $(i, last_\perp, f)$-words, i.e., for all $(i, last_\perp, f)$-words w_p and w'_p, it holds that $w_p \cdot u$ is the prefix of some code in *Codes* iff $w'_p \cdot u$ is the prefix of some code in *Codes*.

The FS $\mathcal{B}_{val} = \langle Act, P_{val}, p^0_{val}, \Delta_{val} \rangle$ satisfying the statement of Lemma 4 is defined as follows. The set of states is $P_{val} = \{p_{full}\} \cup P_{cod} \cup P_{main} \cup P_{choice} \cup \{p_\#\}$, where:

- From state p_{full} there are self-loops on all actions from Act. Thus, the unwinding of $\mathcal{B}_{val}$ from p_{full} corresponds to T_{full} (see Definition 5).
- P_{cod} consists of states of the form $p_{cod} = (q, b, i, last_\perp, f)$, where $(q, b) \in Q \times \{0, 1\}$, $1 \leq i \leq 2n$, $last_\perp \in \{\perp\} \cup (\{1, \widetilde{1}, \ldots, n, \widetilde{n}\} \times (A \cup (A \times Q)))$, and $f \in \{yes, no\}$, where $last_\perp = \perp$ iff $i = 1$. The states in P_{cod} are used to generate all the codes in *Codes*. Intuitively, (q, b) is the currently processed pair TM state/ boolean value, i is the currently processed position of a *code*. Moreover, $last_\perp$ keeps track of the last generated symbol if $i > 1$, and the flag f is used to keep track whether a symbol of the form $(\widetilde{j}, u)$ with $u \in A \times Q$ has already been generated in the previous $i - 1$ steps. From state p_{cod}, $\mathcal{B}_{val}$ generates all the actions $u \in Act$ in such a way that the following holds. If $Prefix(i - 1, last_\perp, f, u)$ holds (this means that the word $w_p \cdot u$ is the prefix of some code in *Codes*, where w_p is the $(i, last_\perp, f)$-word generated in the previous $i - 1$ steps), then $\mathcal{C}_{val}$ generates the letter u and moves to a state in P_{cod} of the form $(q, b, i + 1, u, f')$ if $i < n$ and to the main state (q, b) (see below) otherwise. If instead $Prefix(i - 1, last_\perp, f)$ does not hold, then $\mathcal{B}_{val}$ generates the action u and moves to the state p_{full}.
- $P_{main} = Q \times \{0, 1\}$. States in P_{main} are associated with the main nodes of $Ext\_VT_{q(C_\alpha),1}$ (corresponding to the main nodes of $VT_{q(C_\alpha),1}$).

- $P_{choice} = (Q \times \{0,1\}) \times (Q \times \{0,1\})$. States in P_{choice} are associated with the choice nodes of $Ext\_VT_{q(C_\alpha),1}$ (corresponding to the choice nodes of $VT_{q(C_\alpha),1}$).
- State $p_\#$ is associated with the nodes of $Ext\_VT_{q(C_\alpha),1}$ corresponding to the leaf nodes of $VT_{q(C_\alpha),1}$.

The initial state p^0_{val} is $(q(C_\alpha), 1, 1, \perp, no)$ and the transition relation Δ_{val} is defined as:

1. *Transitions from* p_{full}: for each $u \in Act$, we have the transition $p_{full} \xrightarrow{u} p_{full}$
2. *Transitions to generate the codes in Codes*: from each state $(q,b,i,last_\perp,f) \in P_{cod}$ and for each $u \in Act$, we have the following transition:
   - *Prefix*$(i-1, last_\perp, f, u)$ *does not hold*: $(q,b,i,last_\perp,f) \xrightarrow{u} p_{full}$
   - *Prefix*$(i-1, last_\perp, f, u)$ *holds* and $i < 2n$: $(q,b,i,last_\perp,f) \xrightarrow{u} (q,b,i+1,u,f')$ where $f' = yes$ if $u \in \{\widetilde{1},\ldots,\widetilde{n}\} \times (A \times Q)$, and $f' = f$ otherwise.
   - *Prefix*$(i-1, last_\perp, f, u)$ *holds* and $i = 2n$: $(q,b,i,last_\perp,f) \xrightarrow{u} (q,b)$
3. *Transitions from main states* $(q,b) \in Q \times \{0,1\}$:
   - $(q,b) \xrightarrow{q} ((q_1,b_1),(q_2,b_2))$ for each $\langle (q_1,b_1),(q_2,b_2)\rangle \in Choices_{q,b}$.
   - $(q,b) \xrightarrow{(q,b)} p_\#$
   - $(q,b) \xrightarrow{u} p_{full}$ for all $u \in Act \setminus (Q \cup (Q \times \{0,1\}))$.
4. *Transitions from choice states* $((q_1,b_1),(q_2,b_2)) \in (Q \times \{0,1\}) \times (Q \times \{0,1\})$:
   - $((q_1,b_1),(q_2,b_2)) \xrightarrow{L} (q_1,b_1,1,\perp,no)$
   - $((q_1,b_1),(q_2,b_2)) \xrightarrow{R} (q_2,b_2,1,\perp,no)$
   - $((q_1,b_1),(q_2,b_2)) \xrightarrow{u} p_{full}$ for each $u \in Act \setminus \{L,R\}$.
5. *Transitions from state* $p_\#$: $p_\# \xrightarrow{u} p_{full}$ for each $u \in Act \setminus \{L,R\}$.

Correctness of construction easily follows.

Proof of Lemma 5. The synchronization-free NFS $\mathcal{S}_{SF}$ satisfying the statement of Lemma 5 is given by $\mathcal{S}_{SF} = \langle Cell_1,\ldots,Cell_n,Control\rangle_{SF}$. Intuitively, $Cell_j$ $(1 \le j \le n)$ keeps track by its finite control of the j-th symbol of a TM configuration, and it can generate only the actions of the form (j,u) (where $u \in A \cup (A \times Q)$). Note that the action (j,u) corresponds to the 2nd symbol in the j-th pair $(\widetilde{j},u),(j,u)$ of the code of some TM configuration. *Control* is instead used to model the control unit of $\mathcal{M}$, and it can generate only the actions in $Act \setminus (\{1,\ldots,n\} \times (A \cup (A \times Q)))$. After having 'correctly' generated the code of a TM configuration C, $Cell_j$ is in state $C(j)$ and *Control* is in a state which keeps track of the position i of the tape head of C together with the i-th symbol of C. Assume that C is not terminal. In order to generate for each $dir \in \{L,R\}$, the j-th pair $(\widetilde{j},u),(j,u)$ $(1 \le j \le n)$ of the code of the dir-successor $succ_{dir}(C)$ of C, the $\mathcal{S}_{SF}$-components behave as follows. Assume that $j \neq i$ and j is *not* the position of the tape head in $succ_{dir}(C)$ (the other cases are similar). Since *Control* keeps track of the pair $(i,C(i))$ and the current position j, it can check (by using the transition function δ of the TM $\mathcal{M}$) whether this condition is satisfied or not. Note that in this case, the j-th symbol of $succ_{dir}(C)$ coincides with the j-th symbol of C, i.e. $u = C(j)$, and, additionally $u \in A$. Then, *Control* guesses a pair $(\widetilde{j},u')$ with $u' \in A$ and generates it. Component $Cell_j$, which is in state $C(j)$, will be able to generate in the next step the matching pair (j,u') iff $u' = C(j)$. In this way, the $\mathcal{S}_{SF}$-components ensure that for each choice

$dir \in \{L,R\}$, exactly one code in *Codes* will be generated, and this code is precisely the encoding of $succ_{dir}(C)$. The crucial point is that even if other words of length $2n$ will be generated (due to all the possible interleaving of the individual and asynchronous computational steps of the single components), exactly one of these words of length $2n$ will be a code in *Codes*. The $\mathcal{S}_{SF}$-components are formally defined below.

$Cell_j = \langle \{j\} \times (A \cup (A \times Q)), \{p_j^0\} \cup A \cup (A \times Q), p_j^0, \Delta_j \rangle$, where Δ_j is defined as:

1. $p_j^0 \xrightarrow{(j,C_\alpha(j))} C_\alpha(j)$
2. $a \xrightarrow{(j,a)} a$ and $a \xrightarrow{(j,a,q)} (a,q)$ for all $a \in A$ and $q \in Q$
3. $(a,q) \xrightarrow{(j,a')} a'$ for all $a,a' \in A$ and $q \in Q$

The first transition is used to generate the 2nd symbol of the j-th pair of the code of the initial TM configuration C_α. Transitions of type 2 (resp., 3) are used to generate the 2nd symbol of the j-th pair of the code of the *next* TM configuration when the tape head is at position $i \neq j$ (resp., $i = j$). Note that the source state of transitions of type 2–3 represents the j-th symbol of the current TM configuration.

$Control = \langle Act \setminus (\{1,\ldots,n\} \times (A \cup (A \times Q))), P, 1, \Delta \rangle$ is defined as follows. The set of states is given by $P = \{p_{fin}\} \cup P_{init} \cup P_{conf} \cup P_{main} \cup P_{choice}$, where:

- p_{fin} is a state with no outgoing transitions.
- $P_{init} = \{1,\ldots,n\}$. States in P_{init} are used to generate the code of C_α, and 1 is the initial state.
- P_{conf} consists of states of the form $p_{conf} = (j,(i,a,q),a_\perp,dir)$, where $1 \leq j,i \leq n$, $a \in A$, $q \in Q \setminus \{q_{acc},q_{rej}\}$, $a_\perp \in A \cup \{\perp\}$ ($\perp$ is for undefined), and $dir \in \{L,R\}$. Intuitively, i is the position of the tape head for the current non-terminal TM configuration C and $(a,q) = C(i)$. Let $\delta(q,a) = \langle (q_L,a_L,\theta_L),(q_R,a_R,\theta_R) \rangle$. Then, from state p_{conf}, *Control* guesses and generates an action of the form $(\widetilde{j},u)$ with the constraint that $u = a_{dir}$ if $j = i$, $u = (a',q_{dir})$ for some $a' \in A$ if $j = i + \theta_{dir}$, and $u \in A$ if $j \notin \{i, i+\theta_{dir}\}$ (note that a_{dir} is the i-th symbol of $succ_{dir}(C)$ and $i+\theta_{dir}$ is the position of the tape head in $succ_{dir}(C)$). If the guess is correct (i.e., u is the j-th symbol of $succ_{dir}(C)$), then $Cell_j$ is able to generate the matching action (j,u) in the next step. In this case, $a_\perp$ is the content of the $succ_{dir}(C)$-cell pointed by the tape head if the position of this cell is smaller than j, and $a_\perp = \perp$ otherwise.
- P_{main} consists of states of the form $p_{main} = (main,(i,a,q))$, where $1 \leq i \leq n$, $a \in A$, and $q \in Q$. Intuitively, i represents the position of the tape head for the new generated TM configuration C and $(a,q) = C(i)$. From state p_{main}, *Control* moves to the choice state $(choice,(i,a,q))$ (see below) by generating the action q if $q \notin \{q_{acc},q_{rej}\}$, and to the state p_{fin} by generating the action $(q,Val(q))$ otherwise.
- P_{choice} consists of states of the form $(choice,(i,a,q))$, where (i,a,q) has the same meaning as above. From these states, *Control* generates the two actions L and R.

The transition relation Δ of *Control* is defined as follows.

1. *Initialization (transitions to generate the code of* C_α*)*:
   - $1 \xrightarrow{(\widetilde{1},C_\alpha(1))} 2,\ldots,n \xrightarrow{(\widetilde{n},C_\alpha(n))} (main,(1,\alpha(0),q_0))$

2. *Transitions to generate the next TM configuration*: from each state $(j,(i,a,q),a_\perp,dir) \in P_{conf}$ with $\delta(q,a) = \langle (q_L,a_L,\theta_L),(q_R,a_R,\theta_R)\rangle$ and $1 \leq i+\theta_{dir} \leq n$, and for each $a' \in A$ such that $a' = a_{dir}$ if $j = i$, we have the following transitions:
   - $j \neq i+\theta_{dir}$, $j < n$: $(j,(i,a,q),a_\perp,dir) \xrightarrow{(\widetilde{j},a')} (j+1,(i,a,q),a_\perp,dir)$
   - $j \neq i+\theta_{dir}$, $j = n$, $a_\perp \neq \perp$: $(n,(i,a,q),a_\perp,dir) \xrightarrow{(\widetilde{n},a')} (main,(i+\theta_{dir},a_\perp,q_{dir}))$
   - $j = i+\theta_{dir}$, $j < n$: $(j,(i,a,q),a_\perp,dir) \xrightarrow{(\widetilde{j},a',q_{dir})} (j+1,(i,a,q),a',dir)$
   - $j = i+\theta_{dir} = n$: $(n,(i,a,q),a_\perp,dir) \xrightarrow{(\widetilde{n},a',q_{dir})} (main,(n,a',q_{dir}))$
3. *Transitions from main states* $(main,(i,a,q)) \in P_{main}$:
   - $q \notin \{q_{acc},q_{rej}\}$: $(main,(i,a,q)) \xrightarrow{q} (choice,(i,a,q))$
   - $q \in \{q_{acc},q_{rej}\}$: $(main,(i,a,q)) \xrightarrow{(q,Val(q))} p_{fin}$
4. *Transitions from choice states* $(choice,(i,a,q)) \in P_{choice}$:
   - $(choice,(i,a,q)) \xrightarrow{L} (1,(i,a,q),\perp,L)$ and $(choice,(i,a,q)) \xrightarrow{R} (1,(i,a,q),\perp,R)$

Now, we prove that the construction is correct. Let C be a *non-terminal good* TM configuration and $dir \in \{L,R\}$. The *starting* (C,dir)*-state* is the state of $\mathcal{S}_{SF}$ in which component $Cell_j$ is in state $C(j)$ and $Control$ is in the state $(1,(i,a,q),\perp,dir) \in P_{conf}$, where i is the position of the tape head in C and $(a,q) = C(i)$. Moreover, for each $dir \in \{L,R\}$, the starting $(\perp,dir)$-state is the initial state of $\mathcal{S}_{SF}$. By construction, the following result easily follows.

Lemma 6. *Fix a starting* $(C_\perp,dir)$*-state* p_{start}*, and let* $C = C_\alpha$ *if* $C_\perp = \perp$*, and* $C = succ_{dir}(C_\perp)$ *otherwise. Then, there is a* unique *path* π *of* $\mathcal{S}_{SF}$ *starting from* p_{start} *labeled by a code* $\in Codes$*. Moreover,* $code = code(C)$ *and there is exactly one transition* $p \xrightarrow{u} p'$ *from the last state* p *of* π *labeled by an action* $u \in Q \cup (Q \times \{0,1\})$*. Furthermore, the action* u *and state* p' *satisfies the following:*

- *C is terminal: $u = (q(C),Val(q(C)))$ and there is no transition outgoing from p' labeled by an action in $\{L,R\}$.*
- *C is not terminal: $u = q(C)$ and there are exactly two transitions from state p' labeled by actions in $\{L,R\}$. Moreover, one, labeled by L, leads to the starting (C,L)-state, and the other one, labeled by R, leads to the starting (C,R)-state.*

By Definition 4 and Lemma 6, we obtain the desired result.

Corollary 1 (Correctness). *The unwinding of* $\mathcal{S}_{SF}$ *from its initial state is in* $Ext(ET_{C_\alpha})$.

Note that the components of $\mathcal{S}_{SF}$ are deterministic and with pairwise disjoint alphabets of actions. Moreover, the size of $\mathcal{S}_{SF}$ is polynomial in the sizes of the TM $\mathcal{M}$ and input α. Thus, by the above corollary, Lemma 5 follows.

4 Additional Hardness Results

We can show that the deterministic versions of Problems 1 and 2 are PSPACE-hard by polynomial-time reductions from the word problem for *linearly-bounded deterministic* Turing Machines. The proposed constructions can be seen as a simplification of those illustrated in the previous section.

Theorem 4. *The deterministic versions of Problems 1 and 2 are* PSPACE*-hard, and for Problems 2,* PSPACE*-hardness holds even if the components of the* synchronization-free *non-flat system are assumed to be deterministic and with pairwise disjoint alphabets.*

The rest of this section is devoted to the proof of the following theorem.

Theorem 5. *For any relation $\trianglelefteq$ on LTS lying between trace containment and bisimulation equivalence, checking whether $[\![\mathcal{S}]\!] \trianglelefteq \mathcal{A}$ for a given synchronous* NFS *$\mathcal{S}$ and a* FS *$\mathcal{A}$ is* PSPACE*-hard even if $\mathcal{A}$ and the $\mathcal{S}$-components are assumed to be deterministic.*

Proof of Theorem 5. By a polynomial-time reduction from the acceptance problem for *non-halting linearly-bounded deterministic* Turing Machines (TM). Fix such a TM machine $\mathcal{M} = \langle A, Q, q_0, \delta, \{q_{acc}\}\rangle$, where A, Q, q_0, q_{acc} (with $q_0 \neq q_{acc}$) are as for alternating Turing Machines, and $\delta : Q \times A \to (Q \times A \times \{+1,-1\})$ is the transition function, where $+1$ (resp., -1) denotes a right (resp., left) tape head move. Fix an input $\alpha \in A^*$ and let $n = |\alpha|$. Since $\mathcal{M}$ is linearly bounded, we can assume that a TM configuration (of $\mathcal{M}$ over α) is a word $C = \beta_1,(a,q),\beta_2 \in A^* \cdot (A \times Q) \cdot A^*$ of length exactly n. $\mathcal{M}$ accepts α iff the unique (infinite) computation of $\mathcal{M}$ over α visits an accepting configuration. W.l.o.g. we assume that the alphabet A contains a special symbol, say #, such that if the computation of $\mathcal{M}$ over α visits an accepting configuration C_{acc}, then C_{acc} is #*-homogeneous*, i.e. the content of each cell of C_{acc} is the special symbol #.

Preliminary step: encoding of acceptance. Let $Act = (\{1,\ldots,n\} \times (A \cup (A \times Q))) \cup \{\flat\}$, where $\flat$ is a special action. For each TM configuration C, $code(C)$ is the word over $Act \setminus \{\flat\}$ given by $(1,C(1)),\ldots,(n,C(n))$. Let *Codes* be the finite set of these codes and T_{Codes} be the deterministic tree encoding of *Codes* (as defined in Section 3).

Definition 6 (Valuation tree). *The* valuation tree T_{val} *is the infinite $Act \setminus \{\flat\}$-labeled tree obtained as the limit of the sequence of finite trees $(T^k_{Codes})_{k\in\mathbb{N}}$, where: $T^0_{Codes} = T_{Codes}$ and T^{k+1}_{Codes} results from rooting a fresh copy of T_{Codes} at each leaf of T^k_{Codes}.*

Note that T_{val} is a deterministic and all its maximal paths from the root are infinite and labeled by concatenations of codes of TM configurations. The *special path* of T_{val} is the unique maximal path from the root whose sequence of labels $code(C_1) \cdot code(C_2) \ldots$ is such that $C_\alpha, C_1, C_2, \ldots$ is the computation of $\mathcal{M}$ over α.

Definition 7 (Emulation tree). *The* deterministic *Act-labeled* emulation tree T_{em} *is defined as follows. Let π be the special path of T_{val} and let $code(C_1) \cdot code(C_2) \ldots$ be its sequence of labels. For each $i \geq 1$ such that C_i is an* accepting *#-homogeneous configuration, let $x_i \xrightarrow{u_i} y_i$ be the edge along π associated with the last symbol of $code(C_i)$. Then, T_{em} is obtained from T_{val} by adding for each of these edges $x_i \xrightarrow{u_i} y_i$ (if any), a new edge labeled by the special action $\flat$ from y_i to a new leaf node.*

Fact: if $\mathcal{M}$ does *not* accept α, then $T_{em} = T_{val}$. Otherwise, $Tr(T_{em}) \not\subseteq Tr(T_{val})$.

Final step. Theorem 5 directly follows from the fact above and the following two Lemmata. The proof of Lemma 7 is trivial and we omit it.

Lemma 7. *One can construct a* deterministic FS *$\mathcal{D}_{val}$ over Act of size* polynomial *in the sizes of $\mathcal{M}$ and α such that the unwinding of $\mathcal{D}_{val}$ from its initial state is T_{val}.*

Lemma 8. *One can construct a* synchronous NFS $\mathcal{S}_{em}$ *over Act (whose components are deterministic) of size* polynomial *in the sizes of* $\mathcal{M}$ *and* α *such that the unwinding of* $[\![\mathcal{S}_{em}]\!]$ *from its initial state is* T_{em}.

Sketched proof. $\mathcal{S}_{em}$ is a synchronous composition of $n+1$ components $Cell_1, \ldots, Cell_n$, and *Control*. Intuitively, $Cell_j$ $(1 \leq j \leq n)$ keeps track by its finite control of the j-th symbol of a TM configuration, and its alphabet is $\{\flat\} \cup (\{j\} \times (A \cup (A \times Q)))$. *Control* is instead used to model the control unit of $\mathcal{M}$, and its alphabet is *Act* (hence, *Control* participates in each transition of $\mathcal{S}_{em}$). After having 'correctly' generated a TM configuration C (this, intuitively, means that $\mathcal{S}_{sem}$ is emulating the computation of $\mathcal{M}$ over α, i.e., the computational path from the initial state to the current state of $\mathcal{S}_{em}$ corresponds to a prefix of the special path of T_{val}), $\mathcal{S}_{em}$ is in a *starting good* state s_{good} such that component $Cell_j$ is in state $C(j)$ and *Control* is in a good local state which keeps track of the position k of the tape head of C together with the k-th symbol of C.[2] From state s_{good}, $\mathcal{S}_{em}$ generates in n steps by 'computational nondeterminism' all the possible codes in *Codes* as follows. At the j-th step, *Control* guesses an action of the form (j,u) and generates it by binary synchronization with $Cell_j$. Note that since $Cell_j$ is in state $C(j)$ and *Control* keeps track of k and $C(k)$, either $Cell_j$ or *Control* is able to detect whether for the communication action (j,u), u is *not* the j-th symbol of the successor $succ(C)$ of C. If it is the case, and $Cell_j$ (resp., *Control*) has detected it, then $Cell_j$ (resp., *Control*) will move to the local bad state *bad* (resp., to a local bad state $p_{bad,j'}$ associated with the step $j' = (j+1) \bmod n$). From state *bad*, $Cell_j$ remaining in *bad* may generate all the actions of its alphabet except $\flat$. From the bad state $p_{bad,j'}$, *Control* can generate only the actions of the form (j',u) and move to a local bad state associated with the step $(j'+1) \bmod n$. If instead for the generated action (j,u), u is the j-th symbol of $succ(C)$, then *Control* will move to a good local state associated with the step $j' = (j+1) \bmod n$ and $Cell_j$ will move to the good local state $[succ(C)](j)$. Thus, after having generated the code of $succ(C)$, $\mathcal{S}_{em}$ will be in the starting good state s'_{good} associated with $succ(C)$. From s'_{good}, $\mathcal{S}_{em}$ can generate the special action $\flat$ (by synchronization among all its components)[3] iff $succ(C)$ is an accepting #-homogeneous configuration. If the action $\flat$ is generated, then the target state has no outgoing transition. □

5 Conclusions

As future research, there is an interesting question left open: the exact complexity of bisimulation checking between a flat system and a non-flat system. Our contribution (Theorem 5) shows that the problem is PSPACE-hard even for synchronous composition without hiding. Note that the problem is in EXPTIME. We believe that filling this gap is a very difficult question. Simple settings are however tractable: Muscholl and Walukiewicz [13] have recently shown that bisimulation checking can be solved in NLOGSPACE when there is no synchronization and both the flat system and the non-flat system components are deterministic. It would be interesting to investigate the nondeterministic framework.

[2] Initially, $\mathcal{S}_{em}$ is in the starting good state associated with the initial TM configuration.

[3] Note that $\flat$ is in the alphabet of each component.

References

1. Alur, R., Dill, D.: A theory of timed automata. Theoretical Computer Science 126(2), 183–235 (1994)
2. Balcázar, J.L., Gabarró, J., Santha, M.: Deciding bisimilarity is p-complete. Formal Asp. Comput. 4(6A), 638–648 (1992)
3. Chandra, A.K., Kozen, D.C., Stockmeyer, L.J.: Alternation. Journal of the ACM 28(1), 114–133 (1981)
4. Kupferman, O., Harel, D., Vardi, M.Y.: On the complexity of verifying concurrent transition systems. In: Mazurkiewicz, A., Winkowski, J. (eds.) CONCUR 1997. LNCS, vol. 1243, pp. 258–272. Springer, Heidelberg (1997)
5. Groote, J.F., Moller, F.: Verification of parallel systems via decomposition. In: Cleaveland, W.R. (ed.) CONCUR 1992. LNCS, vol. 630, pp. 62–76. Springer, Heidelberg (1992)
6. Henzinger, M.R., Henzinger, T.A., Kopke, P.W.: Computing simulations on finite and infinite graphs. In: Proc. 36th FOCS, pp. 453–462. IEEE Computer Society, Los Alamitos (1995)
7. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Englewood Cliffs (1984)
8. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley, Reading (1979)
9. Jategaonkar, L., Meyer, A.R.: Deciding true concurrency equivalences on safe, finite nets. Theoretical Computer Science 154(1), 107–143 (1996)
10. Kučera, A., Jančar, P.: Equivalence-checking on infinite-state systems: Techniques and results. Theory and Practice of Logic Programming 6(3), 227–264 (2006)
11. Laroussinie, F., Schnoebelen, P.: The state explosion problem from trace to bisimulation equivalence. In: Tiuryn, J. (ed.) FOSSACS 2000. LNCS, vol. 1784, pp. 192–207. Springer, Heidelberg (2000)
12. Milner, R.: Communication and Concurrency. Prentice-Hall, Englewood Cliffs (1989)
13. Muscholl, A., Walukiewicz, I.: A lower bound on web services composition. In: Seidl, H. (ed.) FOSSACS 2007. LNCS, vol. 4423, pp. 274–286. Springer, Heidelberg (2007)
14. Rabinovich, A.M.: Complexity of equivalence problems for concurrent systems of finite agents. Information and Computation 139(2), 111–129 (1997)
15. Sawa, Z., Jančar, P.: Behavioural equivalences on finite-state systems are PTIME-hard. Computing and Informatics 24(5) (2005)
16. Sawa, Z., Jančar, P.: Hardness of equivalence checking for composed finite-state systems. Acta Informatica 46(3), 169–191 (2009)
17. Shukla, S.K., Hunt III, H.B., Rosenkrantz, D.J., Stearns, R.E.: On the complexity of relational problems for finite state processes. In: Meyer auf der Heide, F., Monien, B. (eds.) ICALP 1996. LNCS, vol. 1099, pp. 466–477. Springer, Heidelberg (1996)
18. Tasiran, S., Alur, R., Kurshan, R.P., Brayton, R.K.: Verifying abstractions of timed systems. In: Sassone, V., Montanari, U. (eds.) CONCUR 1996. LNCS, vol. 1119, pp. 546–562. Springer, Heidelberg (1996)
19. Čerāns, K.: Decidability of bisimulation equivalences for parallel timer processes. In: Probst, D.K., von Bochmann, G. (eds.) CAV 1992. LNCS, vol. 663, pp. 302–315. Springer, Heidelberg (1993)
20. Valmari, A., Kervinen, A.: Alphabet-based synchronisation is exponentially cheaper. In: Brim, L., Jančar, P., Křetínský, M., Kucera, A. (eds.) CONCUR 2002. LNCS, vol. 2421, pp. 161–176. Springer, Heidelberg (2002)
21. van Glabbeek, R.J.: The linear time-branching time spectrum. In: Baeten, J.C.M., Klop, J.W. (eds.) CONCUR 1990. LNCS, vol. 458, pp. 278–297. Springer, Heidelberg (1990)

A Quasipolynomial Cut-Elimination Procedure in Deep Inference via Atomic Flows and Threshold Formulae

Paola Bruscoli[1,⋆], Alessio Guglielmi[1,⋆⋆], Tom Gundersen[1,⋆⋆⋆], and Michel Parigot[2,†]

[1] University of Bath (UK) and LORIA & INRIA Nancy-Grand Est, France
[2] Laboratoire PPS, UMR 7126, CNRS & Université Paris 7, France

Abstract. Jeřábek showed in 2008 that cuts in propositional-logic deep-inference proofs can be eliminated in quasipolynomial time. The proof is an indirect one relying on a result of Atserias, Galesi and Pudlák about monotone sequent calculus and a correspondence between this system and cut-free deep-inference proofs. In this paper we give a direct proof of Jeřábek's result: we give a quasipolynomial-time cut-elimination procedure in propositional-logic deep inference. The main new ingredient is the use of a computational trace of deep-inference proofs called atomic flows, which are both very simple (they trace only structural rules and forget logical rules) and strong enough to faithfully represent the cut-elimination procedure.

1 Introduction

Deep inference is a deduction framework (see [8,4,2]), where deduction rules apply arbitrarily deep inside formulae, contrary to traditional proof systems like natural deduction and sequent calculus, where deduction rules deal only with their outermost structure. This greater freedom is both a source of immediate technical difficulty and the promise, in the long run, of new powerful proof-theoretic methods. A general methodology allows to design deep-inference deduction systems having more symmetries and finer structural properties than sequent calculus. For instance, cut and identity become really dual of each other,

⋆ Supported by EPSRC grant EP/E042805/1 *Complexity and Non-determinism in Deep Inference* and by an ANR *Senior Chaire d'Excellence* titled *Identity and Geometric Essence of Proofs*.

⋆⋆ Supported by EPSRC grant EP/E042805/1 *Complexity and Non-determinism in Deep Inference* and by an ANR *Senior Chaire d'Excellence* titled *Identity and Geometric Essence of Proofs*.

⋆⋆⋆ Supported by an *Overseas Research Studentship* and a *Research Studentship*, both of the University of Bath, and by an ANR *Senior Chaire d'Excellence* titled *Identity and Geometric Essence of Proofs*.

† Supported by *Project INFER—Theory and Application of Deep Inference* of the *Agence Nationale de la Recherche*.

E.M. Clarke and A. Voronkov (Eds.): LPAR-16, LNAI 6355, pp. 136–153, 2010.

whereas they are only morally so in sequent calculus, and all structural rules can be reduced to their atomic form, whereas this is false for contraction in sequent calculus.

All usual logics have deep-inference deduction systems enjoying cut elimination (see [7] for a complete overview). The traditional methods of cut elimination of sequent calculus can be adapted to a large extent to deep inference, despite having to cope with a higher generality. New methods are also achievable. The standard proof system for propositional classical logic in deep inference is system SKS [4,2]. Its cut elimination has been proved in several different ways [4,2,9].

Recently, Jeřábek showed that cut elimination in SKS proofs can be done in quasipolynomial time [10], i.e., in time $n^{O(log(n))}$. The result is surprising because all known cut-elimination methods for classical-logic proof systems require exponential time, in particular for Gentzen's sequent calculus. Jeřábek obtained his result by relying on a construction over threshold functions by Atserias, Galesi and Pudlák, in the monotone sequent calculus [1]. Note that, contrary to SKS, the monotone sequent calculus specifies a weaker logic than propositional logic because negation is not freely applicable.

The technique that Jeřábek adopts is indirect because cut elimination is performed over proofs in the sequent calculus, which are, in turn, related to deep-inference ones by polynomial simulations, originally studied in [3] and [5].

In this paper we give a direct proof of Jeřábek's result: that is, we give a quasipolynomial-time cut-elimination procedure in propositional-logic deep inference, which, in addition to being internal, has a strong computational flavour. This proof uses two ingredients:

1. an adaptation of Atserias-Galesi-Pudlák technique to deep inference, which slightly simplifies the technicalities associated with the use of threshold functions; in particular, the formulae and derivations that we adopt are simpler than those in [1];
2. a computational trace of deep-inference proofs called atomic flows, which are both very simple (they trace only structural rules and forget logical rules) and strong enough to faithfully represent cut elimination.

Atomic flows, which can be considered as specialised Buss flow graphs [6], play a major role in designing and controlling the cut elimination procedure presented in this paper. They contribute to the overall clarification of the procedure, by reducing our dependency on syntax. The techniques developed via atomic flows tolerate variations in the proof system specification. In fact, their geometric nature makes them largely independent of syntax, provided that certain linearity conditions are respected (and this is usually achievable in deep inference).

The paper is self-contained. Sections 2 and 3 are devoted, respectively, to the necessary background on deep inference and atomic flows. Threshold functions and formulae are introduced in Sect. 5.

We normalise proofs in two steps, each of which has a dedicated section in the paper:

1. We transform any given proof into what we call its 'simple form'. No use is made of threshold formulae and no significant proof complexity is introduced. This is presented in Sect. 4, which constitutes a good exercise on deep inference and atomic flows.
2. In Sect. 6, we show the cut elimination step, starting from proofs in simple form. Here, threshold formulae play a major role.

Section 7 concludes the paper with some comments on future research directions.

2 Propositional Logic in Deep Inference: The SKS System

Formulae and Contexts

Two *logical constants*, f (false) and t (true) and a countable set of *propositional letters*, denoted by p and q, are given. A primitive *negation* $\bar{\cdot}$ is defined on propositional letters: to each propositional letter p is associated its negation $\bar{p}$. *Atoms* are propositional letters and their negation; they are denoted a, b, c, d and e. Negation is extended to the set of atoms by defining $\bar{\bar{p}} = p$, for each negated propositional letter $\bar{p}$. Being in classical logic, one can always exchange an atom with its negation: at the level of atoms, it doesn't matter which one is the propositional letter or its negation.

Formulae, denoted by A, B, C and D, are freely built from logical constants and atoms using *disjunction* and *conjunction*. The disjunction and conjunction of two formulae A and B are denoted respectively $[A \vee B]$ and $(A \wedge B)$: the different brackets have the only purpose of improving legibility. We usually omit external brackets of formulae and sometimes we omit superfluous brackets under associativity. Negation can be extended to arbitrary formulae in an obvious way using De Morgan's laws, but we do not need it in this paper. We write $A \equiv B$ for literal equality of formulae.

We denote (*formula*) *contexts*, i.e., formulae with a hole, by $K\{\ \}$; for example, if $K\{a\}$ is $b \wedge [a \vee c]$, then $K\{\ \}$ is $b \wedge [\{\ \} \vee c]$, $K\{b\}$ is $b \wedge [b \vee c]$ and $K\{a \wedge d\}$ is $b \wedge [(a \wedge d) \vee c]$.

Derivations and Proofs

An (*inference*) *rule* ρ is an expression $\rho\,\dfrac{A}{B}$, where the formulae A and B are called *premiss* and *conclusion*, respectively. In deep inference, rules are applied in arbitrary contexts: an (*inference*) *step* corresponding to rule ρ is an expression $\rho\,\dfrac{K\{C\}}{K\{D\}}$, where $K\{\ \}$ is a context and $\rho\,\dfrac{C}{D}$ is an instance of $\rho\,\dfrac{A}{B}$.

A *derivation*, Φ, *from* A (*premiss*) *to* B (*conclusion*) is a chain of inference steps with A at the top and B at the bottom, and is usually indicated by $\Phi \| \mathcal{S}$ $\dfrac{A}{B}$, where $\mathcal{S}$ is the name of the deduction system or a set of inference rules (we might

omit Φ and $\mathcal{S}$); we also use the notation $\Phi : A \to B$. Sometimes we group $n \geq 0$ inference steps of the same rule ρ together into one step, and we label the step with $n \times \rho$. Besides Φ, we denote derivations with Ψ.

A *proof*, often denoted by Π, is a derivation with premiss t.

The *size* $|A|$ of a formula A is the number of unit and atom occurrences appearing in it. The *size* $|\Phi|$ of a derivation Φ is the sum of the sizes of the formulae occurring in it. The *length* of a derivation is the number of inference steps applied in the derivation. The *width* of a derivation is the maximal size of the formulae occurring in it.

Substitution

By $A\{a_1/B_1, \ldots, a_h/B_h\}$, we denote the operation of simultaneously substituting formulae $B_1, \ldots, B_h$ into all the occurrences of the atoms $a_1, \ldots, a_h$ in the formula A, respectively. By defining the substitution at the level of atoms, where atoms and their negation are equal citizen, we mean that the substitution to the occurrences of an atom doesn't touch the occurrences of its negation. Often, we only substitute certain occurrences of atoms: there will be no ambiguity because this is done in the context of atomic flows, where occurrences are distinguished with superscripts. The notion of substitution is extended to derivations in the natural way.

Inference Rules of SKS

Structural inference rules:

$$\mathsf{ai}{\downarrow}\,\frac{\mathsf{t}}{a \vee \bar{a}} \qquad \mathsf{aw}{\downarrow}\,\frac{\mathsf{f}}{a} \qquad \mathsf{ac}{\downarrow}\,\frac{a \vee a}{a}$$

identity (interaction) *weakening* *contraction*

$$\mathsf{ai}{\uparrow}\,\frac{a \wedge \bar{a}}{\mathsf{f}} \qquad \mathsf{aw}{\uparrow}\,\frac{a}{\mathsf{t}} \qquad \mathsf{ac}{\uparrow}\,\frac{a}{a \wedge a}$$

cut (cointeraction) *coweakening* *cocontraction*

Logical inference rules:

$$\mathsf{s}\,\frac{A \wedge [B \vee C]}{(A \wedge B) \vee C} \qquad \mathsf{m}\,\frac{(A \wedge B) \vee (C \wedge D)}{[A \vee C] \wedge [B \vee D]}$$

switch *medial*

There are also *equality* rules $=\dfrac{C}{D}$, for C and D on opposite sides in one of the following equations:

$$\begin{aligned} A \vee B &= B \vee A & A \vee \mathsf{f} &= A \\ A \wedge B &= B \wedge A & A \wedge \mathsf{t} &= A \\ [A \vee B] \vee C &= A \vee [B \vee C] & \mathsf{t} \vee \mathsf{t} &= \mathsf{t} \\ (A \wedge B) \wedge C &= A \wedge (B \wedge C) & \mathsf{f} \wedge \mathsf{f} &= \mathsf{f} \end{aligned} \tag{1}$$

Conventions

(a) In derivations we freely use equality rules without mentioning them. For instance

$$\mathsf{ai}{\downarrow}\,\dfrac{\begin{array}{c}=\dfrac{\begin{array}{c}=\dfrac{A}{A\wedge\mathsf{t}}\\ \mathsf{t}\wedge A\end{array}}{}\\ [p\vee\bar{p}]\wedge A\end{array}}{} \quad \text{is written} \quad \mathsf{ai}{\downarrow}\,\dfrac{A}{[p\vee\bar{p}]\wedge A}\ .$$

(b) The structural rules have been given in atomic form in SKS. This is possible because in deep inference the general form of the structural rules, given below, is derivable from their atomic form, moreover it is derivable with a polynomial cost.

$$\mathsf{i}{\downarrow}\,\dfrac{\mathsf{t}}{A\vee\bar{A}} \qquad \mathsf{w}{\downarrow}\,\dfrac{\mathsf{f}}{A} \qquad \mathsf{c}{\downarrow}\,\dfrac{A\vee A}{A}$$

$$\mathsf{i}{\uparrow}\,\dfrac{A\wedge\bar{A}}{\mathsf{f}} \qquad \mathsf{w}{\uparrow}\,\dfrac{A}{\mathsf{t}} \qquad \mathsf{c}{\uparrow}\,\dfrac{A}{A\wedge A}$$

We will freely use a nonatomic rule instance to stand for some derivation in SKS that derives that instance.

Operations on Derivations

Inference rules being applicable in any context, given a context $K\{\ \}$ and a derivation $\Phi : A \to B$, one can form a derivation $K\{\Phi\} : K\{A\} \to K\{B\}$ by adding the context $K\{\ \}$ at each inference step of the derivation. Given two derivations $\Phi : A \to B$ and $\Psi : C \to D$, one can form in this way the derivations $\Phi \wedge C : A \wedge C \to B \wedge C$ and $B \wedge \Psi : B \wedge C \to B \wedge D$. Then one can put one after the other to get a derivation $\begin{array}{c} A\wedge C \\ \| \\ B\wedge D \end{array}$ of $B \wedge D$ from $A \wedge C$; we denote by $\Phi \wedge \Psi : A \wedge C \to B \wedge D$ this derivation which consists in making Φ and then Ψ. In the same way, one can get a derivation $\Phi \vee \Psi : A \vee C \to B \vee D$ of $B \vee D$ from $A \vee C$. We will freely use these constructions throughout the paper.

3 Atomic Flows

Atomic flows, which have been introduced in [9], are, essentially, specialised Buss flow graphs [6]. They are particular directed graphs associated with SKS derivations: every derivation yields one atomic flow obtained by tracing the atoms (propositional letters and their negation) occurrences in the derivation. More precisely, one traces the behaviour of these occurrences through the structural rules: creation / destruction / duplication. No information about instances of logical rules is kept, only structural rules play a role and, as a consequence,

infinitely many derivations correspond to each atomic flow. As shown in [9], it turns out that atomic flows contain sufficient structure to control cut elimination procedures, providing in particular induction measures that can be used to ensure termination. Such cut-elimination procedures require exponential time on the size of the derivation to be normalised. In the present work, we improve the complexity of cut elimination to quasipolynomial time, using in addition threshold formulae, which are independent from the given proof.

Atomic Flow Associated to a Derivation

We first index occurrences of atoms in derivations with natural numbers in such a way that:

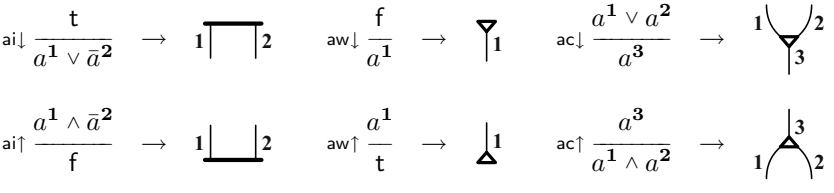
- different occurrences of atoms in formulae have different indexes;
- indexes are preserved by logical rules and the context part of structural rules;
- in each instance of a structural rule, active occurrences of atoms have different indexes; for example an instance of the contraction rule becomes $\mathsf{ac}{\downarrow}\dfrac{a^{\mathbf{1}} \vee a^{\mathbf{2}}}{a^{\mathbf{3}}}$.

We associate inductively (say, in a top-down manner) to each derivation with indexed occurrences of atoms an atomic flow as follows:

- to a formula $A(a^{\mathbf{1}}, \ldots, a^{\mathbf{n}})$ with exactly n occurrences of atoms, the following flow, consisting of n edges, is associated:

$$\mathbf{1} \,\big|\, \cdots \,\big|\, \mathbf{n} \ ;$$

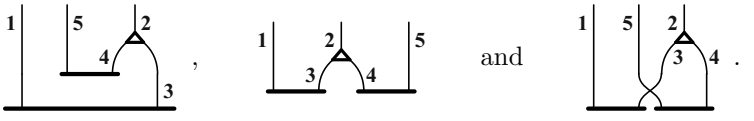
- the logical rules and the context part of structural rule do not change the flow;
- each instance of a structural rule adds a *vertex*, whose incident edges correspond to active occurrences of atoms in the rule; the association of vertices to structural rules is illustrated below:



The left-hand side of each arrow shows an instance of a structural rule, whose atom occurrences are labelled by small numerals. Correspondingly, the right-hand side of the same arrow, shows the vertex associated to the given rule instance: the labelling of incident edges respects the labelling of atom occurrences. In a top-down inductive reading of the proof, the upper edges of the vertices are meant to be associated to the already defined flow and the lower edges are new ones. Moreover, we qualify each vertex according to the rule it corresponds to:

for example, in a given atomic flow, we might talk about a *contraction vertex*, or a *cut vertex*, and so on. Instead of small numerals, sometimes we use ϵ or ι to label edges (as well as atom occurrences), but we do not always use labels.

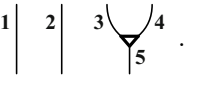
All edges are directed, but we do not explicitly show the orientation. Instead, we consider it as implicitly given by the way we draw them, i.e., edges are oriented along the vertical direction. So, the vertices corresponding to dual rules, are mutually distinct: for example, an identity vertex and a cut vertex are different because the orientation of their edges is different. On the other hand, the horizontal direction plays no role in distinguishing atomic flows; this corresponds to commutativity of logical relations. Here are for instance three representations of the same flow:



It should be noted that atomic flows built from derivations have no directed cycles and bear a natural polarity assignment (corresponding to atoms versus negated atoms in the derivation), that is a mapping of each edge to an element of $\{-,+\}$, such that the two edges of each identity or cut vertex map to different values and the three edges of each contraction or cocontraction vertex map to the same value. We denote atomic flows by ϕ and ψ.

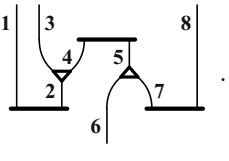
Examples of Atomic Flows Associated to Derivations

$$\mathsf{ac}{\downarrow}\ \frac{a^{\mathbf{1}} \wedge \left[b^{\mathbf{2}} \vee \left[a^{\mathbf{3}} \vee a^{\mathbf{4}}\right]\right]}{a^{\mathbf{1}} \wedge \left[b^{\mathbf{2}} \vee a^{\mathbf{5}}\right]} \quad \rightarrow \quad$$



.

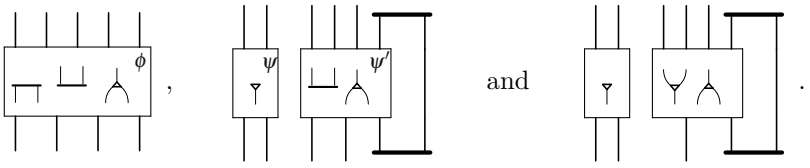
$$\begin{array}{l}
\mathsf{ai}{\downarrow}\ \dfrac{\left(a^{\mathbf{1}} \wedge \left[\bar{a}^{\mathbf{3}} \vee \mathsf{t}\right]\right) \wedge \bar{a}^{\mathbf{8}}}{\left(a^{\mathbf{1}} \wedge \left[\bar{a}^{\mathbf{3}} \vee \left[\bar{a}^{\mathbf{4}} \vee a^{\mathbf{5}}\right]\right]\right) \wedge \bar{a}^{\mathbf{8}}} \\
= \dfrac{\left(a^{\mathbf{1}} \wedge \left[\left[\bar{a}^{\mathbf{3}} \vee \bar{a}^{\mathbf{4}}\right] \vee a^{\mathbf{5}}\right]\right) \wedge \bar{a}^{\mathbf{8}}}{} \\
\mathsf{s}\ \dfrac{\left[\left(a^{\mathbf{1}} \wedge \left[\bar{a}^{\mathbf{3}} \vee \bar{a}^{\mathbf{4}}\right]\right) \vee a^{\mathbf{5}}\right] \wedge \bar{a}^{\mathbf{8}}}{} \\
\mathsf{ac}{\downarrow}\ \dfrac{\left[\left(a^{\mathbf{1}} \wedge \bar{a}^{\mathbf{2}}\right) \vee a^{\mathbf{5}}\right] \wedge \bar{a}^{\mathbf{8}}}{} \\
\mathsf{ai}{\uparrow}\ \dfrac{\left[\mathsf{f} \vee a^{\mathbf{5}}\right] \wedge \bar{a}^{\mathbf{8}}}{} \\
= \dfrac{a^{\mathbf{5}} \wedge \bar{a}^{\mathbf{8}}}{} \\
\mathsf{ac}{\uparrow}\ \dfrac{\left(a^{\mathbf{6}} \wedge a^{\mathbf{7}}\right) \wedge \bar{a}^{\mathbf{8}}}{} \\
= \dfrac{a^{\mathbf{6}} \wedge \left(a^{\mathbf{7}} \wedge \bar{a}^{\mathbf{8}}\right)}{} \\
\mathsf{ai}{\uparrow}\ \dfrac{}{a^{\mathbf{6}} \wedge \mathsf{f}}
\end{array} \quad \rightarrow \quad$$



.

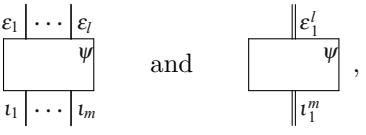
Abstract Notation of Atomic Flows

When certain details of a flow are not important, but only the vertex kinds and its upper and lower edges are, we can use boxes, labelled with all the vertex kinds that can appear in the flow they represent. For example:



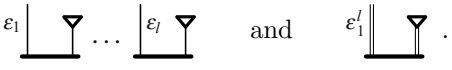
When no vertex labels appear on a box, we assume that the vertices in the corresponding flow can be any (so, it does not mean that there are no vertices in the flow).

We sometimes use a double line notation for representing multiple edges. For example, the following diagrams represent the same flow:

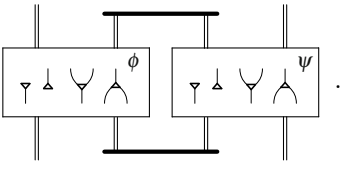


where $l, m \geq 0$; note that we use $\boldsymbol{\epsilon}_1^l$ to denote the vector $(\epsilon_1, \ldots, \epsilon_l)$. We might label multiple edges with either a vector of the associated atom occurrences in a derivation or one of the formulae the associated atom occurrences belong to in a derivation.

We extend the double line notation to collections of isomorphic flows. For example, for $l \geq 0$, the following diagrams represent the same flow:



We observe that the flow of every SKS derivation can always be represented as follows:



4 Normalisation Step 1: Simple Form

The first step in our normalisation procedure, defined here, consists in routine deep-inference manipulations, which are best understood in conjunction with atomic flows. For this reason, this section is a useful exercise for a reader who is not familiar with deep inference and atomic flows.

In Theorem 5 of this section, we show that every proof can be transformed into 'simple form'. Proofs in simple form are such that we can substitute formulae for all the atom occurrences that appear in cut instances, without substituting for atom occurrences that appear in the conclusion of the derivation. Of course, doing this would invalidate identity and cut instances, but in Sect. 6 we see how we can build a valid cut-free proof from the broken derivations obtained by substituing formulae into derivations in simple form.

We first show some standard deep-inference results. We will see how we can permute all the identity (resp., cut) rule instances to the top (resp., bottom) of a proof, without changing the atomic flow of the proof, and without significantly changing the size of the proof.

Lemma 1. *Given a context $K\{\ \}$ and a formula A, there exist derivations*

$$\begin{array}{c} A \wedge K\{\mathsf{t}\} \\ \| \{\mathsf{s}\} \\ K\{A\} \end{array} \quad \textit{and} \quad \begin{array}{c} K\{A\} \\ \| \{\mathsf{s}\} \\ K\{\mathsf{f}\} \vee A \end{array} ,$$

each of whose width is the size of $K\{A\}$ plus one and length is bounded by a polynomial in the size of $K\{\ \}$.

Proof. The result follows by structural induction on $K\{\ \}$: The base cases are:

$$= \frac{A \wedge \{\mathsf{t}\}}{\{A\}} \quad \text{and} \quad = \frac{\{A\}}{\{\mathsf{f}\} \vee A} \; .$$

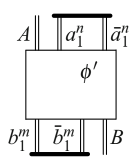
The inductive cases are

$$\begin{array}{c} = \dfrac{A \wedge (B \wedge K\{\mathsf{t}\})}{B \wedge (A \wedge K\{\mathsf{t}\})} \\ \| \{\mathsf{s}\} \\ B \wedge K\{A\} \end{array} , \qquad \begin{array}{c} \mathsf{s}\,\dfrac{A \wedge [B \vee K\{\mathsf{t}\}]}{B \vee (A \wedge K\{\mathsf{t}\})} \\ \| \{\mathsf{s}\} \\ B \vee K\{A\} \end{array} \quad \text{and}$$

$$\begin{array}{c} B \wedge K\{A\} \\ \| \{\mathsf{s}\} \\ \mathsf{s}\,\dfrac{B \wedge [K\{\mathsf{f}\} \vee A]}{(B \wedge K\{\mathsf{f}\}) \vee A} \end{array} , \qquad \begin{array}{c} B \vee K\{A\} \\ \| \{\mathsf{s}\} \\ = \dfrac{B \vee [K\{\mathsf{f}\} \vee A]}{[B \vee K\{\mathsf{f}\}] \vee A} \end{array} .$$

□

Lemma 2. *Given a derivation $\Phi : A \to B$, with flow*

$\phi =$  ,

there exists a derivation

$$\Psi = \begin{array}{c} n{\times}\mathsf{ai}\!\downarrow \dfrac{A}{A \wedge [a_1 \vee \bar{a}_1] \wedge \cdots \wedge [a_n \vee \bar{a}_n]} \\ \Psi' \| \\ m{\times}\mathsf{ai}\!\uparrow \dfrac{(b_1 \wedge \bar{b}_1) \vee \cdots \vee (b_m \wedge \bar{b}_m) \vee B}{B} \end{array} ,$$

for some atoms $a_1, \ldots, a_n, b_1, \ldots, b_m$ and some derivation Ψ', such that the flow of Ψ is ϕ, the flow of Ψ' is ϕ' and the size of Ψ is bounded by a polynomial in the size of Φ.

Proof. For every relevant interaction we perform the following transformation:

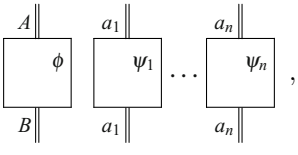
$$\mathsf{ai}{\downarrow}\,\dfrac{\begin{array}{c}A\\ \Phi'\|\\ K\{\mathsf{t}\}\end{array}}{K\{a\vee\bar{a}\}}\ \Phi''\|\ B \quad \text{is transformed into} \quad \begin{array}{c}\mathsf{ai}{\downarrow}\,\dfrac{A}{[a\vee\bar{a}]\wedge A}\\ [a\vee\bar{a}]\wedge\Phi'\|\\ [a\vee\bar{a}]\wedge K\{\mathsf{t}\}\\ \|\,\{\mathsf{s}\}\\ K\{a\vee\bar{a}\}\\ \Phi''\|\\ B\end{array},$$

which is possible by Lemma 1. Instances of cut rules can be dealt with in a symmetric way.

Each transformation increases the width of the derivation by a constant and increases the length by at most a polynomial in the width of the derivation. Hence, the size of Ψ is bounded by a polynomial in the size of Φ. □

We now show how to extend substitutions from formulae to derivations. Using atomic flows, we single out some atom occurrences that we substitute for. Substitutions play a crucial role in Theorem 5 and in Theorem 11. It is important to notice that a substitution only copies atomic flows, it does not introduce new vertices; and that the cost of substitution is polynomial.

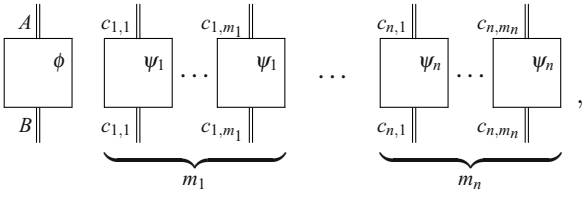
Lemma 3. *Given a derivation $\Phi : A \to B$, let its associated flow have shape*



such that, for $1 \le i \le n$, all the edges of ψ_i are mapped to from occurrences of a_i, then, for any formulae $C_1, \ldots, C_n$ there exists a derivation

$$\Psi \;=\; \begin{array}{c}A\{a_1^{\psi_1}/C_1, \ldots, a_n^{\psi_n}/C_n\}\\ \|\\ B\{a_1^{\psi_1}/C_1, \ldots, a_n^{\psi_n}/C_n\}\end{array},$$

whose flow is



where, for every $1 \le i \le n$, the atom occurrences of C_i are $c_{i,1}, \ldots, c_{i,m_i}$; moreover, the size of Ψ is bounded by a polynomial in the size of Φ and, for each $1 \le i \le n$, the size of C_i.

Proof. We sketch the proof: For each $1 \leq i \leq n$, we can proceed by structural induction on C_i and then on ψ_i. For the two cases of $C_i \equiv D \vee E$ and $C_i \equiv D \wedge E$ we have to consider, for each vertex of ψ_i, one of the following situations (notice that ψ_i can not contain interaction or cut vertices):

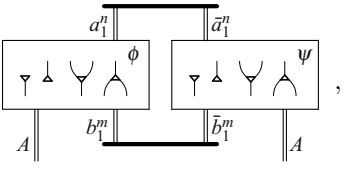
$$\mathsf{w}\downarrow\frac{\mathsf{w}\downarrow\dfrac{\mathsf{f}}{E}}{D \vee E}\quad,\qquad \mathsf{w}\downarrow\frac{\mathsf{w}\downarrow\dfrac{\mathsf{f}}{\mathsf{f} \wedge E}}{D \wedge E}\quad,\qquad \mathsf{c}\downarrow\frac{\mathsf{c}\downarrow\dfrac{D \vee D \vee E \vee E}{D \vee D \vee E}}{D \vee E}\quad,\qquad \mathsf{c}\downarrow\frac{\mathsf{m}\dfrac{(D \wedge E) \vee (D \wedge E)}{[D \vee D] \wedge [E \vee E]}}{\mathsf{c}\downarrow\dfrac{[D \vee D] \wedge E}{D \wedge E}}\quad,$$

and their dual ones. □

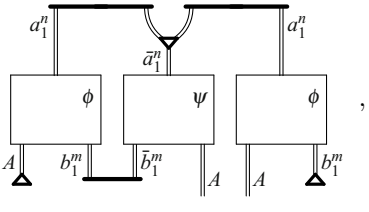
Notation 4. When we write $\Phi\{a_1^{\psi_1}/C_1, \ldots, a_n^{\psi_n}/C_n\}$, we mean the derivation Ψ obtained in the proof of Lemma 3.

We now present the main result of this section. We show that any derivation can be transformed into a derivation whose atomic flow is on what we call 'simple form'. Referring to the second flow in Theorem 5, we observe that we could substitue for the atom occurrences corresponding to the rightmost copy of ϕ without substituting for any atom occurrence appearing in the conclusion of the proof. This is one of the two main ingredients in our normalisation procedure.

Theorem 5. *Given a proof Φ of A, with flow*



there exists a proof Ψ of A, with flow

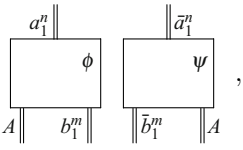


such that the size of Ψ is bounded by a polynomial in the size of Φ.

Proof. Consider the derivation

$$\begin{array}{c}\left[a_1^{\phi} \vee \bar{a}_1^{\psi}\right] \wedge \cdots \wedge \left[a_n^{\phi} \vee \bar{a}_n^{\psi}\right] \\ \Phi' \| \\ \left(b_1^{\phi} \wedge \bar{b}_1^{\psi}\right) \vee \cdots \vee \left(b_m^{\phi} \wedge \bar{b}_m^{\psi}\right) \vee A\end{array}\quad,$$

with atomic flow



,

which exists and whose size is bounded by a polynomial in the size of Φ by Lemma 2. Let $a_1, \ldots, a_n, b_1, \ldots, b_m, c_1, \ldots, c_l$ be all the atoms whose occurrences are mapped to edges in ϕ and let

$$\sigma = \{a_1^\phi / (a_1 \wedge a_1), \ldots, a_n^\phi / (a_n \wedge a_n), b_1^\phi / (b_1 \wedge b_1), \ldots, b_m^\phi / (b_m \wedge b_m), c_1^\phi / (c_1 \wedge c_1), \ldots, c_l^\phi / (c_l \wedge c_l)\} \ .$$

We then construct Ψ:

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathsf{t}}{[a_1 \vee \bar{a}_1] \wedge [a_1 \vee \bar{a}_1] \wedge \cdots \wedge [a_n \vee \bar{a}_n] \wedge [a_n \vee \bar{a}_n]}\, 2n\times\mathsf{ai}\downarrow}{[(a_1 \wedge a_1) \vee \bar{a}_1 \vee \bar{a}_1] \wedge \cdots \wedge [(a_n \wedge a_n) \vee \bar{a}_n \vee \bar{a}_n]}\, 2n\times\mathsf{s}}{[(a_1 \wedge a_1) \vee \bar{a}_1] \wedge \cdots \wedge [(a_n \wedge a_n) \vee \bar{a}_n]}\, n\times\mathsf{ac}\downarrow}{\big((b_1 \wedge b_1) \wedge \bar{b}_1\big) \vee \cdots \vee \big((b_m \wedge b_m) \wedge \bar{b}_m\big) \vee A\sigma}\, \Phi'\sigma \qquad ,$$

$$\dfrac{\big(b_1 \wedge \bar{b}_1\big) \vee \cdots \vee \big(b_m \wedge \bar{b}_m\big) \vee A}{A}\, m\times\mathsf{ai}\uparrow \quad \{\mathsf{aw}\uparrow\}$$

with the required atomic flow, where, by Lemma 3, the derivation $\Phi'\sigma$ exists and its size is bounded by a polynomial in the size of Φ'. □

5 Threshold Formulae

Threshold formulae realise boolean threshold functions, which are defined as boolean functions that are true if and only if at least k of n inputs are true (see [11] for a thorough reference on threshold functions).

There are several ways of encoding threshold functions into formulae, and the problem is to find, among them, an encoding that allows us to obtain Theorem 10. Efficiently obtaining the property stated in Theorem 10 crucially depends also on the proof system we adopt.

The following class of threshold formulae, which we found to work for system SKS, is a simplification of the one adopted in [1].

In the rest of this paper, whenever we have a sequence of atoms $a_1, \ldots, a_n$, we will assume, without loss of generality, that n is a power of two.

Definition 6. *For every $n = 2^m$, with $m \geq 0$, and $k \geq 0$, we define the operator θ_k^n inductively as follows:*

$$\theta^n_k(a_1,\ldots,a_n) = \\ = \begin{cases} \mathsf{t} & \text{if } k = 0 \\ \mathsf{f} & \text{if } k > n \\ a_1 & \text{if } n = k = 1 \\ \bigvee_{\substack{i+j=k \\ 0\le i,j\le n/2}} \left(\theta^{n/2}_i(a_1,\ldots,a_{n/2}) \wedge \theta^{n/2}_j(a_{n/2+1},\ldots,a_n)\right) & \text{otherwise.} \end{cases}$$

For any n atoms $a_1, \ldots, a_n$, we call $\theta^n_k(a_1,\ldots,a_n)$ the threshold formula at level k *(*with respect to $a_1, \ldots, a_n$*).*

The size of the threshold formulae dominates the cost of the normalisation procedure, so, we evaluate their size.

Lemma 7. *For any $n = 2^m$, with $m \ge 0$, and $k \ge 0$ the size of $\theta^n_k(a_1,\ldots,a_n)$ has a quasipolynomial bound in n.*

Proof. We show that the size of $\theta^n_k(a_1,\ldots,a_n)$ is bounded by $n^{2\log n}$. We reason by induction on n; the case $n = 1$ trivially holds. For $n > 1$, we consider that the size of $\theta^n_k(a_1,\ldots,a_n)$ is bounded by

$$\sum_{\substack{i+j=k \\ 0\le i\le n \\ 0\le j\le n}} 2n/2^{2\log n/2} \quad .$$

We then have

$$\sum_{\substack{i+j=k \\ 0\le i\le n \\ 0\le j\le n}} 2n/2^{2\log n/2} \le \sum_{\substack{i+j=n/2 \\ 0\le i\le n \\ 0\le j\le n}} 2n/2^{2\log n/2} \le (n+2)n/2^{2\log n/2} \quad ,$$

and since $n + 2 \le n^2$ and $n/2 < n$, we have

$$(n+2)n/2^{2\log n/2} \le n^2 n^{2\log n/2} = n^{2\log n - 2\log 2 + 2} = n^{2\log n} \quad ,$$

as required. □

Lemma 8. *For any $n = 2^m$, with $m \ge 0$, $k \ge 0$ and $1 \le i \le n$, there exists a derivation*

$$\Gamma^i_k \;=\; \begin{array}{c} \theta^n_k(a_1,\ldots,a_n)\{a_i/\mathsf{f}\} \\ \| \{\mathsf{aw}{\downarrow},\mathsf{aw}{\uparrow}\} \\ \theta^n_{k+1}(a_1,\ldots,a_n)\{a_i/\mathsf{t}\} \end{array} \quad ,$$

whose size has a quasipolynomial bound in n.

Proof. The result follows by Lemma 7 and structural induction on Definition 6. It is worth noting that both the premiss and the conclusion of Γ^i_k are logically equivalent to $\theta^{n-1}_k(a_1,\ldots,a_{i-1},a_{i+1},\ldots,a_n)$. □

Lemma 9. *Given a formula A and an atom a that occurs in A, there exist derivations* $\begin{array}{c} a \wedge A\{a/\mathsf{t}\} \\ \| \{\mathsf{ac}\uparrow,\mathsf{s}\} \\ A \end{array}$ *and* $\begin{array}{c} A \\ \| \{\mathsf{ac}\downarrow,\mathsf{s}\} \\ A\{a/\mathsf{f}\} \vee a \end{array}$ *such that their sizes are both bounded by a polynomial in the size of A.*

Proof. The result follows by induction on the number of occurrences of a in A, and Lemma 1. □

We now present the main result of this section. We show that, using threshold functions, we are able to deduce a conjunction of disjunctions from a disjunction of (slightly different) conjunctions. This construction is based on (seen top-down) contractions meeting cocontrations, and can be thought of as a generalisation of the simple sharing mechanism that allows us to deduce $a \wedge \cdots \wedge a$ from $a \vee \cdots \vee a$.

In Theorem 11 we will see how using this sharing mechanism allows us to glue together several 'broken' derivations in order to build a cut-free proof.

Theorem 10. *Let, for some $n = 2^m$ with $m \geq 0$, $a_1, \ldots, a_n$ be distinct atoms. Then, for every $1 \leq k \leq n+1$, there exists a derivation*

$$\Gamma_k \;=\; \begin{array}{c} \big(a_1 \wedge \theta^n_{k-1}(a_1,\ldots,a_n)\{a_1/\mathsf{f}\}\big) \vee \cdots \vee \big(a_n \wedge \theta^n_{k-1}(a_1,\ldots,a_n)\{a_n/\mathsf{f}\}\big) \\ \Big\| \mathsf{SKS}\setminus\{\mathsf{ai}\downarrow,\mathsf{ai}\uparrow\} \\ [a_1 \vee \theta^n_k(a_1,\ldots,a_n)\{a_1/\mathsf{f}\}] \wedge \cdots \wedge [a_n \vee \theta^n_k(a_1,\ldots,a_n)\{a_n/\mathsf{f}\}] \end{array},$$

such that the size of Γ_k has a quasipolynomial bound in n.

Proof. For $1 \leq k \leq n+1$, we construct:

$$\Gamma_k \;=\; \begin{array}{c} \big(a_1 \wedge \theta^n_{k-1}(a_1,\ldots,a_n)\{a_1/\mathsf{f}\}\big) \vee \cdots \vee \big(a_n \wedge \theta^n_{k-1}(a_1,\ldots,a_n)\{a_n/\mathsf{f}\}\big) \\ \Gamma^1_k \vee \cdots \vee \Gamma^n_k \,\big\|\, \{\mathsf{aw}\downarrow,\mathsf{aw}\uparrow\} \\ (a_1 \wedge \theta^n_k(a_1,\ldots,a_n)\{a_1/\mathsf{t}\}) \vee \cdots \vee (a_n \wedge \theta^n_k(a_1,\ldots,a_n)\{a_n/\mathsf{t}\}) \\ \Phi_1 \,\big\|\, \{\mathsf{ac}\uparrow,\mathsf{s}\} \\ \bigvee_{1\leq i\leq n} \theta^n_k(a_1,\ldots,a_n) \\ \| \{\mathsf{c}\downarrow\} \\ \theta^n_k(a_1,\ldots,a_n) \\ \| \{\mathsf{c}\uparrow\} \\ \bigwedge_{1\leq i\leq n} \theta^n_k(a_1,\ldots,a_n) \\ \Phi_2 \,\big\|\, \{\mathsf{ac}\downarrow,\mathsf{s}\} \\ [a_1 \vee \theta^n_k(a_1,\ldots,a_n)\{a_1/\mathsf{f}\}] \wedge \cdots \wedge [a_n \vee \theta^n_k(a_1,\ldots,a_n)\{a_n/\mathsf{f}\}] \end{array},$$

where Φ_1 and Φ_2 exist by Lemma 9 and, for $1 \leq i \leq n$, Γ^i_k exists by Lemma 8. The size of Γ_k is quasipolynomial in n, by Lemma 8 and Lemma 9. □

6 Normalisation Step 2: Cut Elimination

We now show the main construction of this paper. A cut-elimination result for derivations in simple form. The procedure uses a class of external and independent derivations in order to glue together pieces of the original proof. One valid class of such derivations are the ones shown in Sect. 5.

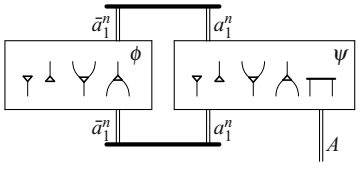
Theorem 11. *Let*

1. $N > 0$ *be an integer;*
2. $a_1, \ldots, a_n$ *be distinct atoms, where* $n = 2^m$ *for some* $m \geq 0$*;*
3. *there be, for every* $0 < k < N$ *and* $1 \leq i \leq n$*, a formula* $C_k^{a_i}$*;*
4. *there be, for every* $1 \leq k \leq N$*, a derivation*

$$\Gamma_k \;=\; \begin{array}{c} \left(a_1 \wedge C_{k-1}^{a_1}\right) \vee \cdots \vee \left(a_n \wedge C_{k-1}^{a_n}\right) \\ \Vert\, \mathsf{SKS}\backslash\{\mathsf{ai}{\downarrow},\mathsf{ai}{\uparrow}\} \\ \left[a_1 \vee C_k^{a_1}\right] \wedge \cdots \wedge \left[a_n \vee C_k^{a_n}\right] \end{array} ,$$

where $C_0^{a_1} \equiv \cdots \equiv C_0^{a_n} \equiv \mathsf{t}$ *and* $C_N^{a_1} \equiv \cdots \equiv C_N^{a_n} \equiv \mathsf{f}$.

For every proof Φ *of* A*, whose flow is*

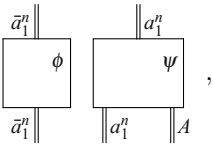


where only occurrences of the atoms $\bar{a}_1, \ldots, \bar{a}_n$ *are mapped to edges in* ϕ*, there exists a cut-free proof* Ψ *of* A *whose size is bounded by a polynomial in* N*, the size of* Φ *and, for* $1 \leq k \leq N$*, the size of* Γ_k*.*

Proof. For every $1 \leq i \leq n$, let m_i (resp., m_i') be the number of interactions (resp., cuts) where a_i^ψ and $\bar{a}_i^\phi$ appears in Φ, and consider the derivation

$$\Phi' \;=\; \begin{array}{c} \bigwedge_{1\leq j\leq m_1} \left[a_1^\psi \vee \bar{a}_1^\phi\right] \wedge \cdots \wedge \bigvee_{1\leq j\leq m_n} \left[a_n^\psi \vee \bar{a}_n^\phi\right] \\ \Vert \\ A \vee \bigvee_{1\leq j\leq m_1'} \left(a_1^\psi \wedge \bar{a}_1^\phi\right) \vee \cdots \vee \bigvee_{1\leq j\leq m_n'} \left(a_n^\psi \wedge \bar{a}_n^\phi\right) \end{array} ,$$

with atomic flow



which exists by Lemma 2. Then, for $0 \leq k \leq N$, construct the following derivation:

$$\Phi_k \;=\; \begin{array}{c} \left[a_1 \vee C_k^{a_1}\right] \wedge \cdots \wedge \left[a_n \vee C_k^{a_n}\right] \\ \Vert\, \{\mathsf{c}{\uparrow},\mathsf{w}{\uparrow}\} \\ \bigwedge_{1\leq j\leq m_1} \left[a_1 \vee C_k^{a_1}\right] \wedge \cdots \wedge \bigvee_{1\leq j\leq m_n} \left[a_n \vee C_k^{a_n}\right] \\ \Phi'\{\bar{a}_1^\phi/C_k^{a_1},\ldots,\bar{a}_n^\phi/C_k^{a_n}\} \Vert\, \mathsf{SKS}\backslash\{\mathsf{ai}{\uparrow}\} \\ A \vee \bigvee_{1\leq j\leq m_1'} \left(a_1 \wedge C_k^{a_1}\right) \vee \cdots \vee \bigvee_{1\leq j\leq m_n'} \left(a_n \wedge C_k^{a_n}\right) \\ \Vert\, \{\mathsf{c}{\downarrow},\mathsf{w}{\downarrow}\} \\ A \vee \left(a_1 \wedge C_k^{a_1}\right) \vee \cdots \vee \left(a_n \wedge C_k^{a_n}\right) \end{array} ,$$

which exists, and whose size is bounded by a polynomial in the size of Φ and the size of Γ_k, by Lemma 3. We then construct the cut-free derivation Ψ as follows:

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{\mathsf{t}}{([a_1 \vee \mathsf{t}] \wedge \cdots \wedge [a_n \vee \mathsf{t}])}\, n\times\mathsf{aw}\downarrow
}{A \vee (a_1 \wedge C_0^{a_1}) \vee \cdots \vee (a_n \wedge C_0^{a_n})}\, \Phi_0 \| \mathsf{SKS}\setminus\{\mathsf{ai}\uparrow\}
}{A \vee ([a_1 \vee C_1^{a_1}] \wedge \cdots \wedge [a_n \vee C_1^{a_n}])}\, A\vee\Gamma_1 \| \mathsf{SKS}\setminus\{\mathsf{ai}\downarrow,\mathsf{ai}\uparrow\}
}{A \vee [A \vee (a_1 \wedge C_1^{a_1}) \vee \cdots \vee (a_n \wedge C_1^{a_n})]}\, A\vee\Phi_1 \| \mathsf{SKS}\setminus\{\mathsf{ai}\uparrow\}
}{\vdots}
}{A \vee A \vee \cdots \vee A \vee \left([a_1 \vee C_{N-1}^{a_1}] \wedge \cdots \wedge [a_n \vee C_{N-1}^{a_n}]\right)}\, A\vee A\vee\cdots\vee A\vee\Gamma_{N-1} \| \mathsf{SKS}\setminus\{\mathsf{ai}\downarrow,\mathsf{ai}\uparrow\}
}{A \vee A \vee \cdots \vee A \vee \left[A \vee \left(a_1 \wedge C_{N-1}^{a_1}\right) \vee \cdots \vee \left(a_n \wedge C_{N-1}^{a_n}\right)\right]}\, A\vee A\vee\cdots\vee A\vee\Phi_{N-1} \| \mathsf{SKS}\setminus\{\mathsf{ai}\uparrow\}
}{A \vee A \vee \cdots \vee A \vee A \vee ([a_1 \vee C_N^{a_1}] \wedge \cdots \wedge [a_n \vee C_N^{a_n}])}\, A\vee A\vee\cdots\vee A\vee\Gamma_N \| \mathsf{SKS}\setminus\{\mathsf{ai}\downarrow,\mathsf{ai}\uparrow\}
}{A \vee A \vee \cdots \vee A \vee A \vee [A \vee (a_1 \wedge \mathsf{f}) \vee \cdots \vee (a_n \wedge \mathsf{f})]}\, A\vee A\vee\cdots\vee A\vee A\vee\Phi_N \| \mathsf{SKS}\setminus\{\mathsf{ai}\uparrow\}
}{A \vee A \vee \cdots \vee A \vee A \vee A}\, n\times\mathsf{aw}\uparrow
}{A}\, (N-1)\times\mathsf{c}\downarrow
$$

□

It is worth noting that if we fix $N = n + 1$ in Theorem 11, the formulae $C_k^{a_i}$ are bound to be threshold formulae.

Corollary 12. *Given a proof Φ of A, there exists a cut-free proof Ψ of A, whose size is bounded by a quasipolynomial in the size of Φ.*

Proof. The result follows by Theorem 5, Theorem 10 and Theorem 11. □

7 Final Comments

The quasipolynomial cut-elimination procedure makes use of the cocontraction rule. But the cocontraction rule can also be eliminated. A natural question is whether one can extend the quasipolynomial cut elimination to a cocontraction elimination or to say it in another way, whether one can eliminate cuts in quasipolynomial time without the help of cocontractions. This is probably an important question because all indications we have point to an essential role being played by cocontraction in keeping the complexity low. Cocontraction has something to do with sharing, it seems to provide a typical 'dag-like' speed-up over the corresponding 'tree-like' expansion.

The role played by cocontractions is the most immediate explanation of why quasipolynomial cut elimination works in Deep Inference and not, at the present stage, in the sequent calculus (for full propositional logic). The reason seems

to be that exploiting cocontraction in the absence of cut is an intrinsic feature of deep inference, not achievable in Gentzen theory because of the lack of a top-down symmetry therein.

Another natural question is whether quasipolynomial time is the best we can do: there is no obvious objection to the existence of a polynomial cut-elimination procedure. It is possible to express threshold functions with polynomial formulae, but the hardest problem seems to be to obtain corresponding derivations of polynomial length. Deep inference flexibility in constructing derivations might help here.

The cut-elimination procedure presented here is peculiar because it achieves its result by using an external scheme, constituted by the threshold functions and the corresponding derivations, which does not depend on the particular derivation we are working on. It is as if the threshold construction was a catalyzer that shorten the cut elimination. It would be interesting to interpret this phenomenon computationally, in some sort of Curry-Howard correspondence, where the threshold construction implements a clever operator. We intend to explore this path in the near future.

This leads to the wider question of a computational interpretation of deep inference. Atomic flows are a weak computational trace, which takes only the structural rules into account. It is surprising that such a trace, which forgets all the information given by the logical rules, is powerful enough to drive the cut-elimination procedure. We intend to carefully study its computational power and to see whether one can construct on this ground an original computational interpretation of proofs.

References

1. Atserias, A., Galesi, N., Pudlák, P.: Monotone simulations of non-monotone proofs. Journal of Computer and System Sciences 65(4), 626–638 (2002)
2. Brünnler, K.: Deep Inference and Symmetry in Classical Proofs. Logos Verlag, Berlin (2004), http://www.iam.unibe.ch/~kai/Papers/phd.pdf
3. Brünnler, K.: Deep inference and its normal form of derivations. In: Beckmann, A., Berger, U., Löwe, B., Tucker, J.V. (eds.) CiE 2006. LNCS, vol. 3988, pp. 65–74. Springer, Heidelberg (2006), http://www.iam.unibe.ch/~kai/Papers/n.pdf
4. Brünnler, K., Tiu, A.F.: A local system for classical logic. In: Nieuwenhuis, R., Voronkov, A. (eds.) LPAR 2001. LNCS (LNAI), vol. 2250, pp. 347–361. Springer, Heidelberg (2001), http://www.iam.unibe.ch/~kai/Papers/lcl-lpar.pdf
5. Bruscoli, P., Guglielmi, A.: On the proof complexity of deep inference. ACM Transactions on Computational Logic 10(2), 1–34 (2009), Article 14, http://cs.bath.ac.uk/ag/p/PrComplDI.pdf
6. Buss, S.R.: The undecidability of k-provability. Annals of Pure and Applied Logic 53(1), 75–102 (1991)
7. Guglielmi, A.: Deep inference and the calculus of structures, http://alessio.guglielmi.name/res/cos
8. Guglielmi, A.: A system of interaction and structure. ACM Transactions on Computational Logic 8(1), 1–64 (2007), http://cs.bath.ac.uk/ag/p/SystIntStr.pdf

9. Guglielmi, A., Gundersen, T.: Normalisation control in deep inference via atomic flows. Logical Methods in Computer Science 4(1:9), 1–36 (2008), http://arxiv.org/pdf/0709.1205
10. Jeřábek, E.: Proof complexity of the cut-free calculus of structures. Journal of Logic and Computation (2008) (in press), http://www.math.cas.cz/~jerabek/papers/cos.pdf
11. Wegener, I.: The Complexity of Boolean Functions. John Wiley & Sons Ltd and B. G. Teubner, Stuttgart (1987)

Pairwise Cardinality Networks*

Michael Codish and Moshe Zazon-Ivry

Department of Computer Science, Ben-Gurion University, Israel
{mcodish,moshezaz}@cs.bgu.ac.il

Abstract. We introduce pairwise cardinality networks, networks of comparators, derived from pairwise sorting networks, which express cardinality constraints. We show that pairwise cardinality networks are superior to the cardinality networks introduced in previous work which are derived from odd-even sorting networks. Our presentation identifies the precise relationship between odd-even and pairwise sorting networks. This relationship also clarifies why pairwise sorting networks have significantly better propagation properties for the application of cardinality constraints.

1 Introduction

Cardinality constraints take the form, $x_1 + x_2 + \cdots + x_n \prec k$, where $x_1, \ldots, x_n$ are Boolean variables, k is a natural number, and $\prec$ is one of $\{<, \leq, >, \geq, =\}$. Cardinality constraints are well studied and arise in many different contexts. One typical example is the Max-SAT problem where for a given propositional formula (in CNF) with clauses $\{C_1, \ldots, C_n\}$, we seek an assignment that satisfies a maximal number of clauses. One approach is to add a fresh blocking variable to each clause giving $\varphi = \{C_1 \vee x_1, \ldots, C_n \vee x_n\}$. Now we seek a minimal value k such that $\varphi \wedge (x_1 + x_2 + \cdots + x_n < k)$ is satisfiable. We can do this by encoding the cardinality constraint to a propositional formula ψ_k and repeatedly applying a SAT solver to find the smallest k such that $\varphi \wedge \psi_k$ is satisfiable.

There are many works that describe techniques to encode cardinality constraints to propositional formulas. The starting points for this paper are the work by Asin *et al.*[1] and an earlier paper [4] which describes how pseudo Boolean constraints (which are more general than cardinality constraints) are translated to SAT in the MiniSAT solver. Both of these papers consider an encoding technique based on the use of sorting networks.

A (Boolean) sorting network is a circuit that receives n Boolean inputs $x_1, \ldots, x_n$, and permutes them to obtain the sorted outputs $y_1, \ldots, y_n$. The circuit consists of a network of comparators connected by "wires". Each comparator has two inputs, u_1, u_2 and two outputs, v_1, v_2. The "upper" output, v_1, is the maximal value on the inputs, $u_1 \vee u_2$. The "lower" output, v_2, is the minimal value, $u_1 \wedge u_2$. In brief, naming the wires between the comparators as propositional variables, we can view a sorting network with inputs $x_1, \ldots, x_n$

* Supported by the G.I.F. grant 966-116.6 and by the Frankel Center for Computer Science at the Ben-Gurion University.

E.M. Clarke and A. Voronkov (Eds.): LPAR-16, LNAI 6355, pp. 154–172, 2010.

and outputs $y_1, \ldots, y_n$ as a propositional formula, ψ, obtained as the conjunction of its comparators. In the context of the Max-SAT problem described above the formula ψ_k expressing the cardinality constraint $x_1 + x_2 + \cdots + x_n < k$, is obtained by sorting the $x_1, \ldots, x_n$ (in decreasing order), and setting the k^{th} largest output to 0. As the outputs are sorted, this implies that all outputs from position k are zero and hence that there are less than k ones amongst the input values. If the outputs are $y_1, \ldots, y_n$, the cardinality constraint is expressed as $\psi_k = \psi \wedge \neg y_k$.

Adding a clause $\neg y_k$ to the formula ψ that represents a sorting network with input wires $x_1, \ldots, x_n$ and output wires $y_1, \ldots, y_n$, assigns a value (zero) to an output wire. This, in turn, imposes constraints on other input and output wires. So in some sense, we are running the network backwards. That sorting networks are bi-directional is well-understood, see for example [2]. However, this point is often overlooked. It is this bi-directionality that impacts the choice of sorting network construction best suited for application to cardinality constraints.

Sorting networks have been intensively studied since the mid 1960's. For an overview see for example, Knuth [5], or Parberry [6]. One of the best sorting network constructions, and possibly the one used most in applications, is due to Batcher as presented in [2]. Parberry [7] describes this network as follows:

> For all practical values of $n > 16$, the best known sorting network is the odd-even sorting network of Batcher, which is constructed recursively and has depth $(\log n)(\log n + 1)/2$ and size $n(\log n)(\log n - 1)/4 + n - 1$.

The presentations in [1] and in [4] describe the use of odd-even sorting networks to encode cardinality constraints. In this paper we take a similar approach but apply the so called *"pairwise sorting network"* instead. Parberry [7], introduces the pairwise network:

> It is the first sorting network to be competitive with the odd-even sort for all values of n. The value of the pairwise sorting network is not that it is superior to the odd-even sorting network in any sense, but that it is the first serious rival to appear in over 20 years.

In this paper, almost 20 years after the introduction of pairwise sorting networks, we are in the position to state that pairwise sorting networks are significantly superior to odd-even sorting networks, at least for encoding cardinality constraints. To obtain our results, we first observe that both types of networks are composed of two main components, which we term: *pairwise splitters* and *pairwise mergers*. Usually, the odd-even sorting network is viewed as a network of *odd-even mergers*. However, we show that each such odd-even merger is precisely equivalent to the composition of a pairwise splitter and a pairwise merger. Consequently, in the odd-even sorting network, pairwise splitters and mergers alternate, whilst in the pairwise network, the splitters and mergers occur in separate blocks, splitters before mergers.

The precise and clear presentation of the relationship between pairwise and odd-even sorting networks is new and is a contribution on its own right. It is also

the basis for our results. As we illustrate in the rest of the paper, the splitters inhibit the propagation of data from the sorted outputs $y_1, \ldots, y_n$ towards the original inputs $x_1, \ldots, x_n$ occurring in the cardinality constraint. Hence, when encoding cardinality constraints, the splitters are best positioned closer to the x_i's than to the y_i's.

Cardinality networks, similar to sorting networks, are networks of comparators that express cardinality constraints. As described above, a cardinality network is easily constructed using a sorting network. However, both the odd-even as well as the pairwise sorting networks involve $O(n \log^2 n)$ comparators to sort n bits. Earlier work describes cardinality networks of size $O(n \log^2 k)$ for constraints of the form $x_1 + x_2 + \cdots + x_n \prec k$ which is an improvement for the typical case where k is considerably smaller than n. For example, in [8], Wah and Chen illustrate a $O(n \log^2 k)$ construction for the (k, n) selection problem which is to select the k smallest (or largest) elements from a set of n numbers. Also in [1], Asin *et al.* define a simplified (odd-even) merge component and illustrate its application to construct a cardinality network of size $O(n \log^2 k)$.

In this paper we show that when expressing a cardinality constraint $x_1 + x_2 + \cdots + x_n \prec k$ in terms of a pairwise or an odd-even sorting network, the network collapses automatically (by partial evaluation) to a cardinality network with $O(n \log^2 k)$ comparators. No further construction is required. Experimental evaluation indicates that the choice of a pairiwse sorting network results in smaller encodings.

In Section 2 we present the classic odd-even and pairwise sorting networks. Section 3 clarifies a precise relationship between these two types of networks. This simplifies, and perhaps demystifies, the presentation of the pairwise network. In Section 4 we show that a cardinality network of size $O(n \log^2 k)$ is derived by partial evaluation from a pairwise sorting network. Section 5 presents a preliminary experimental evaluation and Section 6 concludes.

2 Sorting Networks – From Batcher to Parberry

Sorting networks were originally described as circuits, composed using comparators, which given values on their input positions compute (in parallel) the sorted values on their output positions. In the context of cardinality networks, it is beneficial to view sorting networks as relations between the "input" and "output" positions. Given such a relation, and values for some of the (input or output) positions, we seek values for the rest of the positions that satisfy the relation.

We represent a comparator as a relation $comparator(a, b, c, d)$ where intuitively a and b are two Boolean inputs and $\langle c, d \rangle$ a permutation of $\langle a, b \rangle$ with $c \geq d$. More formally, the relation is defined as follows:

$$comparator(a, b, c, d) \leftrightarrow (c \leftrightarrow a \vee b) \wedge (d \leftrightarrow a \wedge b)$$

A network of comparators is a conjunction of their corresponding relations. A sorting network is a relation on tuples of Boolean variables expressed in terms of a network of comparators. To ease presentation, we will assume that the lengths of tuples are powers of 2.

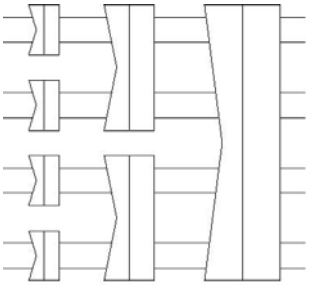


Fig. 1. An odd-even sorting network as a network of odd-even mergers

The Odd-Even Sorting Network

The odd-even sorting network, due to Batcher [2], is based on the merge-sort approach: to sort a list of $2n$ values, first partition the list into two lists with n values each, then recursively sort these two lists, and finally merge the two sorted lists. The network is termed "odd-even" because of the way the merge component is defined.

We present the odd-even merge component as a ternary relation on tuples of Boolean values. The relation $\mathsf{OEMerge}(A, B, C)$ is defined as a conjunction of comparators and expresses that merging sorted lists A and B, each of length n gives sorted list C, of length $2n$. The relation is defined for $n = 1$ and then recursively for $n > 1$ as follows:

$$\mathsf{OEMerge}(\langle a_1\rangle, \langle b_1\rangle, \langle c_1, c_2\rangle) \;\leftrightarrow\; \mathit{comparator}(a_1, b_1, c_1, c_2).$$

$$\begin{aligned}&\mathsf{OEMerge}(\langle a_1, \ldots, a_n\rangle, \langle b_1 \ldots, b_n\rangle, \langle d_1, c_2, \ldots, c_{2n-1}, e_n\rangle) \;\leftrightarrow\\ &\quad \mathsf{OEMerge}(\langle a_1, a_3 \ldots, a_{n-1}\rangle, \langle b_1, b_3 \ldots, b_{n-1}\rangle, \langle d_1, \ldots, d_n\rangle) \;\bigwedge\\ &\quad \mathsf{OEMerge}(\langle a_2, a_4 \ldots, a_n\rangle, \langle b_2, b_4 \ldots, b_n\rangle, \langle e_1, \ldots, e_n\rangle) \;\bigwedge\\ &\quad \textstyle\bigwedge_{i=1}^{n-1} \mathit{comparator}(e_i, d_{i+1}, c_{2i}, c_{2i+1}).\end{aligned}$$

The odd-even sorting network [2] is a binary relation on sequences of length 2^m and is defined as follows for $m = 0$ and recursively for $m > 0$. For $m > 0$ we denote the sequence length $2^m = 2n$.

$$\mathsf{OESort}(\langle a_1\rangle, \langle a_1\rangle).$$

$$\begin{aligned}&\mathsf{OESort}(\langle a_1, \ldots, a_{2n}\rangle, \langle c_1, \ldots, c_{2n}\rangle) \;\leftrightarrow\\ &\quad \mathsf{OESort}(\langle a_1, \ldots, a_n\rangle, \langle d_1, \ldots, d_n\rangle) \;\bigwedge\\ &\quad \mathsf{OESort}(\langle a_{n+1}, \ldots, a_{2n}\rangle, \langle d'_1, \ldots, d'_n\rangle) \;\bigwedge\\ &\quad \mathsf{OEMerge}(\langle d_1, \ldots, d_n\rangle, \langle d'_1, \ldots, d'_n\rangle, \langle c_1, \ldots, c_{2n}\rangle).\end{aligned}$$

The recursive definition of an odd-even sorting network unravels to a network of merge components. Figure 1 illustrates the network that defines the relation between 8 unsorted "inputs", on the left, and their 8 sorted "outputs", on the right. Each depicted odd-even merger represents a relation between two sorted sequences of length n (on its left side) and their sorted merge of length $2n$ (on its right side).

The Pairwise Sorting Network

The pairwise sorting network, due to Parberry [7], is also based on the merge-sort approach but with one simple, yet influential, difference in the first stage of the construction: To sort a list of $2n$ values, first split the list "pairwise" into two lists, the first with the n "larger" values from these pairs, and the second with the n smaller values. The resulting network is termed "pairwise" because of the way the elements to be sorted are pairwise split before recursively sorting the two parts and merging the resulting sorted lists.

$$\mathsf{PWSplit}(\langle a_1,\ldots,a_{2n}\rangle,\langle b_1,\ldots b_n\rangle,\langle c_1,\ldots,c_n\rangle) \quad \leftrightarrow \\ \bigwedge_{1\le i\le n} comparator(a_{2i-1},a_{2i},b_i,c_i).$$

The pairwise sorting network [7] is a binary relation on sequences of length 2^m and is defined as follows for $m = 0$ and recursively for $m > 0$. For $m > 0$ we denote the sequence length $2^m = 2n$.

$$\mathsf{PWSort}(\langle a_1\rangle,\langle a_1\rangle).$$

$$\begin{aligned}&\mathsf{PWSort}(\langle a_1,\ldots,a_{2n}\rangle,\langle d_1,\ldots,d_{2n}\rangle) \quad \leftrightarrow \\ &\quad \mathsf{PWSplit}(\langle a_1,\ldots,a_{2n}\rangle,\langle b_1,\ldots b_n\rangle,\langle c_1,\ldots,c_n\rangle) \quad \bigwedge \\ &\quad \mathsf{PWSort}(\langle b_1,\ldots,b_n\rangle,\langle b'_1,\ldots,b'_n\rangle) \quad \bigwedge \\ &\quad \mathsf{PWSort}(\langle c_1,\ldots,c_n\rangle,\langle c'_1,\ldots,c'_n\rangle) \quad \bigwedge \\ &\quad \mathsf{PWMerge}(\langle b'_1,\ldots,b'_n\rangle,\langle c'_1,\ldots,c'_n\rangle,\langle d_1,\ldots,d_{2n}\rangle).\end{aligned}$$

The description of the pairwise merger (PWMerge) given by Parberry in [7] is not straightforward to follow. We provide a simple specification of the PWMerge relation in the next section. For now, let us note a property of the pairwise merger by comparison to the odd-even merger. Consider the merging of two lists of bits, $\langle a_1,\ldots,a_n\rangle$ and $\langle b_1,\ldots,b_n\rangle$. The odd-even merger assumes only that the two lists are sorted. In contrast, the pairwise merger assumes also that each pair $\langle a_i,b_i\rangle$ is sorted "internally". Namely, that $a_i \ge b_i$. Indeed, in [7], the pairwise merger is called: "sorting sorted pairs".

The recursive definition of a pairwise sorting network unravels to a network of split and merge components. Figure 2 illustrates the network that defines the relation between 8 unsorted "inputs", on the left, and their 8 sorted "outputs",

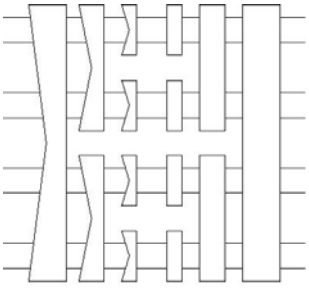


Fig. 2. A pairwise sorting network as a network of splitters and pairwise mergers

on the right. Scanning from left to right, we first have the network of splitters and then a network of mergers.

3 Sorting Networks – From Parberry Back to Batcher

In this section we clarify a precise relationship between the odd-even and pairwise sorting networks. While very simple, this relationship is not to be found in the literature. This relationship will provide the basis for our argument that pairwise sorting networks are significantly better than odd-even networks in the context of cardinality constraints. Parberry, in [7], provides little insight when stating that (1) *"one can prove by induction on n that the n input pairwise sorting network is not isomorphic to the odd-even sorting network"*; and (2) *"it is also easy to prove that the pairwise sorting network has the same size and depth bounds as Batcher's odd-even sorting network"*.

We first introduce a simple recursive definition for the pairwise merger and claim that it is indeed a suitable pairwise merger and that it has the same size and depth bounds as Parberry's pairwise merger.

Theorem 1 (pairwise merge). *Consider the following specification of a pairwise merger for merging sequences of length n defined for $n = 1$ and then recursively for $n > 1$.*

$$\mathsf{PWMerge}(\langle a_1\rangle, \langle b_1\rangle, \langle a_1, b_1\rangle).$$

$$\begin{aligned}\mathsf{PWMerge}(\langle a_1, \ldots, a_n\rangle, \langle b_1 \ldots, b_n\rangle, \langle d_1, c_2, \ldots, c_{2n-1}, e_n\rangle) \quad &\leftrightarrow \\ \mathsf{PWMerge}(\langle a_1, a_3 \ldots, a_{n-1}\rangle, \langle b_1, b_3 \ldots, b_{n-1}\rangle, \langle d_1, \ldots, d_n\rangle) \quad &\bigwedge \\ \mathsf{PWMerge}(\langle a_2, a_4 \ldots, a_n\rangle, \langle b_2, b_4 \ldots, b_n\rangle, \langle e_1, \ldots, e_n\rangle) \quad &\bigwedge \\ \textstyle\bigwedge_{i=1}^{n-1} comparator(e_i, d_{i+1}, c_{2i}, c_{2i+1}).\end{aligned}$$

This specification is: (1) a correct merger for the pairwise sorting network; (2) a network of depth $\log n$ *and size* $n \log n - n + 1$*; and (3) isomorphic to the iterative specification given by Parberry in [7] (pg.4).*

Proof. (sketch) For (1) we need to show that if $\langle a_1, \ldots, a_n\rangle$ and $\langle b_1 \ldots, b_n\rangle$ are sorted sequences and for each position $a_i \geq b_i$, then $\langle d_1, c_2, \ldots, c_{2n-1}, e_n\rangle$ is sorted (and has the same total number of 1's and 0's as in $\langle a_1, \ldots, a_n\rangle$ and $\langle b_1 \ldots, b_n\rangle$). This follows by a simple induction. Showing (2), is also straightforward, solving $S(1) = 0$ and $S(n) = 2S(n/2) + (n-1)$ for the size, and $D(1) = 0$, $D(n) = D(n/2) + 1$ for the depth. (3) Follows by induction on n, which we assume to be a power of 2. Assuming that the two recursive calls to PWMerge are isomorphic to their iterative specifications, we have that they each consist in $(\log_2 n - 1)$ layers generated by the iterations of Parberry's specification. Each respective pair of these layers (one from the odd call and one from the even call) are shown to combine to give a corresponding layer of the full network. The last iteration layer in the full network is precisely that introduced by the conjunction $\bigwedge_{i=1}^{n-1} comparator(e_i, d_{i+1}, c_{2i}, c_{2i+1})$.

We now proceed to observe the relationship between the pairwise and odd-even mergers.

Theorem 2 (odd-even merge). *An odd-even merger is equivalent to the composition of a pairwise split and a pairwise merger. For* $n \geq 1$,

$$\begin{aligned}&\mathsf{OEMerge}(\langle a_1,\ldots,a_n\rangle,\langle b_1\ldots,b_n\rangle,\langle c_1,\ldots,c_{2n}\rangle) \quad \leftrightarrow \\ &\quad \mathsf{PWSplit}(\langle a_1,b_1,a_2,b_2,\ldots,a_n,b_n\rangle,\langle a'_1,a'_2\ldots,a'_n\rangle,\langle b'_1,b'_2\ldots,b'_n\rangle) \quad \bigwedge \\ &\quad \mathsf{PWMerge}(\langle a'_1,a'_2\ldots,a'_n\rangle,\langle b'_1,b'_2\ldots,b'_n\rangle,\langle c_1,\ldots,c_{2n}\rangle).\end{aligned}$$

Proof. (brief description) The proof is by induction on n and follows the structure of the definition of OEMerge where the theorem holds for the recursive calls to OEMerge. (See Appendix.)

Figure 3 illustrates the relationship between the two types of sorting networks. There are three networks in the figure. On the left, we have a pairwise sorting network (of size 8). In the middle of this network, the column of four splitters (each of size 2) followed by the column of four pairwise mergers (also of size 2), form together a single column of four odd-even mergers, each of which is a 2×2 sorter. These 2×2 sorters are boxed in the left network. Now, the column of two splitters (each of size 4) can migrate to the right, because splitting before or after sorting has the same effect. This migration results in the middle network of the figure. In this middle network we now have, in the middle of the network, two 4×4 odd-even sorters. These 4×4 sorters are boxed in the middle network. The transition to the right network is once again by migrating a splitter over the two smaller sorting networks. This results in an 8×8 odd-even network. Figure 4 provides a high-level perspective on this transition. The left is a pairwise sorting network, composed of a split component followed by two recursive sorting networks and finishing with a pairwise merger. The splitter can be migrated to

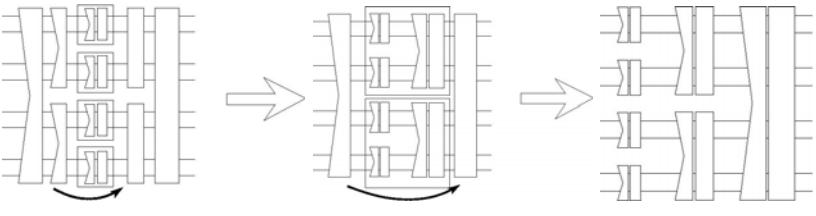


Fig. 3. Migration of splitters: from pairwise to odd-even

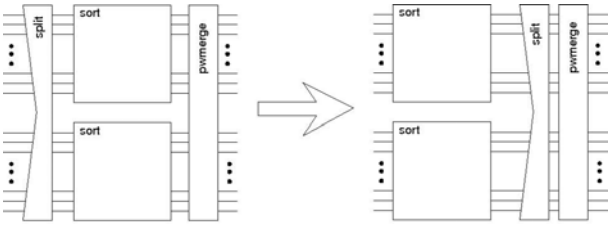


Fig. 4. High-level view on: from pairwise to odd-even

the right, as splitting and sorting (twice) has the same effect as (twice) sorting and then splitting.

In [4] and in [1], the authors prove that for the CNF encodings of cardinality constraints using odd-even sorting networks, unit propagation preserves arc consistency. This means that for a constraint of the form $x_1 + x_2 + \cdots + x_n < k$, as soon as $k-1$ of the x_i variables have become true, then the rest will become false by unit propagation. If $\langle x_1, \ldots, x_n \rangle$ and $\langle y_1, \ldots, y_n \rangle$ are the inputs and outputs of an odd-even sorting network, it means that setting any $k-1$ variables from $\langle x_1, \ldots, x_n \rangle$ the value 1, then by unit propagation the first $k-1$ variables in $\langle y_1, \ldots, y_n \rangle$ will be assigned the value 1. Moreover, if also $y_k = 0$ then the rest of the $n-k+1$ variables from $\langle x_1, \ldots, x_n \rangle$ will be assigned the value 0 by unit propagation. We note that pairwise sorting networks enjoy all of the same arc consistency properties as do the odd-even networks. The proofs of these claims are similar to those presented in [1]. We will see that pairwise networks enjoy one additional propagation property which does not hold for the odd-even networks.

4 The Pairwise Cardinality Network

In this section we show how the application of a pairwise sorting network to encode a cardinality constraint $x_1 + x_2 + \cdots + x_n \prec k$ can be collapsed to a network with $O(n \log^2 k)$ comparators. To obtain this result we first enhance the definition of the pairwise merger, adding a linear number of clauses which express that the outputs of the merger are sorted. These clauses are redundant in the sense that in any satisfying assignment of the formula representing a pairwise sorting network, the outputs of the mergers are sorted anyway. Their purpose is to amplify propagation. We focus on the case when $\prec$ is the less-than relation. We assume that n is a power of 2 and that $k \leq n/2 + 1$. Otherwise we can encode the dual constraint (counting the number of empty seats is easier than counting the passengers in an almost full aircraft). In general, it is common that k is significantly smaller than n and in the worst case $k = n/2+1$. The following definition specifies the enhanced pairwise merger.

Definition 1 (enhanced pairwise merger). *The enhanced pairwise merger is defined for* $n \geq 1$

$$\begin{aligned}&\mathsf{PWMerge}'(\langle a_1, \ldots, a_n \rangle, \langle b_1 \ldots, b_n \rangle, \langle c_1, c_2, \ldots, c_{2n} \rangle) \quad \leftrightarrow \\ &\qquad \mathsf{PWMerge}(\langle a_1, \ldots, a_n \rangle, \langle b_1 \ldots, b_n \rangle, \langle c_1, c_2, \ldots, c_{2n} \rangle) \quad \bigwedge \\ &\qquad \mathsf{sorted}(\langle c_1, c_2, \ldots, c_{2n} \rangle.\end{aligned}$$

where

$$\mathsf{sorted}(\langle c_1, \ldots, c_{2n} \rangle) \quad \leftrightarrow \quad \textstyle\bigwedge_{i=1}^{2n-1} (c_i \vee \neg c_{i+1}).$$

The pairwise cardinality network for $x_1 + x_2 + \cdots + x_n < k$ is obtained as a pairwise sorting network with inputs $x_1, \ldots, x_n$ and outputs $y_1, \ldots, y_n$. We assume that the pairwise mergers in the network are enhanced and we set the k^{th} output y_k to zero (by adding the clause $\neg y_k$). The network then collapses to a smaller network by propagating the known value $y_k = 0$ (backwards) through

the network. In particular, from the rightmost enhanced pairwise merger we have $\mathsf{sorted}(\langle y_1, \ldots, y_n \rangle)$ and obtain from $y_k = 0$ by unit propagation $y_i = 0$ for $k < i \leq n$.

The following definition specifies how comparators may be eliminated due to partially known inputs. It works like this (we focus on the propagation of zero's): For each comparator, if the upper output value is zero then we can remove the comparator, setting the other output bit to zero as well as the two input bits; and if either one of the input bits is zero, then we can remove the comparator setting the lower output bit to zero while the upper output bit is identified with the other input bit. The elimination of comparators is a simple form of partial evaluation and applied at the comparator level, before representing the network as a CNF formula. Except for the identification of an output bit with an input bit (e.g. $b = c$ in the definition below), it could also be performed as unit propagation at the CNF level.

Definition 2 (partial evaluation of comparators).

$$\begin{aligned} \mathit{comparator}(a,b,c,d) \wedge \neg c &\models_{pe} \neg a \wedge \neg b \wedge \neg d \\ \mathit{comparator}(a,b,c,d) \wedge \neg a &\models_{pe} \neg d \wedge \ (b = c) \\ \mathit{comparator}(a,b,c,d) \wedge \neg b &\models_{pe} \neg d \wedge \ (a = c) \end{aligned}$$

Figure 5 illustrates an 8 by 8 pairwise sorter and details the comparators in each of the components (as vertical lines between the wires). The figure highlights the rightmost merger

$$\mathsf{PWMerge}(\langle a_1, \ldots, a_4 \rangle, \langle b_1, \ldots, b_4 \rangle, \langle c_1, \ldots, c_8 \rangle)$$

To express the cardinality constraint $x_1 + x_2 + \cdots + x_8 < 4$, we set the output c_4 to 0. In the figure, applying partial evaluation, five of the nineteen comparators consequently become redundant (indicated by dashed lines) and can be removed. The reader is encouraged to check that in the corresponding odd-even network only 4 comparators can be eliminated.

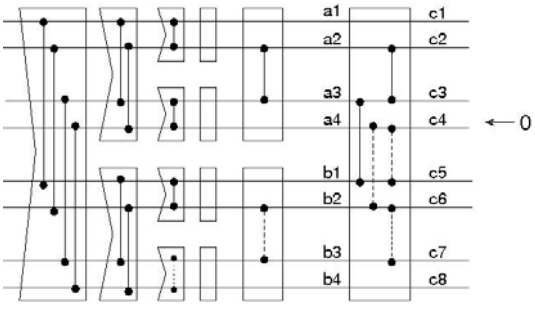


Fig. 5. A size 8 pairwise cardinality network (dashed comparators are redundant)

The next two theorems express a propagation property of the pairwise merger. First, consider the relation $\mathsf{PWMerge}(\langle a_1,\ldots,a_n\rangle,\langle b_1\ldots,b_n\rangle,\langle c_1,c_2,\ldots,c_{2n}\rangle)$ specified in Theorem 1. In the context of the pairwise sorting network, the inputs, upper $\langle a_1,\ldots,a_n\rangle$ and lower $\langle b_1\ldots,b_n\rangle$ to the merger, are both sorted as sequences as well as sorted pairwise. It means that (in any satisfying assignment) there are at least as many ones in the upper inputs as in the lower inputs. Hence, if there are less than k ones in the output sequence (namely, $c_k=0$), then there must be less than $\lceil k/2\rceil$ ones in the lower input sequence (namely $b_{\lceil k/2\rceil}=0$). This facilitates constraint propagation in the design of the pairwise cardinality network. For example, in Figure 5, The zero on output c_4 of the rightmost merger propagates to a zero on its lower input, b_2. This process continues as b_2 is also an output of the next merger (to the left).

The next theorem states that if there are less than k ones in the outputs of the merger, then it follows from the definition of the merger and partial evaluation alone that there are less than $\lceil k/2\rceil$ ones in the lower set of inputs. The consequence itself is obvious for each merger in the context of the sorting network (because in any satisfying assignment there are more ones in the upper set of inputs than in the lower). That it follows (independent of the context of the merger) by partial evaluation is not obvious and is useful below to simplify a pairwise network. Note that the claim is with regards to the non-enhanced pairwise merger. Namely it does not rely on the fact that the outputs of the merger are sorted.

Theorem 3. *For $k\le n$,*

$$\mathsf{PWMerge}\left(\begin{array}{l}\langle a_1,\ldots,a_n\rangle,\langle b_1\ldots,b_n\rangle,\\ \langle c_1,c_2,\ \ldots,\ c_{2n-1},c_{2n}\rangle\end{array}\right)\wedge\left(\bigwedge_{i=k}^{2n}\overline{c}_i\right)\quad\models_{pe}\quad\left(\bigwedge_{j=\lceil k/2\rceil}^{n}\overline{b}_j\right)$$

Proof. (brief description) The proof is by induction on n. It follows from the recursive definition of PWMerge focusing on the parity of k and of $\lceil k/2\rceil$. (See Appendix.)

The next theorem is similar. It states that if there are less than k ones in the outputs of the merger, then it follows from partial evaluation that for the smallest $k'\ge k$ which is a power of 2, the k'-th bit in the upper set of inputs is a zero. Once again, it is obvious that in the actual context of the merger in the sorting network, all of the inputs $a_k,\ldots,a_n$ will be set to zero. But the weaker result (for k') follows from the definition of the merger and partial evaluation alone.

Theorem 4. *Let $k\le n$ and let k' be the smallest power of 2 that is greater or equal to k. Then*

$$\mathsf{PWMerge}\left(\begin{array}{l}\langle a_1,\ldots,a_n\rangle,\langle b_1\ldots,b_n\rangle,\\ \langle c_1,c_2,\ \ldots,\ c_{2n-1},c_{2n}\rangle\end{array}\right)\wedge\left(\bigwedge_{i=k}^{2n}\overline{c}_i\right)\quad\models_{pe}\quad\overline{a}_{k'}$$

The proof is similar to that of Theorem 3 but with fewer cases due to the fact that k' is a power of 2.

Proof. (by induction on n) The base case for $n = 1$ is trivial. For $n > 1$ assume that the statement holds for all $n' < n$ and denote $k' = 2p$. We have

$$\left(\begin{array}{l}\bigwedge_{i=1}^{n-1} comparator(e_i, d_{i+1}, c_{2i}, c_{2i+1}) \\ \wedge\ (\bigwedge_{i=k}^{2n} \overline{c}_i)\end{array}\right) \quad \models_{pe} \quad \bigwedge_{i=p}^{n} \overline{e}_i \qquad (1)$$

The inductive hypothesis implies that the p^{th} element, $a_{2p} = a_{k'}$, of the sequence $\langle a_2, a_4, \ldots, a_n\rangle$ is zero. So we get $\overline{a}_{k'}$ as claimed.

It is important to note that Theorem 4 does not claim that all of the a_j with $j \geq k'$ become negated by partial evaluation. This does not hold. However, when using enhanced pairwise mergers we have $\mathsf{sorted}(\langle a_1, \ldots, a_n\rangle)$ from the corresponding merger where $\langle a_1, \ldots, a_n\rangle$ are outputs. This gives by unit propagation that $a_j = 0$ for $k' \leq j \leq n$.

The next theorem is inspired by the work presented in [1] where the authors observe, in the context of the $n \times n \rightarrow 2n$ odd-even sorting network, that if we are only interested in the $n+1$ largest elements of the output, the merger can be simplified to a network with two inputs, each of length n and an output of length $n + 1$. We show a similar result for the pairwise merger but emphasize that if we are only interested in the $n + 1$ largest outputs because $c_{n+1}{=}0, \ldots, c_{2n}{=}0$ (as in the context of a cardinality constraint with $k \leq n + 1$), then we obtain by partial evaluation a simplified merge network with $n + 1$ outputs in which the last output is surely a zero. We keep it in the presentation to simplify the proofs.

Theorem 5. *Consider the following specification for a simplified pairwise merger.*

$$\mathsf{SMerge}(\langle a_1\rangle, \langle b_1\rangle, \langle a_1, b_1\rangle).$$

$$\begin{array}{l}\mathsf{SMerge}(\langle a_1, \ldots, a_n\rangle, \langle b_1 \ldots, b_n\rangle, \langle d_1, c_2, \ldots, c_{n+1}\rangle) \quad \leftrightarrow \\ \quad \mathsf{SMerge}(\langle a_1, a_3 \ldots, a_{n-1}\rangle, \langle b_1, b_3 \ldots, b_{n-1}\rangle, \langle d_1, \ldots, d_{n/2+1}\rangle) \quad \bigwedge \\ \quad \mathsf{SMerge}(\langle a_2, a_4 \ldots, a_n\rangle, \langle b_2, b_4 \ldots, b_n\rangle, \langle e_1, \ldots, e_{n/2+1}\rangle) \quad \bigwedge \\ \quad \overline{e}_{n/2+1} \wedge \bigwedge_{i=1}^{n/2} comparator(e_i, d_{i+1}, c_{2i}, c_{2i+1}).\end{array}$$

Then,

$$\mathsf{PWMerge}\left(\begin{array}{l}\langle a_1, \ldots, a_n\rangle, \langle b_1 \ldots, b_n\rangle, \\ \langle c_1, c_2, \ \ldots, \ c_{2n-1}, c_{2n}\rangle\end{array}\right) \wedge \bigwedge_{i=n+1}^{2n} \overline{c}_i \quad \models_{pe} \quad \mathsf{SMerge}\left(\begin{array}{l}\langle a_1, \ldots, a_n\rangle, \\ \langle b_1 \ldots, b_n\rangle, \\ \langle c_1, \ldots, c_{n+1}\rangle\end{array}\right)$$

Proof. (by induction on n) For $n = 1$ there is nothing to prove. Let us observe the case, $n = 2$. We have from the definitions

$$\begin{array}{l}\mathsf{PWMerge}(\langle a_1, a_2\rangle, \langle b_1, b_2\rangle, \langle a_1, c_2, c_3, b_2\rangle) = comparator(a_2, b_1, c_2, c_3) \\ \quad \mathsf{SMerge}(\langle a_1, a_2\rangle, \langle b_1, b_2\rangle, \langle a_1, c_2, c_3\rangle) = \overline{b}_2 \wedge comparator(a_2, b_1, c_2, c_3)\end{array}$$

And the result holds:

$$\mathsf{PWMerge}\left(\langle a_1, a_2\rangle, \langle b_1, b_2\rangle, \langle a_1, c_2, c_3, b_2\rangle\right) \wedge \overline{b}_2 \quad \models_{pe} \quad \mathsf{SMerge}\left(\begin{array}{l}\langle a_1, a_2\rangle, \langle b_1, b_2\rangle, \\ \langle a_1, c_2, c_3\rangle\end{array}\right)$$

For the general case $n > 2$. Assume that $\bigwedge_{i=n+1}^{2n} \overline{c}_i$. Partial evaluation of the $comparator(e_i, d_{i+1}, c_{2i}, c_{2i+1})$ from the definition of PWMerge (for $n/2 + 1 \leq i \leq n-1$) gives $\bigwedge_{i=n/2+2}^{n} \overline{d}_i$ and $\bigwedge_{j=n/2+1}^{n} \overline{e}_j$, and from the same definition we obtain that $c_{2n} = e_n$. The remaining comparators from this part of the definition are

$$S_1 = \bigwedge_{i=1}^{i=n/2} \{comparator(e_i, d_{i+1}, c_{2i}, c_{2i+1})\}$$

Now, applying the inductive hypothesis on the odd and the even cases we get:

$$S_2 = \begin{pmatrix} \mathsf{SMerge}(\langle a_1, a_3 \ldots, a_{n-1}\rangle, \langle b_1, b_3 \ldots, b_{n-1}\rangle, \langle d_1, \ldots, d_{n/2+1}\rangle) & \bigwedge \\ \mathsf{SMerge}(\langle a_2, a_4 \ldots, a_n\rangle, \langle b_2, b_4 \ldots, b_n\rangle, \langle e_1, \ldots, e_{n/2+1}\rangle) \wedge \overline{e}_{n/2+1} & \end{pmatrix}$$

The result follows because $\mathsf{SMerge}(\langle a_1, \ldots, a_n\rangle, \langle b_1 \ldots, b_n\rangle, \langle c_1, \ldots, c_{n+1}\rangle) \leftrightarrow S_1 \wedge S_2$.

The next theorem complements the previous one. Consider

$$\mathsf{PWMerge}(\langle a_1, \ldots, a_n\rangle, \langle b_1 \ldots, b_n\rangle, \langle c_1, c_2, \ldots, c_{2n-1}, c_{2n}\rangle)$$

When negating the outputs $\langle c_k, \ldots, c_{2n}\rangle$ and assuming that $\langle a_1, \ldots, a_n\rangle$ are constrained to be sorted, the pairwise merger reduces by partial evaluation to a simplified merger, the size and depth of which depend exclusively on k. This is the key property that enables the construction of a cardinality network of size $O(n \log^2 k)$. Looking over a pairwise sorting network from outputs to inputs, we have a series of mergers followed by a series of splitters. If the first merger (from the outputs) has zeros from its k^{th} position then at the next level there are zeros from the k^{th} and from the $(k/2)^{th}$ positions. After $\log k$ levels, some of the mergers become trivial (zeros on all inputs and outputs).

Theorem 6. *Let $k \leq n$ and let k' be the smallest power of 2 that is greater or equal to k. Then*

$$\mathsf{PWMerge}\begin{pmatrix}\langle a_1, \ldots, a_n\rangle, \\ \langle b_1, \ldots, b_n\rangle, \\ \langle c_1, \ldots, c_{2n}\rangle\end{pmatrix} \wedge sorted(\langle a_1, \ldots, a_n\rangle) \wedge \bigwedge_{i=k}^{2n} \overline{c}_i \models_{pe} \mathsf{SMerge}\left(\langle a_1, \ldots, a_{k'}\rangle, \langle b_1, \ldots, b_{k'}\rangle, \langle c_1, \ldots c_{k'+1}\rangle\right)$$

Proof. (See Appendix.)

We are now in position to state the main theorem of the paper.

Theorem 7. *The pairwise cardinality network encoding a cardinality constraint $x_1 + x_2 + \cdots + x_n \prec k$ collapses by partial evaluation to a network with $O(n \log^2 k)$ comparators.*

Proof. (sketch) Construct an $n \times n$ pairwise sorting network. For simplicity, assume that k and n are powers of 2 and that $k \leq n/2$.

1. View the network like this: in the middle we have n/k sorting networks of size $k \times k$. These give a total size of $O(n/k * k \log^2 k) = O(n \log^2 k)$.

2. On the "right" of these $k \times k$ networks we have $1 + 2 + \cdots + \frac{n}{2k} = \frac{n}{k} - 1$ pairwise mergers, each of size $O(k \log k)$ after partial evaluation. This gives another $O(n \log k)$.
3. Now view the full sorting network. Let $c(n, k)$ denote the number of comparators in the split components of the network after partial evaluation originating from setting the k^{th} output to zero. That $c(n, k)$ is in $O(n \log^2 k)$ comes from the recurrence

$$c(n,k) = \begin{cases} 0 & \text{if } k = 1 \\ k \log^2 k & \text{if } n = k \text{ and } k > 1 \\ c(n/2,k) + c(n/2,k/2) + n/2 & \text{otherwise.} \end{cases}$$

That the pairwise cardinality network for n variables and bound k reduces by partial evaluation to a network with $O(n \log^2 k)$ comparators is theoretically pleasing. However, note that it is also possible to build the same reduced network directly using simplified pairwise mergers and propagating the bound k which is halved in each step for the lower recursively defined pairwise sorter.

Finally, we note that Theorem 7 holds also when encoding a cardinality network using an odd-even sorting network. The proof is based on the observation that an odd-even merger $\mathsf{OEMerge}(\langle a_1, \ldots, a_n \rangle, \langle b_1 \ldots, b_n \rangle, \langle c_1, \ldots, c_{2n} \rangle)$ where $c_k = 0$ simplifies to a network of size $O(k \log k)$.

5 A Preliminary Evaluation

This section describes a preliminary comparison of the use of odd-even and pairwise sorting networks for the applications involving cardinality constraints.

Table 1 shows some statistics regarding the size of the networks obtained from sorting networks after application of partial evaluation. Here size is measured counting number of comparators. Note that each comparator can be encoded using 6 CNF clauses, or alternatively, based on the technique proposed in [1] as a "half comparator" and encoded using only 3 clauses.

The first column in Table 1 indicates the size of the network, the second column indicates the number of comparators before application of unit propagation. The third column indicates the type of network considered, pairwise (**pw**)

Table 1. # of comparators for cardinality networks obtained via partial evaluation

n	full	Method	k=4	k=8	k=16	k=32	k=n/2
128	1471	pw	258	416	644	955	1248
		oe	315	572	879	1148	1335
256	3839	pw	515	812	1226	1841	3288
		oe	635	1164	1851	2532	3510
1024	24063	pw	2053	3143	4475	6425	20933
		oe	2555	4716	7683	11268	22260
2048	58367	pw	4102	6230	8680	12056	51130
		oe	5115	9452	15459	23048	54259

Table 2. Results for Boolean cardinality matrix problems (n, k_1, k_2) with $n = 100$

	Measure	Method	k_1=5	k_1=10	k_1=15	k_1=20	k_1=25
$k_2 = k_1 + 2$	cnf size (×1000)	pw	350	542	763	761	782
		oe	467	683	883	896	920
		card	473	665	760	878	914
	cpu time (sec.)	pw	6	19	27	89	270
		oe	23	1152	2395	946	1250
		card	41	382	2916	1110	832
$k_2 = k_1 + 3$	cnf size (×1000)	pw	350	554	763	766	782
		oe	467	697	883	901	920
		card	509	684	840	904	924
	cpu time (sec.)	pw	5	8	46	100	280
		oe	37	257	2583	1122	1546
		card	27	621	1798	1317	1158

or odd-even (**oe**). The next columns indicate the size of the network after unit propagation for various values of k. The last column considers the worst case with $k = n/2$. The table indicates that cardinality networks expressed using pairwise sorting networks are more amenable to partial evaluation.

Table 2 describes results when solving a Boolean cardinality matrix problem encoded using three techniques. An instance, (n, k_1, k_2), is to find values for the elements of an $n \times n$ matrix of Boolean variables where the cardinality of each row and column is between values k_1 and k_2. The table summarizes results for $n = 100$, for various values of k_1 with $k_2 = k_1 + 2$ and $k_2 = k_1 + 3$.

The first column indicates the value of k_2 in terms of k_1: $k_2 = k_1 + 2$ or $k_2 = k_1 + 3$. The second column indicates what is being measured: CNF size after partial evaluation (in 1000's of clauses) or CPU time for solving the problem (in seconds). We are running MiniSAT version 2 through its Prolog interface as described in [3]. The machine is a laptop running Linux with 2 Genuine Intel(R) CPUs, each 2GHz with 1GB RAM. The third column indicates the encoding method: using a pairwise cardinality network (based on a pairwise sorting network) (**pw**), or using an odd-even sorting network (**oe**), or using the cardinality network described in [1] (based on a construction built from a cascade of odd-even sorting networks). The next columns provide the data for various values of k_1. The results indicate a clear advantage for the use of pairwise sorting networks.

6 Summary and Conclusion

Sorting networks are often applied when encoding cardinality constraints. We argue the advantage in basing such encodings on the pairwise sorting network instead of on the odd-even sorting network as typically chosen, for example in [4] and in [1].

Our presentation clarifies the precise relationship between the pairwise network introduced in 1992 and the odd-even network from 1968. The simplicity

of this connection is surprising and perhaps demystifies the intuition underlying the pairwise network, which from 1992 is not referred to at all in the literature.

In contrast to previous works, such as [8] and [1], which encode cardinality constraints by application of specially constructed networks of comparators, our contribution is based directly on the (automatic) simplification of a sorting network. It is straightforward to apply this kind of simplification directly on the procedure that generates the network for $x_1 + x_2 + \cdots + x_n \prec k$, instead of first generating the $O(n \log^2 n)$ sorting network and then simplifying it to a network of size $O(n \log^2 k)$.

Acknowledgement. We thank the anonymous reviewers for useful comments on the earlier version of this paper.

References

1. Asín, R., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E.: Cardinality networks and their applications. In: Kullmann, O. (ed.) SAT 2009. LNCS, vol. 5584, pp. 167–180. Springer, Heidelberg (2009)
2. Batcher, K.E.: Sorting networks and their applications. In: AFIPS Spring Joint Computing Conference. AFIPS Conference Proceedings, vol. 32, pp. 307–314. Thomson Book Company, Washington (1968)
3. Codish, M., Lagoon, V., Stuckey, P.J.: Logic programming with satisfiability. Theory and Practice of Logic Programming 8(1), 121–128 (2008)
4. Eén, N., Sörensson, N.: Translating pseudo-boolean constraints into sat. JSAT 2(1-4), 1–26 (2006)
5. Knuth, D.E.: The Art of Computer Programming. Sorting and Searching, vol. III. Addison-Wesley, Reading (1973)
6. Parberry, I.: Parallel Complexity Theory. Research Notes in Theoretical Computer Science. Pitman Publishing, London (1987)
7. Parberry, I.: The pairwise sorting network. Parallel Processing Letters 2, 205–211 (1992)
8. Wah, B.W., Chen, K.-L.: A partitioning approach to the design of selection networks. IEEE Trans. Computers 33(3), 261–268 (1984)

A Appendix: Proof Sketches

Proof. (of Theorem 2) By induction on n. The base case, $n = 1$, follows directly from the definitions. For $n > 1$, assume that the theorem holds for all $n' < n$. By definition,

$$\begin{aligned}&\mathsf{OEMerge}(\langle a_1,\ldots,a_n\rangle,\langle b_1\ldots,b_n\rangle,\langle d_1,c_2,\ldots,c_{2n-1},e_n\rangle) \;\leftrightarrow\\ &\quad \mathsf{OEMerge}(\langle a_1,a_3\ldots,a_{n-1}\rangle,\langle b_1,b_3\ldots,b_{n-1}\rangle,\langle d_1,\ldots,d_n\rangle) \;\bigwedge\\ &\quad \mathsf{OEMerge}(\langle a_2,a_4\ldots,a_n\rangle,\langle b_2,b_4\ldots,b_n\rangle,\langle e_1,\ldots,e_n\rangle) \;\bigwedge\\ &\quad \textstyle\bigwedge_{i=1}^{n-1} comparator(e_i,d_{i+1},c_{2i},c_{2i+1})\end{aligned}$$

and the induction hypothesis holds for the recursive odd and even cases giving:

$$\begin{aligned}&\mathsf{OEMerge}(\langle a_1,a_3,\ldots,a_{n-1}\rangle,\langle b_1,b_3,\ldots,b_{n-1}\rangle,\langle d_1,\ldots,d_n\rangle) \;\leftrightarrow\\ &\quad \mathsf{PWSplit}\begin{pmatrix}\langle a_1,b_1,a_3,b_3,\ldots,a_{n-1},b_{n-1}\rangle\\ \langle a'_1,a'_3\ldots,a'_{n-1}\rangle,\langle b'_1,b'_3\ldots,b'_{n-1}\rangle\end{pmatrix}\wedge\\ &\quad \mathsf{PWMerge}(\langle a'_1,a'_3,\ldots,a'_{n-1}\rangle,\langle b'_1,b'_3,\ldots,b'_{n-1}\rangle,\langle d_1,\ldots,d_n\rangle)\\ &\mathsf{OEMerge}(\langle a_2,a_4,\ldots,a_n\rangle,\langle b_2,b_4,\ldots,b_n\rangle,\langle e_1,\ldots,e_n\rangle) \;\leftrightarrow\\ &\quad \mathsf{PWSplit}\begin{pmatrix}\langle a_2,b_2,a_4,b_4,\ldots,a_n,b_n\rangle,\\ \langle a'_2,a'_4\ldots,a'_n\rangle,\langle b'_2,b'_4\ldots,b'_n\rangle\end{pmatrix}\wedge\\ &\quad \mathsf{PWMerge}(\langle a'_2,a'_4,\ldots,a'_n\rangle,\langle b'_2,b'_4,\ldots,b'_n\rangle,\langle e_1,\ldots,e_n\rangle)\end{aligned}$$

substituting this in the definition (and rearranging the conjuncts) gives:

$$\begin{aligned}&\mathsf{OEMerge}(\langle a_1,\ldots,a_n\rangle,\langle b_1\ldots,b_n\rangle,\langle c_1,\ldots,c_{2n}\rangle) \;\leftrightarrow\\ &\quad \mathsf{PWSplit}\begin{pmatrix}\langle a_1,b_1,a_3,b_3,\ldots,a_{n-1},b_{n-1}\rangle\\ \langle a'_1,a'_3\ldots,a'_{n-1}\rangle,\langle b'_1,b'_3\ldots,b'_{n-1}\rangle\end{pmatrix}\wedge\\ &\quad \mathsf{PWSplit}\begin{pmatrix}\langle a_2,b_2,a_4,b_4,\ldots,a_n,b_n\rangle,\\ \langle a'_2,a'_4\ldots,a'_n\rangle,\langle b'_2,b'_4\ldots,b'_n\rangle\end{pmatrix}\wedge\\ &\quad \mathsf{PWMerge}(\langle a'_1,a'_3,\ldots,a'_{n-1}\rangle,\langle b'_1,b'_3,\ldots,b'_{n-1}\rangle,\langle d_1,\ldots,d_n\rangle)\wedge\\ &\quad \mathsf{PWMerge}(\langle a'_2,a'_4,\ldots,a'_n\rangle,\langle b'_2,b'_4,\ldots,b'_n\rangle,\langle e_1,\ldots,e_n\rangle)\wedge\\ &\quad \textstyle\bigwedge_{i=1}^{n-1} comparator(e_i,d_{i+1},c_{2i},c_{2i+1})\end{aligned}$$

which by definitions of PWSplit and PWMerge gives the required result.

$$\begin{aligned}&\mathsf{OEMerge}(\langle a_1,\ldots,a_n\rangle,\langle b_1\ldots,b_n\rangle,\langle c_1,\ldots,c_{2n}\rangle) \;\leftrightarrow\\ &\quad \mathsf{PWSplit}(\langle a_1,b_1,a_2,b_2,\ldots,a_n,b_n\rangle,\langle a'_1,a'_2\ldots,a'_n\rangle,\langle b'_1,b'_2\ldots,b'_n\rangle) \;\bigwedge\\ &\quad \mathsf{PWMerge}(\langle a'_1,a'_2\ldots,a'_n\rangle,\langle b'_1,b'_2\ldots,b'_n\rangle,\langle c_1,\ldots,c_{2n}\rangle)\end{aligned}$$

Proof. (of Theorem 3). By induction on n. The base case, $n = 1$, holds vacuously. For $n > 1$, assume that the statement holds for all $n' < n$ and consider the following cases according to the parities of k and $\lceil k/2\rceil$. Recall that,

$$\begin{aligned}&\mathsf{PWMerge}(\langle a_1,\ldots,a_n\rangle,\langle b_1\ldots,b_n\rangle,\langle d_1,c_2,\ldots,c_{2n-1},e_n\rangle) \;\leftrightarrow\\ &\quad \mathsf{PWMerge}(\langle a_1,a_3\ldots,a_{n-1}\rangle,\langle b_1,b_3\ldots,b_{n-1}\rangle,\langle d_1,\ldots,d_n\rangle) \;\bigwedge\\ &\quad \mathsf{PWMerge}(\langle a_2,a_4\ldots,a_n\rangle,\langle b_2,b_4\ldots,b_n\rangle,\langle e_1,\ldots,e_n\rangle) \;\bigwedge\\ &\quad \textstyle\bigwedge_{i=1}^{n-1} comparator(e_i,d_{i+1},c_{2i},c_{2i+1}).\end{aligned}$$

Let $\{b'_1,b'_2,\ldots,b'_{n/2}\}$, and $\{b''_1,b''_2,\ldots,b''_{n/2}\}$ be the the odd and the even subsequences of $\{b_1,b_2,\ldots,b_n\}$ respectively. We consider two cases depending on the parity of k.

k is even: Denote $k = 2p$. We have

$$\begin{pmatrix}\bigwedge_{i=1}^{n-1} comparator(e_i,d_{i+1},c_{2i},c_{2i+1})\\ \wedge\ (\bigwedge_{i=k}^{2n}\overline{c}_i)\end{pmatrix} \quad\models_{pe}\quad (\bigwedge_{i=p+1}^{n}\overline{d}_i)\wedge(\bigwedge_{j=p}^{n}\overline{e}_j) \qquad (2)$$

Now consider two subcases depending on the parity of p: (1) Assume $p = 2q$. From Equation (2) and the inductive hypothesis we get $\bigwedge_{i=q+1}^{n/2} \overline{b'}_i$ and $\bigwedge_{j=q}^{n/2} \overline{b''}_j$ which together imply that $\bigwedge_{i=k/2}^{n} \overline{b}_i$. (2) Assume $p = 2q+1$. From Equation (2) and the inductive hypothesis we get $\bigwedge_{i=q+1}^{n/2} \overline{b'}_i$ and $\bigwedge_{j=q+1}^{n/2} \overline{b''}_j$ which together imply that $\bigwedge_{i=p}^{n} \overline{b}_i$ or in terms of k that $\bigwedge_{i=k/2}^{n} \overline{b}_i$.
k is odd: Denote $k = 2p + 1$. We have

$$\left(\begin{array}{l} \bigwedge_{i=1}^{n-1} comparator(e_i, d_{i+1}, c_{2i}, c_{2i+1}) \\ \wedge\ (\bigwedge_{i=k}^{2n} \overline{c}_i) \end{array}\right) \quad \models_{pe} \quad (\bigwedge_{i=p+2}^{n} \overline{d}_i) \wedge (\bigwedge_{j=p+1}^{n} \overline{e}_j). \qquad (3)$$

Two subcases are considered for the parity of p: (1) Assume $p = 2q$. From Equation (3) and the inductive hypothesis we get $\bigwedge_{i=q+1}^{n/2} \overline{b'}_i$ and $\bigwedge_{j=q+1}^{n/2} \overline{b''}_j$. Therefore, $\bigwedge_{i=p+1}^{n} \overline{b}_i$ or $\bigwedge_{i=\lceil k/2 \rceil}^{n} \overline{b}_i$. (2) Assume $p = 2q+1$. From Equation (3) and the inductive hypothesis we get $\bigwedge_{i=q+2}^{n/2} \overline{b'}_i$ and $\bigwedge_{j=q+1}^{n/2} \overline{b''}_j$. Therefore, $\bigwedge_{i=p+1}^{n} \overline{b}_i$ or $\bigwedge_{i=\lceil k/2 \rceil}^{n} \overline{b}_i$. In all four subcases we have $\bigwedge_{i=\lceil k/2 \rceil}^{n} \overline{b}_i$ which proves the theorem.

To prove Theorem 6 we use the following lemmata

Lemma 1. *For $n > 1$,*

$$\mathsf{PWMerge}\left(\begin{array}{l}\langle a_1, \ldots, a_n\rangle, \\ \langle b_1 \ldots, b_n\rangle, \\ \langle c_1, \ldots, c_{2n}\rangle\end{array}\right) \wedge \bigwedge_{i=n/2+1}^{n} (\overline{a}_i \wedge \overline{b}_i) \models_{pe} \mathsf{PWMerge}\left(\begin{array}{l}\langle a_1, \ldots, a_{n/2}\rangle, \\ \langle b_1 \ldots, b_{n/2}\rangle, \\ \langle c_1, \ldots, c_n\rangle\end{array}\right) \wedge \bigwedge_{i=n+1}^{2n} \overline{c}_i$$

Proof. (by induction on n). For the base case $n = 2$ we have (by definition)

$$\mathsf{PWMerge}(\langle a_1, a_2\rangle, \langle b_1, b_2\rangle, \langle a_1, c_2, c_3, b_2\rangle) \leftrightarrow comparator(a_2, b_1, c_2, c_3)$$
$$\mathsf{PWMerge}(\langle a_1\rangle, \langle b_1\rangle, \langle a_1, b_1\rangle) \leftrightarrow true$$

We need to show that if a_2 and b_2 are negated then c_3 and b_2 become negated, and that $\langle a_1, b_1\rangle = \langle a_1, c_2\rangle$. Both facts follow because when $a_2 = 0$ the comparator gives by partial evaluation that $c_3 = 0$ and $b_1 = c_2$.

For the general case, we apply the inductive hypothesis **(*)** to the odd and the even cases in the definition of the pairwise merger. Giving respectively:

$$\mathsf{PWMerge}\left(\begin{array}{l}\langle a_1, a_3, \ldots, a_{n-1}\rangle, \\ \langle b_1, b_3 \ldots, b_{n-1}\rangle, \\ \langle d_1, \ldots, d_n\rangle\end{array}\right) \wedge \bigwedge_{i=n/4+1}^{n/2} (\overline{a}_{2i-1} \wedge \overline{b}_{2i-1}) \models_{pe}$$
$$\mathsf{PWMerge}\left(\begin{array}{l}\langle a_1, a_3, \ldots, a_{n/2-1}\rangle, \\ \langle b_1, b_3 \ldots, b_{n/2-1}\rangle, \\ \langle d_1, \ldots, d_{n/2}\rangle\end{array}\right) \wedge \bigwedge_{i=n/2+1}^{n} \overline{d}_i$$
$$\mathsf{PWMerge}\left(\begin{array}{l}\langle a_2, a_4, \ldots, a_n\rangle, \\ \langle b_2, b_4 \ldots, b_n\rangle, \\ \langle e_1, \ldots, e_n\rangle\end{array}\right) \wedge \bigwedge_{i=n/4+1}^{n/2} (\overline{a}_{2i} \wedge \overline{b}_{2i}) \models_{pe}$$
$$\mathsf{PWMerge}\left(\begin{array}{l}\langle a_2, a_4, \ldots, a_{n/2}\rangle, \\ \langle b_2, b_4 \ldots, b_{n/2}\rangle, \\ \langle e_1, \ldots, e_{n/2}\rangle\end{array}\right) \wedge \bigwedge_{i=n/2+1}^{n} \overline{e}_i$$

Several of the comparators from this application **(*)** are reduced by partial evaluation as follows:

$$\overline{e}_n \wedge \bigwedge_{i=n/2+1}^{n-1} \overline{e}_i \wedge \overline{d}_{i+1} \wedge comparator(e_i, d_{i+1}, c_{2i}, c_{2i+1}) \models_{pe} \bigwedge_{i=n+2}^{2n} \overline{c}_i$$

$$e_{n/2} \wedge \overline{d}_{n/2+1} \wedge \mathit{comparator}(e_{n/2}, d_{n/2+1}, c_n, c_{n+1}) \models_{pe} \overline{c}_{n+1}$$

Together this gives $\bigwedge_{i=n+1}^{2n} \overline{c}_i$, and now the required result follows directly from the definition of the pairwise merger.

Lemma 2. *Let $k \leq n$ and let k' be the smallest power of 2 which is greater or equal to k. Then,*

$$\mathsf{PWMerge}\begin{pmatrix}\langle a_1,\ldots,a_n\rangle,\\ \langle b_1\ldots,b_n\rangle,\\ \langle c_1,\ldots,c_{2n}\rangle\end{pmatrix} \wedge \textstyle\bigwedge_{i=k'+1}^{n} \overline{a}_i \wedge \overline{b}_i \models_{pe} \mathsf{PWMerge}\begin{pmatrix}\langle a_1,\ldots,a_{k'}\rangle,\\ \langle b_1\ldots,b_{k'}\rangle,\\ \langle c_1,\ldots,c_{2k'}\rangle\end{pmatrix} \wedge \textstyle\bigwedge_{i=2k'+1}^{2n} \overline{c}_i$$

Proof. The proof is by induction on n. For $n = k'$ it is trivial. For the general case, $n > k'$, we apply the induction hypothesis **(*)** to the odd and even cases in the definition of the pairwise merger. Note that each such application gives a pairwise merger of size k':

$$\mathsf{PWMerge}\begin{pmatrix}\langle a_1,a_3,\ldots,a_{n-1}\rangle,\\ \langle b_1,b_3\ldots,b_{n-1}\rangle,\\ \langle d_1,\ldots,d_n\rangle\end{pmatrix} \wedge \textstyle\bigwedge_{i=k'/2+1}^{n/2}(\overline{a}_{2i-1} \wedge \overline{b}_{2i-1}) \models_{pe}$$
$$\mathsf{PWMerge}\begin{pmatrix}\langle a_1,a_3,\ldots,a_{2k'-1}\rangle,\\ \langle b_1,b_3\ldots,b_{2k'-1}\rangle,\\ \langle d_1,\ldots,d_{2k'}\rangle\end{pmatrix} \wedge \textstyle\bigwedge_{i=2k'+1}^{n} \overline{d}_i$$
$$\mathsf{PWMerge}\begin{pmatrix}\langle a_2,a_4,\ldots,a_n\rangle,\\ \langle b_2,b_4\ldots,b_n\rangle,\\ \langle e_1,\ldots,e_n\rangle\end{pmatrix} \wedge \textstyle\bigwedge_{i=k'/2+1}^{n/2} \overline{a}_{2i} \wedge \overline{b}_{2i} \wedge \models_{pe}$$
$$\mathsf{PWMerge}\begin{pmatrix}\langle a_2,a_4,\ldots,a_{2k'}\rangle,\\ \langle b_2,b_4\ldots,b_{2k'}\rangle,\\ \langle e_1,\ldots,e_{2k'}\rangle\end{pmatrix} \wedge \textstyle\bigwedge_{i=2k'+1}^{n} \overline{e}_i$$

The application **(*)** of the pairwise merger definition introduces comparators which are reduced by partial evaluation:

$$\overline{e}_n \wedge \bigwedge_{i=2k'+1}^{n-1} \overline{e}_i \wedge \overline{d}_{i+1} \wedge \mathit{comparator}(e_i, d_{i+1}, c_{2i}, c_{2i+1}) \models_{pe} \bigwedge_{i=4k+2}^{2n} \overline{c}_i$$

$$e_{2k'} \wedge \overline{d}_{2k'+1} \wedge \mathit{comparator}(e_{2k'}, d_{2k'+1}, c_{4k'}, c_{4k'+1}) \models_{pe} \overline{c}_{4k'+1}$$

Together this gives $\bigwedge_{i=4k'+1}^{2n} \overline{c}_i$ and from the definition of the pairwise merger we get $\mathsf{PWMerge}(\langle a_1,\ldots,a_{2k'}\rangle, \langle b_1\ldots,b_{2k'}\rangle, \langle c_1,\ldots,c_{4k'}\rangle) \wedge \bigwedge_{i=4k'+1}^{2n} \overline{c}_i$ which together with the lemma statement and application of the previous lemma gives:
$\mathsf{PWMerge}(\langle a_1,\ldots,a_{2k'}\rangle, \langle b_1\ldots,b_{2k'}\rangle, \langle c_1,\ldots,c_{4k'}\rangle) \wedge \bigwedge_{i=k'+1}^{2k'} \overline{a}_i \wedge \overline{b}_i \models_{pe}$
$\mathsf{PWMerge}(\langle a_1,\ldots,a_{k'}\rangle, \langle b_1\ldots,b_{k'}\rangle, \langle c_1,\ldots,c_{2k'}\rangle) \wedge \bigwedge_{i=2k'+1}^{4k} \overline{c}_i$.

These two give us the required $\mathsf{PWMerge}(\langle a_1,\ldots,a_{k'}\rangle, \langle b_1\ldots,b_{k'}\rangle, \langle c_1,\ldots,c_{2k'}\rangle) \wedge \bigwedge_{i=2k'+1}^{2n} \overline{c}_i$.

Proof. (of Theorem 6) From Theorem 3 we have:

$$\mathsf{PWMerge}\begin{pmatrix}\langle a_1,\ldots,a_n\rangle, \langle b_1\ldots,b_n\rangle,\\ \langle c_1, c_2, \ \ldots, \ c_{2n-1}, c_{2n}\rangle\end{pmatrix} \wedge (\textstyle\bigwedge_{i=k'}^{2n} \overline{c}_i) \quad \models_{pe} \quad (\bigwedge_{j=k'/2}^{n} \overline{b}_j)$$

In particular we have $\bigwedge_{i=k'+1}^{n} \overline{b}_i$. In addition, from Theorem 4 we have:

$$\mathsf{PWMerge}\begin{pmatrix}\langle a_1,\ldots,a_n\rangle, \langle b_1\ldots,b_n\rangle,\\ \langle c_1, c_2, \ \ldots, \ c_{2n-1}, c_{2n}\rangle\end{pmatrix} \wedge (\textstyle\bigwedge_{i=k'}^{2n} \overline{c}_i) \quad \models_{pe} \quad \overline{a}_{k'}$$

and consequently that $\overline{a}_{k'} \wedge sorted(\langle a_1, \ldots, a_n \rangle) \models_{pe} \bigwedge_{i=k'+1}^{n} \overline{a}_i$. From lemma 2, and the above we obtain $\mathsf{PWMerge}(\langle a_1, \ldots, a_{k'} \rangle, \langle b_1, \ldots, b_{k'} \rangle, \langle c_1, \ldots, c_{2k'} \rangle)$. We are given that $\bigwedge_{i=k'+1}^{2k'} \overline{c}_i$. Hence, according to Theorem 5 we get $\mathsf{SMerge}(\langle a_1, \ldots, a_{k'} \rangle, \langle b_1, \ldots, b_{k'} \rangle, \langle c_1, \ldots c_{k'+1} \rangle)$.

B Appendix: Specifying Sorting Networks in Prolog

This appendix illustrates working Prolog specifications for the sorting network constructions presented in the paper. The code is simplified assuming that input/output sizes are powers of 2. Networks are represented as lists of comparators. Comparators are atoms of the form `comparator(A,B,C,D)` where `A,B` are the inputs and `C,D` are the outputs.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Construct a Batcher odd-even sorting network %
% (see 4th page of article)                    %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

oe_sort(As,Cs,Comparators) :-
        oe_sort(As,Cs,Comparators-[]).

oe_sort([A],[A],Cmp-Cmp) :- !.
oe_sort(As,Cs,Cmp1-Cmp4) :-
        split(As,As1,As2),
        oe_sort(As1,Ds1,Cmp1-Cmp2),
        oe_sort(As2,Ds2,Cmp2-Cmp3),
        oe_merge(Ds1,Ds2,Cs,Cmp3-Cmp4).

% merge two sorted sequences to a sorted sequence
oe_merge([A],[B],[C1,C2],
               [comparator(A,B,C1,C2)|Cmp]-Cmp) :- !.
oe_merge(As,Bs,[D|Cs],Cmp1-Cmp4) :-
        oddEven(As,AsOdd,AsEven),
        oddEven(Bs,BsOdd,BsEven),
        pw_merge(AsOdd,BsOdd,[D|Ds],Cmp1-Cmp2),
        pw_merge(AsEven,BsEven,Es,Cmp2-Cmp3),
        combine(Ds,Es,Cs,Cmp3-Cmp4).


% split down the middle
split(Xs,As,Bs) :-
        length(Xs,N), N1 is ceil(N/2),
        length(As,N1), append(As,Bs,Xs).


% split to odd and even
oddEven([Odd,Even|As],[Odd|Odds],[Even|Evens]) :-
        oddEven(As,Odds,Evens).
oddEven([],[],[]).


% combines the even and odd sorted elements
combine([],[B],[B],Cmp-Cmp).
combine([A|As],[B|Bs],[C1,C2|Cs],
                [comparator(A,B,C1,C2)|Cmp1]-Cmp2) :-
        combine(As,Bs,Cs,Cmp1-Cmp2).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Construct an (alternative) odd-even merger. %
% It is specified as the combination of a     %
% pairwise split and a pairwise merge         %
% (see Theorem 2)                             %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

alternative_oe_merge(As,Bs,Cs,Cmp1-Cmp3) :-
        interleave(As,Bs,ABs),
        pw_split(ABs,ABs1,ABs2,Cmp1-Cmp2),
        pw_merge(ABs1,ABs2,Cs,Cmp2-Cmp3).

interleave([],[],[]).
interleave([A|As],[B|Bs],[A,B|ABs]) :-
        interleave(As,Bs,ABs).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Construct a pairwise sorting network    %
% (see 5th page of article)               %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

pw_sort(As,Cs,Comparators) :-
        pw_sort(As,Cs,Comparators-[]).

pw_sort([A],[A],Cmp-Cmp) :- !.
pw_sort(As,Cs,Cmp1-Cmp5) :-
        pw_split(As,As1,As2,Cmp1-Cmp2),
        pw_sort(As1,Ds1,Cmp2-Cmp3),
        pw_sort(As2,Ds2,Cmp3-Cmp4),
        pw_merge(Ds1,Ds2,Cs,Cmp4-Cmp5).

% split pairs from a sequence to their larger and smaller elements
pw_split([],[],[],Cmp-Cmp).
pw_split([A1,A2|As],[B|Bs],[C|Cs],
                  [comparator(A1,A2,B,C)|Cmp1]-Cmp2) :-
        pw_split(As,Bs,Cs,Cmp1-Cmp2).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Construct a pairwise merger. It merges  %
% two sorted sequences of sorted pairs    %
% (see Theorem 1, page 5)                 %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

pw_merge([A],[B],[A,B], Cmp-Cmp) :-  !.
pw_merge(As, Bs, [D|Cs], Cmp1-Cmp4) :-
        oddEven(As,AsOdd,AsEven),
        oddEven(Bs,BsOdd,BsEven),
        pw_merge(AsOdd,BsOdd,[D|Ds],Cmp1-Cmp2),
        pw_merge(AsEven,BsEven,Es,Cmp2-Cmp3),
        combine(Ds,Es,Cs,Cmp3-Cmp4).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Construct a pairwise merger following the  %
% description from page 4 of [Parberry92].   %
% This is the network referred to in the     %
% proof of Theorem 2.                        %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

parberry_pw_merge(As,Bs,Cs,Ds,Cmp1-Cmp2) :-
        length(As,K),
        parberry_pw_merge_by_level(K,As,Bs,Cs,Ds,Cmp1-Cmp2).

parberry_pw_merge_by_level(1,As,Bs,As,Bs,Cmp-Cmp).
parberry_pw_merge_by_level(M,As,Bs,Cs,Ds,Cmp1-Cmp3) :-
        M>1, !, M1 is M//2,
        length(NewAs1,M1), append(NewAs1,As2,As),
        length(NewBs2,M1), append(Bs1,NewBs2,Bs),
        compare(Bs1,As2,NewBs1,NewAs2,Cmp1-Cmp2),
        append(NewAs1,NewAs2,NewAs), append(NewBs1,NewBs2,NewBs),
        parberry_pw_merge_by_level(M1,NewAs,NewBs,Cs,Ds,Cmp2-Cmp3).

compare([],[],[],[],Cmp-Cmp).
compare([A|As],[B|Bs],[C|Cs],[D|Ds],
                          [comparator(A,B,C,D)|Cmp1]-Cmp2) :-
        compare(As,Bs,Cs,Ds, Cmp1-Cmp2).
```

Logic and Computation in a Lambda Calculus with Intersection and Union Types

Daniel J. Dougherty[1] and Luigi Liquori[2]

[1] Worcester Polytechnic Institute, USA
[2] INRIA, France

Abstract. We present an explicitly typed lambda calculus "*à la* Church" based on the union and intersection types discipline; this system is the counterpart of the standard type assignment calculus "*à la* Curry." Our typed calculus enjoys the Subject Reduction and Church-Rosser properties, and typed terms are strongly normalizing when the universal type is omitted. Moreover both type checking and type reconstruction are decidable. In contrast to other typed calculi, a system with union types will fail to be "coherent" in the sense of Tannen, Coquand, Gunter, and Scedrov: different proofs of the same typing judgment will not necessarily have the same meaning. In response, we introduce a decidable notion of equality on type-assignment derivations inspired by the equational theory of bicartesian-closed categories.

1 Introduction

We address the problem of designing a λ-calculus *à la* Church corresponding to Curry-style type assignment to an untyped λ-calculus with intersection and union types [16,3]. In particular, we define a typed language such that its relationship with the intersection-union type assignment system fulfills the following *desiderata*: (i) typed and type assignment derivations are *isomorphic*, *i.e.*, the application of an *erasing function* on all typed terms and contexts (in a typed derivation judgment) produces a derivable type assignment derivation with the same structure, and every type assignment derivation is obtained from a typed one with the same structure by applying the same erasure; (ii) type checking and type reconstruction are decidable; (iii) reduction on typed terms has the same fundamental nice properties of reduction on terms receiving a type in the type-assignment system, such as confluence, preservation of typing under reduction, and strong normalization of terms typable without the universal type ω.

The challenges in defining such a calculus are already present in the context of intersection types, as evidenced by the polymorphic identity, with the following type-derivation in Curry style:

$$\dfrac{\dfrac{x{:}\sigma_1 \vdash x : \sigma_1}{\vdash \lambda x.x : \sigma_1 \to \sigma_1}\,(\to I) \qquad \dfrac{x{:}\sigma_2 \vdash x : \sigma_2}{\vdash \lambda x.x : \sigma_2 \to \sigma_2}\,(\to I)}{\vdash \lambda x.x : (\sigma_1 \to \sigma_1) \wedge (\sigma_2 \to \sigma_2)}\,(\wedge I)$$

E.M. Clarke and A. Voronkov (Eds.): LPAR-16, LNAI 6355, pp. 173–191, 2010.

This is untypable using a naïve corresponding rule *à la* Church for the introduction of intersection types:

$$\dfrac{\dfrac{x{:}\sigma_1 \vdash x : \sigma_1}{\vdash \lambda x{:}\sigma_1.x : \sigma_1 \to \sigma_1}\ (\to I) \qquad \dfrac{x{:}\sigma_2 \vdash x : \sigma_2}{\vdash \lambda x{:}\sigma_2.x : \sigma_2 \to \sigma_2}\ (\to I)}{\vdash \lambda x{:}\boxed{?}.x : (\sigma_1 \to \sigma_1) \wedge (\sigma_2 \to \sigma_2)}\ (\wedge I)$$

A solution to this problem was introduced in [15] where a calculus is designed whose terms comprise two parts, carrying computational and logical information respectively. The first component (the *marked-term*) is a simply typed λ-term, but types are variable-marks. The second component (the *proof-term*) records both the associations between variable-marks and types and the structure of the derivation. The technical tool for realizing this is an unusual formulation of context, which assigning types to term-variables *at a given mark/location*. The calculus of proof-terms can be seen as an encoding of a fragment of intuitionistic logic; it codifies a set of proofs that is strictly bigger than those corresponding to intersection type derivations (see [23]). There are other proposals in the literature for a λ-calculus typed with intersection types [17,20,4,26,22]. The languages proposed in these papers have been designed with various purposes, and they do not satisfy one or more of our desiderata above. A fuller discussion of this related work can be found in [15].

In this paper we extend the system of [15] to a calculus with union types. This is non-trivial, essentially because the standard typing rule for ∨-elimination is, as has been noted by many authors, so awkward. The difficulty manifests itself primarily in the (necessary) complexity of the definition of β-reduction on typed terms (see Section 5.2). On the other hand our solution exposes an interesting duality between the techniques required for intersections and for unions (Remark 5.2). Our typed reduction is well-behaved: it confluent, obeys subject reduction, and is strongly normalizing on terms typed with the universal type. But it must be taken on its own terms, not as a commentary on the untyped system.

Beyond solving the technical problem of extending the proof-term technique to handle union types, this paper makes a contribution to the study of the semantics of typed calculi viewed as foundations for typed programming language with unions, specifically to the investigation of *coherence*. In a typed programming language typing is an integral part of the semantics of a term. Indeed, the meaning of a typed term is not a function of the raw term but rather of the *typing judgment* of which the term is the subject. Reynolds [21] has dubbed this the *intrinsic* approach to semantics, as opposed to the *extrinsic* semantics given to terms in a type assignment system.

Now, in many type systems typing judgments can be derived in several ways, so the question of the relationship between the meanings of these judgments arises naturally. This question has been addressed in the literature in several settings, including languages with subtyping and generic operators [18], languages with subtyping and intersection types [19], and languages with polymorphism and recursive types [24,8]. The answer in all these cases has been, "all derivations of the same type judgment have the same meaning." Following [24] this phenomenon has come to be called *coherence*.

In the cited work judgments take their meaning in categories where intersections are modeled as categorical *products*: for a discussion of this point see [21] Section 16.6.

But coherence fails for a language with union types, if unions are modeled in the natural way as categorical coproducts. As a simple example, let σ be any type and consider the judgment $\vdash \lambda x.x : (\sigma \to \sigma) \vee (\sigma \to \sigma)$. There are obviously two derivations of this judgment, one corresponding to injection "from the left" and the other to injection "from the right." No reasonable semantics will equate these injections: it is an easy exercise to show that for any σ, if the two injections from σ to $(\sigma \vee \sigma)$ are equal, then any two arrows with source σ will be equal.

So the coherence question requires new analysis in the presence of union types. In this paper we reformulate the question as, "when are two different derivations of the same typing judgment equal?" (Cf. the discussion in [14], page 117, of the coherence problem for monoidal categories.) In Section 6 we show decidability of coherence under two important "type theories" (in the sense of [3]).

The failure of coherence has consequences for reduction of typed terms. In an intrinsic semantics the meaning of a term is a function of its type-derivation. Since reduction must, above all else, respect semantics, it follows that reduction should "respect" the type-derivation. When the language is coherent this is no constraint, and reduction can be defined purely in terms of the raw term that is the subject of the typing judgment. Thus, in typical typed calculi, reduction on typed terms is simply β-reduction, "ignoring the types." But in a system where coherence fails it is crucially important that reduction avoid the blunder of reducing a typed term and failing to preserve the semantics of the term's type-derivation. In the system presented in this paper this condition is reflected in the rather complex definition of reduction in Section 5 and in the fact that typed reduction can even "get stuck" relative to untyped reduction. For similar reasons the Subject Expansion property fails even though the type system has a universal type ω.

Some details have been omitted here for lack of space. Familiarity with [3] and with [15] will be helpful; the latter paper is a good source of examples restricted to the intersection types setting.

2 Intersection and Union Types

2.1 $\Lambda_u^{\wedge\vee}$: Curry-Style Type Assignment with Intersections and Unions

The set Λ is the set of untyped terms of the λ-calculus:

$$M ::= x \mid \lambda x.M \mid MM$$

We consider terms modulo α-conversion. Capture-avoiding substitution $M[N/x]$ of term N for variable x into term M is defined in the usual way. The reduction relation $\to_\beta$ is defined on untyped terms as the compatible closure of the relation $(\lambda x.M)N \to_\beta M[N/x]$.

Fix a set $\mathcal{V}$ of *type variables* and let ω be a distinguished *type constant.* The set $\mathcal{T}$ of *types* is generated from $\mathcal{V}$ and ω by the binary constructors $\to, \wedge$, and $\vee$. We use lowercase Greek letters to range over types.

Definition 1. *The Intersection-Union Type Assignment System* $\Lambda_{\mathrm{u}}^{\wedge\vee}$ *is the set of inference rules in Figure 1 for assigning intersection and union types to terms of the untyped* λ*-calculus.*

Let $B \triangleq \{x_1{:}\sigma_1, \ldots, x_n{:}\sigma_n\}$ ($i \neq j$ implies $x_i \not\equiv x_j$), and $B, x{:}\sigma \triangleq B \cup \{x{:}\sigma\}$

$$\frac{}{B \vdash M : \omega}\ (\omega) \qquad \frac{x{:}\sigma \in B}{B \vdash x : \sigma}\ (Var)$$

$$\frac{B, x{:}\sigma_1 \vdash M : \sigma_2}{B \vdash \lambda x.M : \sigma_1 \to \sigma_2}\ (\to I) \qquad \frac{B \vdash M : \sigma_1 \to \sigma_2 \quad B \vdash N : \sigma_1}{B \vdash MN : \sigma_2}\ (\to E)$$

$$\frac{B \vdash M : \sigma_1 \quad B \vdash M : \sigma_2}{B \vdash M : \sigma_1 \wedge \sigma_2}\ (\wedge I) \qquad \frac{B \vdash M : \sigma_1 \wedge \sigma_2 \quad i = 1,2}{B \vdash M : \sigma_i}\ (\wedge E_i)$$

$$\frac{B \vdash M : \sigma_i \quad i = 1,2}{B \vdash M : \sigma_1 \vee \sigma_2}\ (\vee I_i) \qquad \frac{B, x{:}\sigma_1 \vdash M : \sigma_3 \quad B, x{:}\sigma_2 \vdash M : \sigma_3 \quad B \vdash N : \sigma_1 \vee \sigma_2}{B \vdash M[N/x] : \sigma_3}\ (\vee E)$$

Fig. 1. The Intersection-Union Type Assignment System $\Lambda_{\mathrm{u}}^{\wedge\vee}$

Here are two crucial properties of the system $\Lambda_{\mathrm{u}}^{\wedge\vee}$.

Theorem 2. *[3]*

- *The terms typable without use of the* ω *rule are precisely the strongly normalizing terms.*
- *If* $B \vdash M : \sigma$ *and* $M \to_{gk} N$ *then* $B \vdash N : \sigma$*. Here* $\to_{gk}$ *is the well-known "Gross-Knuth" parallel reduction [13].*

2.2 $\Lambda_{\mathrm{t}}^{\wedge\vee}$: Church-Style Typing with Intersections and Unions

The key idea in the design of the intersection-union typed system is to split the term into two parts, carrying out the computational and the logical information respectively. Namely, the first one is a term of a typed λ-calculus, while the second one is a proof-term describing the shape of the type derivation.

The technical tool for connecting the two parts is an unusual formulation of contexts. In fact, a context associates to a variable both a variable-mark *and* a type, such that different variables are associated to different variable-marks.

2.3 The Proof-Term Calculus $\Lambda\mathcal{P}^{\wedge\vee}$

The terms of $\Lambda\mathcal{P}^{\wedge\vee}$ are encodings, via the Curry-Howard isomorphism, of the proofs of type-assignment derivations. The main peculiarity of this calculus is that it is defined

on another categories of variables called *variable-marks*; the calculus will be used to record the structure of a derivation through an association between variable-marks and types.

Definition 3. *Fix a set of variable-marks* ι. *The raw terms of* $\Lambda\mathcal{P}^{\wedge\vee}$ *are given as follows:*

$$\Delta ::= \iota \mid * \mid \lambda\iota{:}\sigma.\Delta \mid \Delta\Delta \mid \langle \Delta, \Delta \rangle \mid [\Delta, \Delta] \mid \mathsf{pr}_i\Delta \mid \mathsf{in}_i\Delta \qquad i = 1,2$$

The $\Lambda\mathcal{P}^{\wedge\vee}$ *calculus works modulo* α*-conversion (denoted by* $=_\alpha$*) defined as usual. Capture-avoiding substitution of the proof-term* Δ_2 *for variable* ι *in term* Δ_1 *is denoted* $\Delta_1[\Delta_2/\iota]$.

Definition 4. *The typing judgments for proof-terms* $\Lambda\mathcal{P}^{\wedge\vee}$ *are defined by the rules in Figure 2.*

Let $G \triangleq \{\iota_1{:}\sigma_1, \ldots, \iota_n{:}\sigma_n\}$ ($i \neq j$ implies $\iota_i \not\equiv \iota_j$), and $G, \iota{:}\sigma \triangleq G \cup \{\iota{:}\sigma\}$

$$\frac{}{G \vdash * : \omega}\ (\omega) \qquad \frac{\iota{:}\sigma \in G}{G \vdash \iota : \sigma}\ (Var)$$

$$\frac{G, \iota{:}\sigma_1 \vdash \Delta : \sigma_2}{G \vdash \lambda\iota{:}\sigma_1.\Delta : \sigma_1 \to \sigma_2}\ (\to I) \qquad \frac{G \vdash \Delta_1 : \sigma_1 \to \sigma_2 \quad G \vdash \Delta_2 : \sigma_1}{G \vdash \Delta_1\Delta_2 : \sigma_2}\ (\to E)$$

$$\frac{G \vdash \Delta_1 : \sigma_1 \quad G \vdash \Delta_2 : \sigma_2}{G \vdash \langle \Delta_1, \Delta_2 \rangle : \sigma_1 \wedge \sigma_2}\ (\wedge I) \qquad \frac{G \vdash \Delta : \sigma_1 \wedge \sigma_2 \quad i = 1,2}{G \vdash \mathsf{pr}_i\Delta : \sigma_i}\ (\wedge E_i)$$

$$\frac{G \vdash \Delta : \sigma_i \quad i = 1,2}{G \vdash \mathsf{in}_i\Delta : \sigma_1 \vee \sigma_2}\ (\vee I_i) \qquad \frac{G, \iota_1{:}\sigma_1 \vdash \Delta_1 : \sigma_3 \quad G, \iota_2{:}\sigma_2 \vdash \Delta_2 : \sigma_3 \quad G \vdash \Delta_3 : \sigma_1 \vee \sigma_2}{G \vdash [\lambda\iota_1{:}\sigma_1.\Delta_1, \lambda\iota_2{:}\sigma_2.\Delta_1]\Delta_3 : \sigma_3}\ (\vee E)$$

Fig. 2. The type system for the proof calculus $\Lambda\mathcal{P}^{\wedge\vee}$

Since $\Lambda\mathcal{P}^{\wedge\vee}$ is a simply-typed λ-calculus it can naturally be interpreted in cartesian closed categories. A term $[\Delta_1, \Delta_2]$ corresponds to the "co-pairing" of two arrows Δ_i to build an arrow out of a coproduct type. Then the term $[\lambda\iota_1{:}\sigma_1.\Delta_1, \lambda\iota_2{:}\sigma_2.\Delta_1]\Delta_3$ corresponds to the familiar case statement.

The type ω plays the role of a terminal object, that is to say it is an object with a single element. The connection with type-assignment is this: every term can be assigned type ω so all "proofs" of that judgment have no content: all these proofs are considered identical ([21], page 372). It is typical to name the unique element of the terminal object as $*$. This explains the typing rule for $*$ in Figure 2.

There is a natural equality theory for the terms $\Lambda\mathcal{P}^{\wedge\vee}$; we record it now and will return to it in Section 5.

Definition 5. *The equational theory $\cong$ on proof-terms is defined by the following axioms (we assume that in each equation the two sides have the same type).*

$$(\lambda\iota{:}\sigma.\Delta_1)\,\Delta_2 = \Delta_1[\Delta_2/\iota] \tag{1}$$

$$\mathsf{pr}_i\langle\Delta_1\,,\,\Delta_2\rangle = \Delta_i \qquad i=1,2 \tag{2}$$

$$[\lambda\iota_1{:}\sigma_1.\Delta_1\,,\,\lambda\iota_2{:}\sigma_2.\Delta_2]\,\mathsf{in}_i\Delta = \Delta_i[\Delta/\iota] \qquad i=1,2 \tag{3}$$

$$\lambda\iota{:}\sigma_1.\Delta\iota = \Delta \qquad \iota\notin \mathsf{Fv}(\Delta) \tag{4}$$

$$\langle\mathsf{pr}_1\Delta,\,\mathsf{pr}_2\Delta\rangle = \Delta \tag{5}$$

$$[\lambda\iota{:}\sigma_1.\Delta(\mathsf{in}_1\iota)\,,\,\lambda\iota{:}\sigma_2.\Delta(\mathsf{in}_2\iota)](\iota) = \Delta(\iota) \tag{6}$$

$$\Delta = * \qquad \textit{at type } \omega \tag{7}$$

The first three equations are the familiar "computational" axioms for the arrow, product, and sum data-types. The next four equation capture various "uniqueness" criteria which induce the $\wedge$, $\vee$, and ω type constructors to behave as categorical products, coproducts, and terminal object. The terminal type acts as an empty product; in terms of the proof theory this corresponds to saying that ω admits a unique proof, and is reflected in Equation 7, which says that all proofs of type ω are equal to $*$.

2.4 Typed Terms with Intersections and Unions

Definition 6. *Fix a set of variable-marks* ι. *The set of* marked-terms *are given as follows:*

$$M ::= x \mid \lambda x{:}\iota.M \mid MM$$

The set of $\Lambda_t^{\wedge\vee}$ *of* typed terms *is the set of expressions* $M@\Delta$ *where* M *is a marked-term and* Δ *is a proof-term.*

As usual we consider terms modulo renaming of bound variables. Formally this is defined via the notion of α-conversion, which requires some extra care in our setting, so we give the definition explicitly:

Definition 7 (α-conversion). *The α-conversion, denoted by $=_\alpha$, on well formed terms can be defined as the symmetric, transitive, reflexive, and contextual closure of :*

$$(\lambda x{:}\iota.M)@\Delta \to_\alpha (\lambda y{:}\iota.M[y/x])@\Delta \qquad y \textit{ fresh in } M$$

$$M@(\lambda\iota_1{:}\sigma.\Delta) \to_\alpha M[\iota_2/\iota_1]@(\lambda\iota_2{:}\sigma.\Delta[\iota_2/\iota_1]) \qquad \iota_2 \textit{ fresh in } \Delta$$

Definition 8 (Church-style typing). *The typing rules are presented in Figure 3. The system proves judgments of the shape* $\Gamma \vdash M@\Delta : \sigma$, *where* Γ *is a context and* $M@\Delta$ *is a typed term.*

Intuitively: in the judgment, the type-context Γ assigns union types to the free-variables of M annotated by variable-marks; if $\Gamma \vdash M@\Delta : \sigma$, then we say that $M@\Delta$ is a term of $\Lambda_t^{\wedge\vee}$. The proof-term keeps track of the type of the used mark together with a trace of

Let $\Gamma \triangleq \{x_1@\iota_1{:}\sigma_1, \ldots, x_n@\iota_2{:}\sigma_n\}$ ($i \neq j$ implies $x_i \not\equiv x_j$), and $\Gamma, x@\iota{:}\sigma \triangleq \Gamma \cup \{x@\iota{:}\sigma\}$

$$\frac{}{\Gamma \vdash M@* : \omega}\ (\omega) \qquad \frac{x@\iota{:}\sigma \in \Gamma}{\Gamma \vdash x@\iota : \sigma}\ (Var)$$

$$\frac{\Gamma, x@\iota{:}\sigma_1 \vdash M@\Delta : \sigma_2}{\Gamma \vdash \lambda x{:}\iota.M@\lambda\iota{:}\sigma_1.\Delta : \sigma_1 \to \sigma_2}\ (\to I) \qquad \frac{\Gamma \vdash M@\Delta_1 : \sigma_1 \to \sigma_2 \quad \Gamma \vdash N@\Delta_2 : \sigma_1}{\Gamma \vdash MN@\Delta_1\Delta_2 : \sigma_2}\ (\to E)$$

$$\frac{\Gamma \vdash M@\Delta_1 : \sigma_1 \quad \Gamma \vdash M@\Delta_2 : \sigma_2}{\Gamma \vdash M@\langle \Delta_1 , \Delta_2 \rangle : \sigma_1 \wedge \sigma_2}\ (\wedge I) \qquad \frac{\Gamma \vdash M@\Delta : \sigma_1 \wedge \sigma_2 \quad i = 1,2}{\Gamma \vdash M@\mathsf{pr}_i\Delta : \sigma_i}\ (\wedge E_i)$$

$$\frac{\Gamma \vdash M@\Delta : \sigma_i \quad i = 1,2}{\Gamma \vdash M@\mathsf{in}_i\Delta : \sigma_1 \vee \sigma_2}\ (\vee I_i) \qquad \frac{\begin{array}{l}\Gamma, x@\iota_1{:}\sigma_1 \vdash M@\Delta_1 : \sigma_3 \\ \Gamma, x@\iota_2{:}\sigma_2 \vdash M@\Delta_2 : \sigma_3 \quad \Gamma \vdash N@\Delta_3 : \sigma_1 \vee \sigma_2\end{array}}{\Gamma \vdash M[N/x]@[\lambda\iota_1{:}\sigma_1.\Delta_1 , \lambda\iota_2{:}\sigma_2.\Delta_2]\Delta_3 : \sigma_3}\ (\vee E)$$

Fig. 3. The type system for the typed calculus $\Lambda_t^{\wedge\vee}$

the *skeleton* of the derivation tree. The proof-term Δ plays the role of a road map to backtrack (*i.e.* roll back) the derivation tree.

2.5 Example of Typing for $\Lambda_t^{\wedge\vee}$

The reader will find a good number of examples showing some typing in the intersection type system in [15]. As an example of the present system using intersection and union types in an essential way, we treat the example (due to Pierce) that shows the failure of subject reduction for simple, non parallel, reduction in [3]. Let I denote the identity. Then, the untyped (parallel) reduction is: $x(\mathsf{I}(yz))(\mathsf{I}(yz)) \Rightarrow_\beta x(yz)(yz)$. Under the type context $B \triangleq x{:}(\sigma_1 \to \sigma_1 \to \tau) \wedge (\sigma_2 \to \sigma_2 \to \tau), y{:}\rho \to \sigma_1 \vee \sigma_2, z : \rho$, the redex can be typed as follows (the derivation for the reductum being simpler):

$$\dfrac{\dfrac{\dfrac{B, w{:}\sigma_1 \vdash x : \sigma_1 \to \sigma_1 \to \tau \quad B, w{:}\sigma_2 \vdash x : \sigma_2 \to \sigma_2 \to \tau}{B, w{:}\sigma_1 \vdash w : \sigma_1 \quad B, w{:}\sigma_2 \vdash w : \sigma_2} \quad \dfrac{B \vdash \mathsf{I} : \sigma_1 \vee \sigma_2 \to \sigma_1 \vee \sigma_2}{B \vdash yz : \sigma_1 \vee \sigma_2}}{B, w{:}\sigma_1 \vdash xw : \sigma_1 \to \tau \quad B, w{:}\sigma_2 \vdash xw : \sigma_2 \to \tau \quad B \vdash \mathsf{I}(yz) : \sigma_1 \vee \sigma_2}}{B, w{:}\sigma_1 \vdash xww : \tau \quad B, w{:}\sigma_2 \vdash xww : \tau}\ (\vee E)$$

$$B \vdash x(\mathsf{I}(yz))(\mathsf{I}(yz)) : \tau$$

We look now for the corresponding typed derivations. The corresponding typed term of $x(\mathsf{I}(yz))(\mathsf{I}(yz))$ is

$$x(\underbrace{(\lambda v{:}\iota_3.v)}_{\mathsf{I}_t}(yz))(\underbrace{(\lambda v{:}\iota_3.v)}_{\mathsf{I}_t}(yz))@[\underbrace{\lambda\iota_1{:}\sigma_1.(\mathsf{pr}_1\iota)\iota_1\iota_1}_{\Delta_1}, \underbrace{\lambda\iota_2{:}\sigma_2.(\mathsf{pr}_2\iota)\iota_2\iota_2}_{\Delta_2}](\underbrace{(\lambda\iota_3{:}\sigma_1 \vee \sigma_2.\iota_3)}_{\Delta_3}(\iota_4\iota_5))$$

Under the type context $\Gamma \triangleq x@\iota{:}(\sigma_1 \to \sigma_1 \to \tau) \wedge (\sigma_2 \to \sigma_2 \to \tau), y@\iota_4{:}\rho \to \sigma_1 \vee \sigma_2, z@\iota_5 : \rho$, and $\Gamma_1 = \Gamma, w@\iota_1{:}\sigma_1$ and $\Gamma_2 = \Gamma, w@\iota_2{:}\sigma_2$, the above term can be typed as follows:

$$\dfrac{\dfrac{\dfrac{\Gamma_1 \vdash x@\mathsf{pr}_1\iota : \sigma_1 \to \sigma_1 \to \tau \quad \Gamma_1 \vdash w@\iota_1 : \sigma_1}{\Gamma_1 \vdash xw@(\mathsf{pr}_1\iota)\iota_1 : \sigma_1 \to \tau} \quad \Gamma_1 \vdash w@\iota_1 : \sigma_1}{\Gamma_1 \vdash xww@(\mathsf{pr}_1\iota)\iota_1\iota_1 : \tau} \quad \dfrac{\dfrac{\Gamma_2 \vdash x@\mathsf{pr}_2\iota : \sigma_2 \to \sigma_2 \to \tau \quad \Gamma_2 \vdash w@\iota_2 : \sigma_2}{\Gamma_2 \vdash xw@(\mathsf{pr}_2\iota)\iota_2 : \sigma_2 \to \tau} \quad \Gamma_2 \vdash w@\iota_2 : \sigma_2}{\Gamma_2 \vdash xww@(\mathsf{pr}_2\iota)\iota_2\iota_2 : \tau} \quad \dfrac{\dfrac{\Gamma \vdash \mathsf{I}_\mathsf{t}@\Delta_3 : \sigma_1 \vee \sigma_2 \to \sigma_1 \vee \sigma_2 \quad \Gamma \vdash yz@\iota_4\iota_5 : \sigma_1 \vee \sigma_2}{\Gamma \vdash \mathsf{I}_\mathsf{t}\,(yz)@(\Delta_3\,(\iota_4\iota_5)) : \sigma_1 \vee \sigma_2}}{\Gamma \vdash x\,(\mathsf{I}_\mathsf{t}\,(yz))\,(\mathsf{I}_\mathsf{t}\,(yz))@[\Delta_1\,,\,\Delta_2]\,(\Delta_3\,(\iota_4\iota_5)) : \tau}$$

3 The Isomorphism between $\Lambda_\mathsf{u}^{\wedge\vee}$ and $\Lambda_\mathsf{t}^{\wedge\vee}$

In this section we prove that the type system for $\Lambda_\mathsf{t}^{\wedge\vee}$ is isomorphic to the classical system for $\Lambda_\mathsf{u}^{\wedge\vee}$ of [3]. The isomorphism is given for a customization of the general definition of isomorphism given in [25], to the case of union types and proof-terms.

From the logical point of view, the existence of an isomorphism means that there is a one-to-one correspondence between the judgments that can be proved in the two systems, and the derivations correspond with each other rule by rule. In what follows, and with a little abuse of notation, marked-terms and untyped terms of the λ-calculus will be ranged over by $M, N, \ldots$, the difference between marked-terms and untyped-terms being clear from the context (*i.e.* the judgment to be proved).

Definition 9 (Church *vs.* Curry).

1. *The type-erasing function* $\mathcal{E} : \Lambda_\mathsf{t}^{\wedge\vee} \Rightarrow \Lambda$ *is inductively defined on terms as follows:*

$$\mathcal{E}(x@\_) \triangleq x \qquad \mathcal{E}(\lambda x{:}\iota.M@\_) \triangleq \lambda x.\mathcal{E}(M@\_) \qquad \mathcal{E}(MN@\_) \triangleq \mathcal{E}(M@\_)\,\mathcal{E}(N@\_)$$

   $\mathcal{E}$ is pointwise extended to contexts in the obvious way.
2. *Let $\mathcal{D}er\Lambda_\mathsf{u}^{\wedge}$ and $\mathcal{D}er\Lambda_\mathsf{t}^{\wedge}$ be the sets of all (un)typed derivations, and let $\mathcal{D}^\mathsf{u}$, and $\mathcal{D}^\mathsf{u}$ denote (un)typed derivations, respectively. The functions $\mathcal{F} : \mathcal{D}er\Lambda_\mathsf{t}^{\wedge} \Rightarrow \mathcal{D}er\Lambda_\mathsf{u}^{\wedge}$ and $\mathcal{G} : \mathcal{D}er\Lambda_\mathsf{u}^{\wedge} \Rightarrow \mathcal{D}er\Lambda_\mathsf{t}^{\wedge}$ are indicated in Figures 4 and 5.*

Theorem 10 (Isomorphism). *The systems $\Lambda_\mathsf{t}^{\wedge\vee}$ and $\Lambda_\mathsf{u}^{\wedge\vee}$ are isomorphic in the following sense. $\mathcal{F} \circ \mathcal{G}$ is the identity in $\mathcal{D}er\Lambda_\mathsf{u}^{\wedge}$ and $\mathcal{G} \circ \mathcal{F}$ is the identity in $\mathcal{D}er\Lambda_\mathsf{t}^{\wedge}$ modulo uniform naming of variable-marks. I.e.,*

$$\mathcal{G}(\mathcal{F}(\Gamma \vdash M@\Delta : \sigma)) = \mathsf{ren}(\Gamma) \vdash \mathsf{ren}(M@\Delta) : \sigma$$

where ren *is a simple function renaming the free occurrences of variable-marks.*

Proof. By induction on the structure of derivations.

$$\mathcal{F}\left(\frac{x@\iota{:}\sigma \in \Gamma}{\Gamma \vdash x@\iota : \sigma}\ (Var)\right) \triangleq \begin{cases} \dfrac{x{:}\sigma \in B}{B \vdash x : \sigma}\ (Var) \\ \mathcal{E}(\Gamma) = B \end{cases}$$

$$\mathcal{F}\left(\frac{}{\Gamma \vdash M@* : \omega}\ (\omega)\right) \triangleq \begin{cases} \dfrac{}{B \vdash M : \omega}\ (\omega) \\ \mathcal{E}(\Gamma) = B \end{cases}$$

$$\mathcal{F}\left(\frac{\mathcal{D}^{\mathrm{t}} : \Gamma, x@\iota{:}\sigma_1 \vdash M@\Delta : \sigma_2}{\Gamma \vdash \lambda x{:}\iota.M@\lambda\iota{:}\sigma_1.\Delta : \sigma_1 \to \sigma_2}\ (\to I)\right) \triangleq \begin{cases} \dfrac{\mathcal{F}(\mathcal{D}^{\mathrm{t}}) : B, x{:}\sigma_1 \vdash M' : \sigma_2}{B \vdash \lambda x.M' : \sigma_1 \to \sigma_2}\ (\to I) \\ \mathcal{E}(\Gamma, x@\iota{:}\sigma_1) = B, x{:}\sigma_1 \ \&\ \mathcal{E}(M@\Delta) = M' \end{cases}$$

$$\mathcal{F}\left(\frac{\mathcal{D}_1^{\mathrm{t}} : \Gamma \vdash M@\Delta_1 : \sigma_1 \to \sigma_2 \quad \mathcal{D}_2^{\mathrm{t}} : \Gamma \vdash N@\Delta_2 : \sigma_1}{\Gamma \vdash MN@\Delta_1\,\Delta_2 : \sigma_2}\ (\to E)\right) \triangleq \begin{cases} \dfrac{\mathcal{F}(\mathcal{D}_1^{\mathrm{t}}) : B \vdash M' : \sigma_1 \to \sigma_2 \quad \mathcal{F}(\mathcal{D}_2^{\mathrm{t}}) : B \vdash N' : \sigma_1}{B \vdash M'N' : \sigma_2}\ (\to E) \\ \mathcal{E}(\Gamma) = B \ \&\ \mathcal{E}(M@\Delta_1) = M' \ \&\ \mathcal{E}(N@\Delta_2) = N' \end{cases}$$

$$\mathcal{F}\left(\frac{\mathcal{D}_1^{\mathrm{t}} : \Gamma \vdash M@\Delta_1 : \sigma_1 \quad \mathcal{D}_2^{\mathrm{t}} : \Gamma \vdash M@\Delta_2 : \sigma_2}{\Gamma \vdash M@\langle \Delta_1\,,\, \Delta_2 \rangle : \sigma_1 \wedge \sigma_2}\ (\wedge I)\right) \triangleq \begin{cases} \dfrac{\mathcal{F}(\mathcal{D}_1^{\mathrm{t}}) : B \vdash M' : \sigma_1 \quad \mathcal{F}(\mathcal{D}_2^{\mathrm{t}}) : B \vdash M' : \sigma_2}{B \vdash M' : \sigma_1 \wedge \sigma_2}\ (\wedge I) \\ \mathcal{E}(\Gamma) = B \ \&\ \mathcal{E}(M@\langle \Delta_1\,,\, \Delta_2 \rangle) = M' \end{cases}$$

$$\mathcal{F}\left(\frac{\mathcal{D}^{\mathrm{t}} : \Gamma \vdash M@\Delta : \sigma_1 \wedge \sigma_2}{\Gamma \vdash M@\mathsf{pr}_i\Delta : \sigma_i}\ \begin{matrix} i = 1,2 \\ (\wedge E_i) \end{matrix}\right) \triangleq \begin{cases} \dfrac{\mathcal{F}(\mathcal{D}^{\mathrm{t}}) : B \vdash M' : \sigma_1 \wedge \sigma_2}{B \vdash M' : \sigma_i}\ \begin{matrix} i = 1,2 \\ (\wedge E_i) \end{matrix} \\ \mathcal{E}(\Gamma) = B \ \&\ \mathcal{E}(M@\Delta) = M' \end{cases}$$

$$\mathcal{F}\left(\frac{\mathcal{D}_1^{\mathrm{t}} : \Gamma \vdash M@\Delta : \sigma_i}{\Gamma \vdash M@\mathsf{in}_i\Delta : \sigma_1 \vee \sigma_2}\ \begin{matrix} i = 1,2 \\ (\vee I_i) \end{matrix}\right) \triangleq \begin{cases} \dfrac{\mathcal{F}(\mathcal{D}^{\mathrm{t}}) : B \vdash M' : \sigma_i}{B \vdash M' : \sigma_1 \vee \sigma_2}\ \begin{matrix} i = 1,2 \\ (\vee I_i) \end{matrix} \\ \mathcal{E}(\Gamma) = B \ \&\ \mathcal{E}(M@\mathsf{in}_i\Delta) = M' \end{cases}$$

$$\mathcal{F}\left(\frac{\begin{array}{l}\mathcal{D}_1^{\mathrm{t}} : \Gamma, x@\iota_1{:}\sigma_1 \vdash M@\Delta_1 : \sigma_3 \\ \mathcal{D}_2^{\mathrm{t}} : \Gamma, x@\iota_2{:}\sigma_2 \vdash M@\Delta_2 : \sigma_3 \\ \mathcal{D}_3^{\mathrm{t}} : \Gamma \vdash N@\Delta_3 : \sigma_1 \vee \sigma_2\end{array}}{\Gamma \vdash M[N/x]@\begin{bmatrix}\lambda\iota_1{:}\sigma_1.\Delta_1\,, \\ \lambda\iota_2{:}\sigma_2.\Delta_2\end{bmatrix}\Delta_3 : \sigma_3}\ (\vee E)\right) \triangleq \begin{cases} \dfrac{\begin{array}{l}\mathcal{F}(\mathcal{D}_1^{\mathrm{t}}) : B, x{:}\sigma_1 \vdash M'' : \sigma_3 \\ \mathcal{F}(\mathcal{D}_2^{\mathrm{t}}) : B, x{:}\sigma_2 \vdash M'' : \sigma_3 \\ \mathcal{F}(\mathcal{D}_3^{\mathrm{t}}) : B \vdash N' : \sigma_1 \vee \sigma_2\end{array}}{B \vdash M' : \sigma_3}\ (\vee E) \\ \mathcal{E}(\Gamma) = B \ \&\ \mathcal{E}(M[N/x]@\begin{bmatrix}\lambda\iota_1{:}\sigma_1.\Delta_1\,, \\ \lambda\iota_2{:}\sigma_2.\Delta_2\end{bmatrix}\Delta_3) = M' \\ \mathcal{E}(M@\Delta_{1,2}) = M'' \ \&\ \mathcal{E}(N@\Delta_3) = N' \end{cases}$$

Fig. 4. The Function $\mathcal{F}$

4 Type Reconstruction and Type Checking Algorithms

The type reconstruction and the type checking algorithms are presented in Figure 6, and the following theorems holds.

Theorem 11 (Type Reconstruction for $\Lambda_{\mathrm{t}}^{\wedge\vee}$).

(Soundness) *If* $\mathsf{Type}(\Gamma, M@\Delta) = \sigma$, *then* $\Gamma \vdash M@\Delta : \sigma$;
(Completeness) *If* $\Gamma \vdash M@\Delta : \sigma$, *then* $\mathsf{Type}(\Gamma, M@\Delta) = \sigma$.

Proof. Soundness is proved by induction over the computation that $\mathsf{Type}(\Gamma, M@\Delta) = \sigma$; completeness is proved by induction on the derivation of $\Gamma \vdash M@\Delta : \sigma$.

$$\mathcal{G}\left(\frac{x{:}\sigma \in B}{B \vdash x : \sigma}\ (Var)\right) \triangleq \begin{cases} \dfrac{x@\iota{:}\sigma \in \Gamma}{\Gamma \vdash x@\iota : \sigma}\ (Var) \\ \mathcal{E}(\Gamma) = B \quad \iota \text{ is fresh} \end{cases}$$

$$\mathcal{G}\left(\frac{}{B \vdash M' : \omega}\ (\omega)\right) \triangleq \begin{cases} \dfrac{}{\Gamma \vdash M@* : \omega}\ (\omega) \\ \mathcal{E}(\Gamma) = B \ \&\ \mathcal{E}(M) = M' \end{cases}$$

$$\mathcal{G}\left(\frac{\mathcal{D}^{u}\!: B, x{:}\sigma_1 \vdash M' : \sigma_2}{B \vdash \lambda x.M' : \sigma_1 \to \sigma_2}\ (\to I)\right) \triangleq \begin{cases} \dfrac{\mathcal{G}(\mathcal{D}^{u}) : \Gamma, x@\iota{:}\sigma_1 \vdash M@\Delta : \sigma_2}{\Gamma \vdash (\lambda x{:}\iota.M)@(\lambda\iota{:}\sigma_1.\Delta) : \sigma_1 \to \sigma_2}\ (\to I) \\ \mathcal{E}(\Gamma, x@\iota{:}\sigma_1) = B, x{:}\sigma_1 \ \&\ \mathcal{E}(M@\Delta) = M' \end{cases}$$

$$\mathcal{G}\left(\frac{\mathcal{D}_1^{u} : B \vdash M' : \sigma_1 \to \sigma_2 \quad \mathcal{D}_2^{u} : B \vdash N' : \sigma_1}{B \vdash M'N' : \sigma_2}\ (\to E)\right) \triangleq \begin{cases} \dfrac{\mathcal{G}(\mathcal{D}_1^{u}) : \Gamma \vdash M@\Delta_1 : \sigma_1 \to \sigma_2 \quad \mathcal{G}(\mathcal{D}_2^{u}) : \Gamma \vdash N@\Delta_2 : \sigma_1}{\Gamma \vdash MN@\Delta_1\,\Delta_2 : \sigma_2}\ (\to E) \\ \mathcal{E}(\Gamma) = B \ \&\ \mathcal{E}(M@\Delta_1) = M' \ \&\ \mathcal{E}(N@\Delta_2) = N' \end{cases}$$

$$\mathcal{G}\left(\frac{\mathcal{D}_1^{u} : B \vdash M' : \sigma_1 \quad \mathcal{D}_2^{u} : B \vdash M' : \sigma_2}{B \vdash M' : \sigma_1 \wedge \sigma_2}\ (\wedge I)\right) \triangleq \begin{cases} \dfrac{\mathcal{G}(\mathcal{D}_1^{u}) : \Gamma \vdash M@\Delta_1 : \sigma_1 \quad \mathcal{G}(\mathcal{D}_2^{u}) : \Gamma \vdash M@\Delta_2 : \sigma_2}{\Gamma \vdash M@\langle \Delta_1\,,\, \Delta_2\rangle : \sigma_1 \wedge \sigma_2}\ (\wedge I) \\ \mathcal{E}(\Gamma) = B \ \&\ \mathcal{E}(M@\langle \Delta_1\,,\, \Delta_2\rangle) = M' \end{cases}$$

$$\mathcal{G}\left(\frac{\mathcal{D}^{u}\!: B \vdash M' : \sigma_1 \wedge \sigma_2}{B \vdash M' : \sigma_i}\ \begin{matrix} i = 1,2 \\ (\wedge E_i) \end{matrix}\right) \triangleq \begin{cases} \dfrac{\mathcal{G}(\mathcal{D}^{u}) : \Gamma \vdash M@\Delta : \sigma_1 \wedge \sigma_2}{\Gamma \vdash M@\mathsf{pr}_i\Delta : \sigma_i}\ \begin{matrix} i = 1,2 \\ (\wedge E_i) \end{matrix} \\ \mathcal{E}(\Gamma) = B \ \&\ \mathcal{E}(M@\Delta) = M' \end{cases}$$

$$\mathcal{G}\left(\frac{\mathcal{D}^{u}\!: B \vdash M' : \sigma_i}{B \vdash M' : \sigma_1 \vee \sigma_2}\ \begin{matrix} i = 1,2 \\ (\vee I_i) \end{matrix}\right) \triangleq \begin{cases} \dfrac{\mathcal{G}(\mathcal{D}^{u}) : \Gamma \vdash M@\Delta : \sigma_i}{\Gamma \vdash M@\mathsf{in}_i\Delta : \sigma_1 \vee \sigma_2}\ \begin{matrix} i = 1,2 \\ (\vee I_i) \end{matrix} \\ \mathcal{E}(\Gamma) = B \ \&\ \mathcal{E}(M@\mathsf{in}_i\Delta) = M' \end{cases}$$

$$\mathcal{G}\left(\frac{\mathcal{D}_1^{u} : B, x{:}\sigma_1 \vdash M' : \sigma_3 \quad \mathcal{D}_2^{u} : B, x{:}\sigma_2 \vdash M' : \sigma_3 \quad \mathcal{D}_3^{u} : B \vdash N' : \sigma_1 \vee \sigma_2}{B \vdash M'[N'/x] : \sigma_3}\ (\vee E)\right) \triangleq \begin{cases} \dfrac{\mathcal{G}(\mathcal{D}_1^{u}) : \Gamma, x@\iota_1{:}\sigma_1 \vdash M@\Delta_1 : \sigma_3 \quad \mathcal{G}(\mathcal{D}_2^{u}) : \Gamma, x@\iota_2{:}\sigma_2 \vdash M@\Delta_2 : \sigma_3 \quad \mathcal{G}(\mathcal{D}_3^{u}) : \Gamma \vdash N@\Delta_3 : \sigma_1 \vee \sigma_2}{\Gamma \vdash M[N/x]@\begin{bmatrix}\lambda\iota_1{:}\sigma_1.\Delta_1, \\ \lambda\iota_2{:}\sigma_2.\Delta_2\end{bmatrix}\Delta_3 : \sigma_3}\ (\vee E) \\ \mathcal{E}(\Gamma) = B \ \&\ \mathcal{E}(M[N/x]@\begin{bmatrix}\lambda\iota_1{:}\sigma_1.\Delta_1, \\ \lambda\iota_2{:}\sigma_2.\Delta_2\end{bmatrix}\Delta_3) = M'[N'/x] \\ \mathcal{E}(M@\Delta_{1,2}) = M' \ \&\ \mathcal{E}(N@\Delta_3) = N' \end{cases}$$

Fig. 5. The Function $\mathcal{G}$

Theorem 12 (Type Checking for $\Lambda_{\mathsf{t}}^{\wedge\vee}$). $\Gamma \vdash M@\Delta : \sigma$, *if and only if* $\mathsf{Typecheck}(\Gamma, M@\Delta, \sigma) = \mathsf{true}$.

Proof. The $\Rightarrow$ part can be proved using completeness of the type reconstruction algorithm (Theorem 11), while the $\Leftarrow$ part can be proved using soundness of the type reconstruction algorithm.

Corollary 13 ($\Lambda_{\mathsf{t}}^{\wedge\vee}$ Judgment Decidability). *It is decidable whether* $\Gamma \vdash M@\Delta : \sigma$ *is derivable.*

$$
\begin{array}{lcll}
\mathsf{Type}(\Gamma, M@\Delta) & \triangleq & \mathsf{match}\ M@\Delta\ \mathsf{with} & \\
\_@* & \Rightarrow & \omega & \\
\_@\mathsf{pr}_i\Delta_1 & \Rightarrow & \sigma_i \quad i=1,2 & \text{if } \mathsf{Type}(\Gamma, M@\Delta_1) = \sigma_1 \wedge \sigma_2 \\
\_@\langle \Delta_1\ ,\ \Delta_2 \rangle & \Rightarrow & \sigma_1 \wedge \sigma_2 & \text{if} \quad \mathsf{Type}(\Gamma, M@\Delta_1) = \sigma_1 \\
 & & & \text{and } \mathsf{Type}(\Gamma, M@\Delta_2) = \sigma_2 \\
\_@\mathsf{in}_i\Delta_1 & \Rightarrow & \sigma_1 \vee \sigma_2 & \text{if } \mathsf{Type}(\Gamma, M@\Delta_1) = \sigma_i \quad i=1,2 \\
\_@\left[\begin{array}{l}\lambda\iota_1{:}\sigma_1.\Delta_1, \\ \lambda\iota_2{:}\sigma_2.\Delta_2\end{array}\right]\Delta_3 & \Rightarrow & \sigma_3 & \text{if} \quad \mathsf{Type}((\Gamma, x@\iota_1{:}\sigma_1), M'@\Delta_1) = \sigma_3 \\
 & & & \text{and } \mathsf{Type}((\Gamma, x@\iota_2{:}\sigma_2), M'@\Delta_2) = \sigma_3 \\
 & & & \text{and } \mathsf{Type}(\Gamma, N@\Delta_3) = \sigma_1 \vee \sigma_3 \text{ and} \\
 & & & \text{and } M \equiv M'[N/x] \\
x@\iota & \Rightarrow & \sigma & \text{if } x@\iota{:}\sigma \in \Gamma \\
\lambda x{:}\iota.M_1@\lambda\iota{:}\sigma_1.\Delta_1 & \Rightarrow & \sigma_1 \to \sigma_2 & \text{if } \mathsf{Type}((\Gamma, x@\iota{:}\sigma_1), M_1@\Delta_1) = \sigma_2 \\
M_1\,M_2@\Delta_1\,\Delta_2 & \Rightarrow & \sigma_2 & \text{if} \quad \mathsf{Type}(\Gamma, M_1@\Delta_1) = \sigma_1 \to \sigma_2 \\
 & & & \text{and } \mathsf{Type}(\Gamma, M_2@\Delta_2) = \sigma_1 \\
\_@\_ & \Rightarrow & \mathsf{false} & \text{otherwise} \\
 & & & \\
\mathsf{Typecheck}(\Gamma, M@\Delta, \sigma) & \triangleq & \mathsf{Type}(\Gamma, M@\Delta) \stackrel{?}{=} \sigma &
\end{array}
$$

Fig. 6. The Type Reconstruction and Type Checking Algorithms for $\Lambda_{\mathsf{t}}^{\wedge\vee}$

5 Reduction in $\Lambda_{\mathsf{t}}^{\wedge\vee}$

As we have seen there is natural erasing function from typed $\Lambda_{\mathsf{t}}^{\wedge\vee}$ terms to untyped terms of $\Lambda_{\mathsf{u}}^{\wedge\vee}$. And reduction in the untyped λ-calculus is simply β-reduction. But as we have discussed in the introduction it would be a mistake to conflate typed and untyped reduction, in part due to the failure of coherence. Reduction on typed terms must respect the semantics of the type derivations, which is to say, *reduction on marked-terms must respect the semantics of the proof-terms.* The definition of the relation $\Rightarrow_\beta$ below ensures this. On the other hand it is useful to perform steps that keep the marked-term unchanged but reduce the proof-term, as long as the semantics of the type-derivation encoded by the proof-term is preserved. This is the role of the relation $\Rightarrow_\Delta$.

5.1 Synchronization

For a given term $M@\Delta$, the computational part (M) and the logical part (Δ) grow up together while they are built through application of rules (Var), ($\to I$), and ($\to E$), but they *get disconnected* when we apply the ($\wedge I$), ($\vee I$) or ($\wedge E$) rules, which change the Δ but not the M. This disconnection is "logged" in the Δ via occurrences of operators $\langle -,\ - \rangle$, $[-,\ -]$, pr_i, and in_i. In order to correctly identify the reductions that need to be performed in parallel in order to preserve the correct syntax of the term, we will define the notion of overlapping. Namely a redex is defining taking into account the surrounding context.

To define β-reduction on typed terms some care is required to manage the variable-marks. For this purpose we view $M@\Delta$ as a pair of trees, so subterms are associated by *tree-addresses*, as usual, sequences of integers.

Definition 14. *For a well-typed $M@\Delta$, we define the binary "synchronization" relation* Sync *between tree-addresses in M and tree-addresses in Δ. The definition is by induction over the typing derivation (see Figure 3). We present a representative set of cases here.*

- *When $M@\Delta$ is $x@\iota$: we of course take the roots to be related:* $\mathsf{Sync}(\langle\rangle,\langle\rangle)$
- *Case*

$$\frac{\Gamma \vdash M@\Delta_1 : \sigma_1 \to \sigma_2 \quad \Gamma \vdash N@\Delta_2 : \sigma_1}{\Gamma \vdash MN@\Delta_1\,\Delta_2 : \sigma_2} \quad (\to E)$$

  For an address a from MN and an address a' from $\Delta_1\,\Delta_2$: $\mathsf{Sync}(a,a')$ *if and only if either*
  - *a is from M and a' is from Δ_1, that is $a = 1b$ and $a' = 1b'$ and they were synchronized in $M@\Delta_1$, that is* $\mathsf{Sync}(b,b')$*, or*
  - *a and a' are from N and Δ_2 respectively and were synchronized in $N@\Delta_2$.*
- *Case*

$$\frac{\Gamma \vdash M@\Delta_1 : \sigma_1 \quad \Gamma \vdash M@\Delta_2 : \sigma_2}{\Gamma \vdash M@\langle \Delta_1\,,\,\Delta_2\rangle : \sigma_1 \wedge \sigma_2} \quad (\wedge I)$$

  Let a be an address in M and a' an address in $\langle\Delta_1\,,\,\Delta_2\rangle$. Then $\mathsf{Sync}(a,a')$ *if and only if $a' = 1b$ and* $\mathsf{Sync}(a,b)$ *from $M@\Delta_1$ or $a' = 2b$ and* $\mathsf{Sync}(a,b)$ *from $M@\Delta_2$.*
- *Case*

$$\frac{\Gamma, x@\iota_1{:}\sigma_1 \vdash M@\Delta_1 : \sigma_3 \quad \Gamma, x@\iota_2{:}\sigma_2 \vdash M@\Delta_2 : \sigma_3 \quad \Gamma \vdash N@\Delta_3 : \sigma_1 \vee \sigma_2}{\Gamma \vdash M[N/x]@[\lambda\iota_1{:}\sigma_1.\Delta_1\,,\,\lambda\iota_2{:}\sigma_2.\Delta_2]\,\Delta_3 : \sigma_3} \quad (\vee E)$$

  Let a be an address in $M[N/x]$. Then for an address a' from $[\lambda\iota_1{:}\sigma_1.\Delta_1\,,\,\lambda\iota_2{:}\sigma_2.\Delta_2]\,\Delta_3$ we have $\mathsf{Sync}(a,a')$ *just in case one of the following holds*
  - *a is an address in M other than that of x, and for some i and some address b' of Δ_i we have* $\mathsf{Sync}(a,b')$ *and a' is the corresponding Δ_i subterm in $[\lambda\iota_1{:}\sigma_1.\Delta_1\,,\,\lambda\iota_2{:}\sigma_2.\Delta_2]\,\Delta_3$ (precisely: $a' = 1ib'$)*
  - *a is an address corresponding to an address b in N after the substitution (precisely, $a = db$ where x occurs at address d in M), a' is an address corresponding to an address b' in Δ_3 ($a' = 2b'$) and we have* $\mathsf{Sync}(b,b')$*.*

Definition 15. *Consider $M@\Delta$. Let a and b be addresses in M. Say that $a \sim b$ if there is some c an address in Δ with both* $\mathsf{Sync}(a,c)$ *and* $\mathsf{Sync}(b,c)$*. In a precisely analogous way we define $\sim$ on addresses in Δ.*

It easy to check that if two addresses are $\sim$ then the corresponding subterms are identical. It is also clear that if $\mathsf{Sync}(a,a')$ and a is the address of a β-redex in M then b is the address of a β-redex in Δ and conversely. It is clear that $\sim$ is an equivalence relation. So we may define: a *synchronized pair* to be a pair (S,S') of sets of addresses in M and Δ respectively such that S and S' are each $\sim$-equivalence classes pointwise related by Sync; that is for each $a \in S$ and each $a' \in S'$ we have $\mathsf{Sync}(a,a')$.

5.2 The Reduction Relation $\Rightarrow$

We define $\Rightarrow$ as the union of two reductions: $\Rightarrow_\beta$ deals with β-reduction occurring in both the marked- and the proof-term; while $\Rightarrow_\Delta$ deals with reductions arising from proof-term simplifications. Proof-term reduction is defined from the equations in Definition 5.

Definition 16. *The reduction relation $\rightarrow_{\wedge\vee}$ on proof-terms is defined by orienting equations (2), (3), and (7) from Definition 5 from left to right.*

Definition 17.

($\Rightarrow_\beta$) *Let $\mathsf{C}\{\,\}_{i\in I}$ (resp, $\mathsf{C}'\{\,\}_{i\in I}$) be a* multihole *marked-term context (resp. proof-term context), and consider*

$$\mathsf{C}\{\}@\mathsf{C}'\{\}$$

where the indicated occurrences of holes form a synchronized pair of sets of subterms. Then the $\Rightarrow_\beta$ reduction is defined as follows:

$$\mathsf{C}\{(\lambda x{:}\iota.M)\,N\}_{i\in I}@\mathsf{C}'\{(\lambda\iota{:}\sigma_j.\Delta_j)\,\Delta'_j\}_{j\in J} \Rightarrow_\beta \mathsf{C}\{M[N/x]\}_{i\in I}@\mathsf{C}'\{\Delta_j[\Delta'_j/\iota]\}_{j\in J}$$
$$\mathsf{C}\{(\lambda x{:}\iota.M)\,N\}_{i\in I}@\mathsf{C}'\{*\}_{j\in J} \Rightarrow_\beta \mathsf{C}\{M[N/x]\}_{i\in I}@\mathsf{C}'\{*\}_{j\in J}$$

($\Rightarrow_\Delta$) *Let $\mathsf{C}'\{\,\}$ be a plain (one-hole) proof-context. Then the $\Rightarrow_\Delta$ reduction is of the form:*

$$M@\mathsf{C}'\{\Delta\} \Rightarrow_\Delta M@\mathsf{C}'\{\Delta'\} \qquad \textit{where } \Delta \rightarrow_{\wedge\vee} \Delta'.$$

Note that the reduction $\Rightarrow_\beta$ is characterized by the two distinct patterns written above. There is no overlap between these two cases, since, as observed just after the definition of $\sim$ a term $(\lambda x{:}\iota.M)\,N$ cannot be synchronized with both an occurrence of $(\lambda\iota{:}\sigma.\Delta)\,\Delta'$ and an occurrence of $*$.

Remarks

- In the definition of $\Rightarrow_\beta$: it is interesting to note the following duality: the typing rule $(\wedge I)$ is what leads us to synchronize one variable-mark ι occurring in a redex in the marked-term (the computation), *e.g.* $(\lambda x{:}\iota.M)\,N$, with potentially many redexes in the proof-term, *e.g.* $(\lambda\iota{:}\sigma_j.\Delta_j)\,\Delta'_j$ with $j \in J$. Symmetrically, the typing rule $(\vee E)$ is what leads us to synchronize one variable-mark occurring in a redex in the logic part, *e.g.* $(\lambda\iota{:}\sigma.\Delta)\,\Delta'$, with potentially many (but equal) redexes in the computational part, *e.g.* i-occurrences of $(\lambda x{:}\iota.M)\,N$ with $i \in I$.
- Implementation of $\Rightarrow$ is potentially complicated by the need to manage the $\sim$ relation. But in an implementation in which subterm-occurrences can be *shared* there is no real need for a "many-to-many" relation on addresses.
- The erasure of the relation $\Rightarrow_\beta$ is similar to (though not identical with) the parallel reduction relation defined in [3].

5.3 Example of Reduction for $\Lambda_t^{\wedge\vee}$ (Continued)

As an example of the treatment of intersection and union types in our system we examine Pierce's example in [3] showing the failure of subject reduction for simple, non parallel, reduction. The term in question is a good example to show the role of synchronization in reduction on $\Lambda_t^{\wedge\vee}$ terms. Then the complete untyped reduction is:

$$x(\mathsf{I}(yz))(\mathsf{I}(yz)) \Rightarrow_\beta \begin{array}{l} \nearrow^\beta x\,(yz)\,(\mathsf{I}\,(yz))\ \searrow_\beta \\ \searrow_\beta x\,(\mathsf{I}\,(yz))\,(yz)\ \nearrow^\beta \end{array} x\,(yz)\,(yz).$$

Under the type context $B \triangleq x{:}(\sigma_1 \to \sigma_1 \to \tau) \wedge (\sigma_2 \to \sigma_2 \to \tau), y{:}\rho \to \sigma_1 \vee \sigma_2, z : \rho$, the first and the last terms can be typed with τ, while terms in the "fork" are not because of the mismatch of the premises in the $(\vee E)$ type assignment rule. The typed term is

$$x(\underbrace{(\lambda v{:}\iota_3.v)}_{\mathsf{I}_\mathsf{t}}(yz))(\underbrace{(\lambda v{:}\iota_3.v)}_{\mathsf{I}_\mathsf{t}}(yz))@[\underbrace{\lambda\iota_1{:}\sigma_1.(\mathsf{pr}_1\iota)\iota_1\iota_1}_{\Delta_1}, \underbrace{\lambda\iota_2{:}\sigma_2.(\mathsf{pr}_2\iota)\iota_2\iota_2}_{\Delta_2}](\underbrace{(\lambda\iota_3{:}\sigma_1.\iota_3)}_{\Delta_3}(\iota_4\iota_5))$$

and the typed synchronized reduction goes as follows

$$x(\mathsf{I}_\mathsf{t}\,(yz))(\mathsf{I}_\mathsf{t}\,(yz))@[\Delta_1\,,\,\Delta_2]\,(\Delta_3\,(\iota_4\iota_5)) \qquad \Rightarrow_\Delta$$

$$x\,\underbrace{\overbrace{(\mathsf{I}_\mathsf{t}\,(yz))}^{\to\beta}\,\overbrace{(\mathsf{I}_\mathsf{t}\,(yz))}^{\to\beta}@[\Delta_1\,,\,\Delta_2]\,\overbrace{(\Delta_3\,(\iota_4\iota_5))}^{\to\beta}\,\overbrace{(\Delta_3\,(\iota_4\iota_5))}^{\to\beta}}_{\text{fire a } \Rightarrow_\beta \text{ redex}} \Rightarrow_\beta x\,(yz)\,(yz)@[\Delta_1\,,\,\Delta_2]\,(\iota_4\iota_5)$$

5.4 Properties of $\Rightarrow$

We have seen that the relationship between the corresponding type systems of $\Lambda_u^{\wedge\vee}$ and $\Lambda_t^{\wedge\vee}$ is essentially one of isomorphism. The relationship between the reduction relations in the two calculi is more interesting. First, modulo erasing there is a sense in which $\Rightarrow$ is a sub-relation of untyped $=_\beta$. More precisely:

Lemma 18. *If $M@\Delta \Rightarrow N@\Delta'$ then $\mathcal{E}(M@\Delta) \to \mathcal{E}(N@\Delta')$.*

Proof. Straightforward, using the auxiliary result that $\mathcal{E}(M[N/x]@\Delta) \equiv \mathcal{E}(M)[\mathcal{E}(N)/x]$.

Reduction out of typed terms is well-behaved in the sense witnessed by the following traditional properties.

Theorem 19. *Let $M@\Delta$ be a typable term of $\Lambda_t^{\wedge\vee}$.*

1. *(Subject Reduction) If $\Gamma \vdash M@\Delta : \sigma$ and $M@\Delta \Rightarrow M'@\Delta'$, then $\Gamma \vdash M'@\Delta' : \sigma$.*
2. *(Church-Rosser) The reduction relation $\Rightarrow_\beta$ is confluent out of $M@\Delta$.*
3. *(Strong Normalization) If $M@\Delta$ is a typable without using rule ω then $\Rightarrow_\beta$ is strongly normalizing out of $M@\Delta$.*

Proof. The proof of Subject Reduction is routine. It should be noted that the typical obstacle to Subject Reduction in the presence of a rule such as $(\vee E)$ does not arise for us, since our reduction relation is already necessarily of a "parallel" character due to the requirement of maintaining synchronization. Confluence can be shown by an easy application of standard techniques (for example, the Tait&Martin-Löf parallel reduction argument). Strong Normalization is immediate from the fact that $\to_{\wedge\vee}$ is strongly normalizing.

On the other hand we may point out two (related) aspects of typed reduction that are at first glance anomalous.

Getting stuck. The need for marked-term β-redexes to be synchronized with proof-term β-redexes mean that a marked-term β-redex might not be able to participate in a reduction. This can happen when a term $P@[\lambda\iota_1{:}\sigma_1.\Delta_1\,,\ \lambda\iota_2{:}\sigma_2.\Delta_2]\,\Delta_3$ is typed by $(\vee E)$ and the marked-term $P \equiv M[N/x]$ is β-redex. Since the corresponding proof-term $[\lambda\iota_1{:}\sigma_1.\Delta_1\,,\ \lambda\iota_2{:}\sigma_2.\Delta_2]\,\Delta_3$ is not a β-redex in the proof-term calculus we can view the typed term as being "stuck." Now the proof-term may reduce via $\Rightarrow_\Delta$ and eventually become a β-redex. Indeed it is not hard to show that if the term is closed (or if every free variable has Harrop type, defined in Section 6.1) then this will always happen. But in general we can have normal forms in the typed calculus whose erasures contain β-redexes in the sense of untyped λ-calculus. This phenomenon is inherent in having a typed calculus with unions. The β-reductions available in the Curry-style system have a character from the coproduct reductions on proof-terms: a term $[\lambda\iota_1{:}\sigma_1.\Delta_1\,,\ \lambda\iota_2{:}\sigma_2.\Delta_2]\,\Delta_3$ has to wait for its argument Δ_3 to manifest itself as being of the form $\mathsf{in}_i\Delta_4$. And in order to maintain the synchronization between marked-terms and proof-terms, the marked-term β-redex must wait as well.

Another manifestation of the constraint that the marked- and proof- components of a term must be compatible is the fact that—even though the type system has a universal type ω—the system does not have the Subject Expansion property.

Failure of Subject Expansion. There exist typed terms $M@\Delta$ and $M'@\Delta'$ such that $M@\Delta \Rightarrow M'@\Delta'$ and $M'@\Delta'$ is typable but $M@\Delta$ is not typable. For example

$$(\lambda x{:}\iota_1.x)@\mathsf{pr}_1\langle\lambda\iota_1{:}\sigma.\iota_1\,,\ \lambda\iota_1{:}\sigma.\lambda\iota_2{:}\tau.\iota_1\rangle \quad\Rightarrow\quad (\lambda x{:}\iota_1.x)@(\lambda\iota_1{:}\sigma.\iota_1)$$

The latter is clearly a typed term with type $\sigma \to \sigma$. But it is easy to see that in order for the former term to be typed it would have to be the case that $(\lambda x{:}\sigma_1.x)@\mathsf{pr}_1\langle\lambda\iota_1{:}\sigma.\iota_1\,,\ \lambda\iota_1{:}\sigma.\lambda\iota_2{:}\tau.\iota_1\rangle$ is a typed term, which means in turn that $(\lambda x{:}\iota_1.x)@(\lambda\iota_1{:}\sigma.\lambda\iota_2{:}\tau.\iota_1)$ is a typed term; and this is not the case.

Of course in untyped λ-calculus we may use ω to type terms which are "erased" in a reduction: this is the essence of why Subject Expansion holds in the presence of ω. But this move is not available to us here. The problem with $(\lambda x{:}\sigma_1.x)@(\lambda\iota_1{:}\sigma.\lambda\iota_2{:}\tau.\iota_1)$ as a typed term is not the lack of a general-enough type, it is the fact that $(\lambda\iota_1{:}\sigma.\lambda\iota_2{:}\tau.\iota_1)$ cannot encode the shape of a derivation of a type-assignment to $(\lambda x.x)$.

6 Deciding Type-Derivation Equality

As described in the introduction, when semantics is given to typing derivations in Church-style, the question arises: "what is the relationship between the semantics of different derivations of the same typing judgment?" In this section we explore the closely related question "when should two derivations of the same judgment be considered equal?"

We locate the semantics of type-derivations in a cartesian closed category with binary coproducts (*i.e.*, a bicartesian closed category but without an initial type). Since we are

interested here in those equalities between derivations which hold in all such categories interpreting the derivations, we focus on the equalities holding in the *free* bi-cartesian closed category over the graph whose objects are the type-variables and the constant ω and whose arrows include the primitive coercion-constants Σ. These equalities are determined by the equational theory $\cong$.

The theory of these equations is surprisingly subtle. On the positive side, it is proved in [10] that a Friedman completeness theorem holds for the theory, that is, that the equations provable in this theory are precisely the equations true in the category of sets. On the other hand the rewriting behavior of the equations is problematic: as described in [9], confluence fails for the known presentations, and there cannot exist a left-linear confluent presentation.

When the equation $[\lambda\iota{:}\sigma_1.\Delta(\mathsf{in}_1\iota)\,,\ \lambda\iota{:}\sigma_2.\Delta(\mathsf{in}_2\iota)] = \Delta$ is dropped, yielding the theory of cartesian closed categories with *weak sums*, the theory admits a strongly normalizing and confluent presentation, so the resulting theory is decidable. In fact the rewriting theory of the cartesian closed categories with weak sums was the object of intense research activity in the 1990's: a selection of relevant papers might include [6,1,9,5,12,7].

So there are rich and well-behaved rewriting theories capturing fragments of $\cong$. But if we want to embrace $\cong$ in its entirety we need to work harder. Ghani [11] presented a complex proof of decidability of the theory via an analysis of a non-confluent rewriting system. The most successful analysis of the theory to date is that by Altenkirch, Dybjer, Hofmann, and Scott [2] based on the semantical technique of Normalization by Evaluation. In that paper a computable function nf is defined mapping terms to "normal forms," together with a computable function d mapping normal forms to terms, satisfying the following theorem.

Theorem 20 ([2]). *For every* Δ, $\Delta \cong \mathsf{d}(\mathsf{nf}(\Delta))$, *and for every* Δ_1 *and* Δ_2, $\Delta_1 \cong \Delta_2$ *if and only if* $\mathsf{d}(\mathsf{nf}(\Delta_1))$ *and* $\mathsf{d}(\mathsf{nf}(\Delta_2))$ *are identical.*

Corollary 21. *Equality between type-derivations is decidable.*

6.1 Type Theories

It is traditional in treatments of Curry-style typing to consider types modulo a subtyping relation $\leq$ under which the set of types makes a partially ordered set. Following [3] we refer to such a set of inequalities as a *type theory*. It is fairly straightforward to incorporate theories of subtyping into our Church-style system; we outline the development briefly here. It is best to start with the proof-terms, as the carriers of semantic information. As suggested in [18,24] we need to view subtyping semantically not as simple set-inclusion but as a relationship witnessed by coercion functions. So in the syntax of proof-terms we postulate a signature Σ of names for primitive coercion functions $\mathsf{c} : \sigma \to \tau$. Fixing a signature Σ of coercion constants corresponds to a type theory in the sense of [3] in an intuitively obvious way: each $\mathsf{c} : \sigma \to \tau$ corresponds to an axiom $\sigma \leq \tau$ in the theory.

Conversely, certain type theories can be naturally captured by defining an appropriate signature. The *minimal type theory* Θ from [3] will correspond to coercions defined by

the proof-terms as in Figure 2 (without the subtyping rule). That is, Θ, corresponds to the empty signature Σ.

Another important type theory is the theory Π obtained from Θ by adding equations making the lattice of types distributive and adding an "Extended Disjunction Property" axiom

$$\sigma \to (\rho \vee \tau) \leq (\sigma \to \rho) \vee (\sigma \to \tau) \qquad \text{when } \phi \text{ is a Harrop type.}$$

(A type is a Harrop type if the disjunction constructor $\vee$ occurs only in negative position.) The importance of the type theory Π in the Curry-style setting is that under Π Subject Reduction holds for ordinary β-reduction: if $B \vdash M : \sigma$ and either $M =_\beta N$ or $M \to_\eta N$ then $B \vdash N : \sigma$.

We now describe how to construct a signature Σ_Π corresponding to the type theory Π. Recall that this theory is obtained from Θ by adding axioms for distributivity and for the extended disjunction property. The latter rule can be captured by a family of constants

$$\mathsf{dp} : (\sigma \to (\rho \vee \tau)) \to ((\sigma \to \rho) \vee (\sigma \to \tau))$$

We need not add constants capturing the distributivity axiom, since the semantics of our proof-terms is based on categories that are cartesian closed with binary coproducts, and in such categories products always distribute over coproducts. Now to introduce coercions into our Church-style typing system we add the following rule

$$\frac{\Gamma \vdash M @ \Delta : \sigma_1 \quad \mathsf{c} : \sigma_1 \to \sigma_2 \in \Sigma}{\Gamma \vdash M @ \mathsf{c}\Delta : \sigma_2} \quad (Coerce)$$

Concerning equality between type-derivations, if we want to reason about equality between type-derivations under the type theory Π we need to take into account the behavior of the basic coercion functions $\mathsf{dp} : (\sigma \to (\rho \vee \tau)) \to ((\sigma \to \rho) \vee (\sigma \to \tau))$.

Whenever the coercions dp are injective we have that for two proof-terms Δ_1 and Δ_2, $\mathsf{dp}\Delta_1 = \mathsf{dp}\Delta_2$ only if $\Delta_1 = \Delta_2$. So in reasoning about such coercions syntactically, there are no additional axioms or rules of inference that apply, in other words *we can treat the* dp *constants as free variables.* Since the techniques of [2] apply perfectly well to open terms, we conclude the following.

Corollary 22. *Equality between type-derivations under the type theory* Π *is decidable.*

7 Future Work

The reduction semantics of our calculus is complex, due to the well-known awkwardness of the $(\vee E)$ rule. Since this is largely due to the global nature of the substitution in the conclusion; this suggests that an explicit substitutions calculus might be better-behaved.

There is a wealth of research yet to be done exploring coherence in the presence of union types: as we have seen the structure of the category of types affects the semantics of derivations. For instance, decidability of equality when coercions are not assumed to be injective needs attention.

We took a fairly naive approach to the semantics of type-derivations in this paper; we were content to derive some results that assumed nothing more about the semantics than cartesian closure and coproducts. But the failure of coherence, implying that the meanings of type-derivations are not "generic," suggests that there is interesting structure to be explored in the semantics of coercions in the presence of unions.

Acknowledgments. We are grateful to Mariangiola Dezani-Ciancaglini for several illuminating discussions about this work.

References

1. Akama, Y.: On Mints' reduction for ccc-calculus. In: Bezem, M., Groote, J.F. (eds.) TLCA 1993. LNCS, vol. 664, pp. 1–12. Springer, Heidelberg (1993)
2. Altenkirch, T., Dybjer, P., Hofmann, M., Scott, P.J.: Normalization by evaluation for typed lambda calculus with coproducts. In: LICS, pp. 303–310 (2001)
3. Barbanera, F., Dezani-Ciancaglini, M., De'Liguoro, U.: Intersection and union types: syntax and semantics. Inf. Comput. 119(2), 202–230 (1995)
4. Capitani, B., Loreti, M., Venneri, B.: Hyperformulae, Parallel Deductions and Intersection Types. BOTH, Electr. Notes Theor. Comput. Sci. 50(2), 180–198 (2001)
5. Di Cosmo, R., Kesner, D.: A confluent reduction for the extensional typed lambda-calculus with pairs, sums, recursion and terminal object. In: Lingas, A., Carlsson, S., Karlsson, R. (eds.) ICALP 1993. LNCS, vol. 700, pp. 645–656. Springer, Heidelberg (1993)
6. Cubric, D.: Embedding of a free cartesian closed category into the category of sets. McGill University (1992) (manuscript)
7. Curien, P.-L., Di Cosmo, R.: A confluent reduction for the lambda-calculus with surjective pairing and terminal object. J. Funct. Program. 6(2), 299–327 (1996)
8. Curien, P.-L., Ghelli, G.: Coherence of subsumption, minimum typing and type-checking in $F_{<=}$. Mathematical Structures in Computer Science 2(1), 55–91 (1992)
9. Dougherty, D.J.: Some lambda calculi with categorical sums and products. In: Kirchner, C. (ed.) RTA 1993. LNCS, vol. 690, pp. 137–151. Springer, Heidelberg (1993)
10. Dougherty, D.J., Subrahmanyam, R.: Equality between functionals in the presence of coproducts. Information and Computation 157(1,2), 52–83 (2000)
11. Ghani, N.: ßn-equality for coproducts. In: Dezani-Ciancaglini, M., Plotkin, G. (eds.) TLCA 1995. LNCS, vol. 902, pp. 171–185. Springer, Heidelberg (1995)
12. Jay, C.B., Ghani, N.: The virtues of eta-expansion. J. Funct. Program. 5(2), 135–154 (1995)
13. Knuth, D.E.: Examples of formal semantics. In: Engeler, E. (ed.) Symposium on Semantics and Algorithmic Languages. LNM, vol. 188, pp. 212–235. Springer, Heidelberg (1970)
14. Lambek, J., Scott, P.: Introduction to Higher-order Categorical Logic. Cambridge Studies in Advanced Mathematics, vol. 7. Cambridge University Press, Cambridge (1986)
15. Liquori, L., Ronchi Della Rocca, S.: Intersection typed system *à la* Church. Information and Computation 9(205), 1371–1386 (2007)
16. MacQueen, D., Plotkin, G., Sethi, R.: An ideal model for recursive polymorphic types. Information and Control 71, 95–130 (1986)
17. Pierce, B.C., Turner, D.N.: Simple type-theoretic foundations for object-oriented programming. Journal of Functional Programming 4(2), 207–247 (1994)
18. Reynolds, J.C.: Using category theory to design implicit conversions and generic op erators. In: Jones, N.D. (ed.) Semantics-Directed Compiler Generation. LNCS, vol. 94, pp. 211–258. Springer, Heidelberg (1980)

19. Reynolds, J.C.: The coherence of languages with intersection types. In: Ito, T., Meyer, A.R. (eds.) TACS 1991. LNCS, vol. 526, pp. 675–700. Springer, Heidelberg (1991)
20. Reynolds, J.C.: Design of the programming language Forsythe. Report CMU–CS–96–146, Carnegie Mellon University, Pittsburgh, Pennsylvania, June 28 (1996)
21. Reynolds, J.C.: Theories of Programming Languages. Cambridge University Press, Cambridge (1998)
22. Ronchi Della Rocca, S.: Intersection typed lambda-calculus. Electr. Notes Theor. Comput. Sci. 70(1) (2002)
23. Ronchi Della Rocca, S., Roversi, L.: Intersection logic. In: Fribourg, L. (ed.) CSL 2001 and EACSL 2001. LNCS, vol. 2142, pp. 421–428. Springer, Heidelberg (2001)
24. Tannen, V., Coquand, T., Gunter, C.A., Scedrov, A.: Inheritance as implicit coercion. Inf. Comput. 93(1), 172–221 (1991)
25. van Bakel, S., Liquori, L., Ronchi della Rocca, S., Urzyczyn, P.: Comparing Cubes of Typed and Type Assignment System. Annal of Pure and Applied Logics 86(3), 267–303 (1997)
26. Wells, J.B., Haack, C.: Branching types. In: Le Métayer, D. (ed.) ESOP 2002. LNCS, vol. 2305, pp. 115–132. Springer, Heidelberg (2002)

Graded Alternating-Time Temporal Logic

Marco Faella[1], Margherita Napoli[2], and Mimmo Parente[2]

[1] Università di Napoli "Federico II", Via Cintia, 80126 - Napoli, Italy
[2] Università di Salerno, Via Ponte don Melillo, 84084 - Fisciano (SA), Italy

Abstract. Graded modalities enrich the universal and existential quantifiers with the capability to express the concept of *at least k* or *all but k*, for a non-negative integer k. Recently, temporal logics such as μ-calculus and Computational Tree Logic, CTL, augmented with graded modalities have received attention from the scientific community, both from a theoretical side and from an applicative perspective. Both μ-calculus and CTL naturally apply as specification languages for *closed* systems: in this paper, we add graded modalities to the Alternating-time Temporal Logic (ATL) introduced by Alur et al., to study how these modalities may affect specification languages for *open* systems. We present, and compare with each other, three different semantics. We first consider a natural interpretation which seems suitable to off-line synthesis applications and then we restrict it to the case where players can only employ memoryless strategies. Finally, we strengthen the logic by means of a different interpretation which may find application in the verification of fault-tolerant controllers. For all the interpretations, we efficiently solve the model-checking problem both in the concurrent and turn-based settings, proving its PTIME-completeness. To this aim we also exploit also a characterization of the maximum grading value of a given formula.

1 Introduction

Graded modalities are logical operators allowing to express quantitative bounds on the set of individuals satisfying a certain property [11]. They are well-known in the knowledge representation field, as well as in classical logic [12] and in description logics [13]. Such modalities have received renewed attention by the theoretical computer science community, especially in the formal verification field: in [15,7] they are applied to the μ-calculus logic, while in [8,10,4] to CTL. Here, we add graded modalities to ATL, as a step from closed to open systems, and provide efficient model-checking algorithms for the resulting logic. At the best of our knowledge, this is the first time that such notions are applied in the game-theoretic setting.

ATL was introduced by Alur et al. [2] as a derivative of CTL that is interpreted on *games*, rather than transition systems. Since its inception, ATL has been quickly adopted in different areas of computer science dealing with multi-agent systems, and it has provided the basis for further extensions [1,20,5].

The temporal part of ATL coincides with the one of CTL, while the path quantifiers of CTL are replaced by *team* quantifiers, that quantify over the strategies

E.M. Clarke and A. Voronkov (Eds.): LPAR-16, LNAI 6355, pp. 192–211, 2010.

of a given team. For instance, for a suitable subformula θ, the ATL formula $\langle\langle 1 \rangle\rangle \theta$ expresses the fact that the team composed of Player 1 alone can ensure that θ holds. More in detail, said formula hides two classical quantifiers: there exists a strategy of Player 1, such that, whatever the other players do, θ holds in the resulting outcomes. (Standard CTL path quantifiers can be obtained as special cases of ATL quantifiers.

In this paper, we enrich the ATL quantifiers with an integral *grade*, and we interpret the resulting formulas using three alternative semantics. First, we consider a very natural extension of the semantics of ATL formulas: for a natural number k, the graded ATL formula $\langle\langle X \rangle\rangle^k \theta$ affirms that the players belonging to the team X have k *different* strategies to enforce θ, that is, the team has k different ways of winning, each satisfying θ, whatever the remaining players do. Intuitively, two distinct runs of the play are counted as different if they present a difference in the choice of the moves leading to satisfy the winning condition. We call this semantics *off-line*, as it seems suitable to off-line synthesis applications. In this context, a two-player game is a model of a control system, and the two players represent the controller and its environment, respectively. Verifying the property $\langle\langle 1 \rangle\rangle^k \theta$, and possibly computing k witnessing strategies for Player 1, corresponds to synthesizing k different controllers, that may later (i.e., off-line) be compared w.r.t. some external criterion.

However, as shown in the following, some cases may exhibit infinitely many winning strategies, calling for a refined counting notion. We therefore introduce the *memoryless* semantics, that only counts the number of different memoryless winning strategies, i.e., strategies whose choices only depend on the current state in the game. This restriction makes perfect sense in the controller synthesis scenario, where memoryless controllers are highly desirable for their simplicity.

Then, we turn to the application of automatic verification of fault-tolerant controllers for open systems. In this case, we do not wish to restrict the moves of a player (i.e., synthesize a controller), but rather we assume that the controller may take any of the (redundant) actions that are present in the game, and we want to evaluate how many faults the controller can tolerate at most before violating its specification, where a fault is represented by the absence at run-time of a move that is present in the model. To this purpose, we introduce a new semantics to count the number of different winning paths that a team can follow, in the worst possible case w.r.t. the choices of the opposite team. We call this semantics "on-line" because it is related to the ability of the player to dynamically alter its behavior to overcome faults. In this sense, the graded ATL formula $\langle\langle X \rangle\rangle^k \theta$, interpreted in the on-line semantics, becomes a necessary condition to guarantee that team X can force θ even in the presence of $k - 1$ faults.

To prove our results, we first consider the case of turn-based games, where states are partitioned among the players and at each state the player who owns it moves along one of the outgoing edges. We compare the semantics and prove that the on-line satisfaction of a negation-free graded ATL formula implies its off-line

satisfaction, while the vice versa is not true. The on-line and the memoryless semantics turn out to be incomparable.

For all semantics we then solve the model-checking problem, computing the truth values of graded ATL formulas on the states of a given game. We provide algorithms that are executed in polynomial time w.r.t. the size of the input game and the number of logical operators in the input formula. A matching lower bound shows that these problems are in fact PTIME-complete. For the off-line and the on-line semantics, the time complexity does not even depend on the constants occurring in the graded team quantifiers. In particular, for the off-line semantics we retain the same complexity as in ATL.

Given an ATL formula, an extended form of model checking is to determine the value of the maximum grade for which that formula is true on a given state of a game. For the off-line and the on-line interpretations, we provide a fixpoint characterization for that value. A fixpoint characterization also suggests the simple Picard iteration method for computing such value. However, in our case, two issues prevent Picard iteration from being applied effectively. First, the maximum grade of a formula can be infinity. Second, even if grade infinity was to be treated separately, Picard iteration would still require a number of iterations proportional to the integer value being computed. For these reasons, the algorithms we give are ad-hoc, and compute the maximum grade of a formula avoiding the above-mentioned issues, while still exploiting the fixpoint characterization.

Finally, we consider concurrent game structures and show that the model-checking problem is PTIME-complete and can be solved with the same complexity as for turn based games.

ATL has been implemented in several tools for the analysis of open systems [3,17], and graded modalities for CTL have been integrated in the NuSMV tool [9,6]. We plan to extend this in the near future to consider graded ATL specifications.

The rest of the paper is organized as follows. Section 2 shows an example. Section 3 presents the basic definitions, including the three alternative semantics for graded ATL. Section 4 presents a fixpoint characterization of the off-line and on-line semantics. Section 5 performs a comparison between the semantics. Section 6 describes the model-checking algorithms, computing the truth values of graded ATL formulas on the states of a given game. In Section 7 we deal with concurrent games.

2 A Motivating Example

We give now an example, before introducing our logic. Consider the game in Figure 1, representing the steps required to open an attachment sent by an external agent in one of three possible file formats. The game is played by two players with a turn-based modality: each state belongs only to one of the players and at each turn the player who owns the current state chooses one of its outgoing edges. In the figure, states of Player 1 are circles and those of Player 2 are squares. In the initial state s_0, Player 2 picks a file type, among PostScript, Adobe PDF, and Microsoft Word. Then, Player 1 tries to open the file (i.e., reach state s_4)

by using appropriate programs. The ATL formula $\langle\langle 1 \rangle\rangle \Diamond s_4$, meaning "Player 1 has a strategy to reach state s_4", is true at s_0, because, no matter what file type Player 2 chooses, Player 1 has a way (sometimes more than one) to reach s_4. Graded ATL provide the means to *count* how many different ways to win Player 1 has and this, clearly, cannot be achieved with classical ATL.

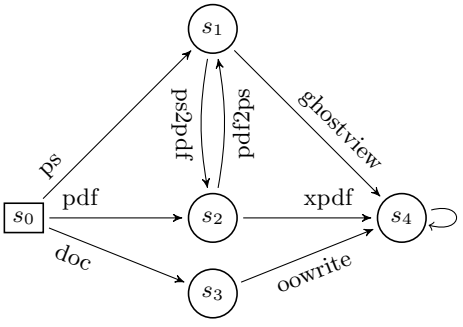


Fig. 1. An attachment-opening game

Assume first that the aim for analyzing the game in Figure 1 is to automatically generate as many scripts as possible, each one of them able to open all types of attachment (i.e., win the game). If we apply the off-line semantics, it turns out that Player 1 has infinitely many winning strategies: When receiving a ps or pdf file, she can choose to run the converting programs ps2pdf and pdf2ps as many times as she likes, going back and forth between the two formats, before opening the file and reaching the target state s_4. In graded ATL terms, in the off-line semantics the formula $\langle\langle 1 \rangle\rangle^k \Diamond s_4$ holds at s_0 for all $k \geq 1$. Clearly, we are not interested in synthesizing infinitely many scripts that only differ in the amount of useless work they perform. Hence, we introduce the memoryless semantics, that only counts the number of memoryless winning strategies. In the current example, there are three memoryless winning strategies: the one that uses no converters, the one that uses only ps2pdf and the one that uses only pdf2ps. Using both converting programs leads to an infinite loop that does not reach state s_4. Formally, in the memoryless semantics the formula $\langle\langle 1 \rangle\rangle^k \Diamond s_4$ holds at s_0 for all $k \leq 3$. This suggests that we can synthesize three substantially different scripts for our problem.

On the other hand, assume that we want to know the degree of fault-tolerance of our configuration in the worst case (w.r.t. the choices of Player 2), where a fault is represented by the malfunction of one of the available programs. The on-line semantics tells us that if Player 2 inadvertently chooses the doc file format, Player 1 can only reach s_4 in one way. In graded ATL terms, $k = 1$ is the maximum integer such that $\langle\langle 1 \rangle\rangle^k \Diamond s_4$ holds at s_0 in the on-line semantics. Thus, the example under consideration shows no fault-tolerance in our sense, since a single fault (i.e., the absence of the "oowrite" program) can prevent Player 1 from opening the attachment.

3 Preliminaries

We consider games played by m players on a finite graph, whose set of states is partitioned in m subsets, each one corresponding to one of the players. The game starts in a state of the graph, and at each step the player who owns the current state chooses one of its outgoing edges. As a consequence, the game *moves* to the destination of that edge. The game continues in this fashion, until an infinite path is formed. Such games are called *turn-based*, as opposed to *concurrent*, since at each step only one player is responsible for the next move. Throughout the paper, we consider a fixed set Σ of *atomic propositions*. The following definitions make this framework formal.

Turn based games. A *Turn Based Game* (in the following, simply *game*) is a tuple $G = (m, S, pl, \delta, [\cdot])$ such that: $m > 0$ is the number of players; S is a finite set of states; $pl : S \to \{1, \ldots, m\}$ is a function mapping each state s to the player who owns it; $\delta \subseteq S \times S$ is the *transition relation* which provides the moves of the players and $[\cdot] : S \to 2^{\Sigma}$ is the function assigning to each state s the set of atomic propositions that are true at s. In the following, unless otherwise noted, we consider a fixed game $G = (m, S, pl, \delta, [\cdot])$. We assume that games are non-blocking, i.e., each state has at least one successor in δ, to which it can move. The players can join to form a *team* which is a subset of $\{1, \ldots, m\}$. For a team $X \subseteq \{1, \ldots, m\}$, we denote by S_X the set of states belonging to team X, i.e. $S_X = \{s \in S \mid pl(s) \in X\}$, and we denote by $\neg X$ the opposite team, i.e., $\neg X = \{1, \ldots, m\} \setminus X$. A (finite or infinite) path in G is a (finite or infinite) path in the directed graph (S, δ). Given a path ρ, we denote by $\rho(i)$ its i-th state, by $\mathit{first}(\rho)$ its first state, and by $\mathit{last}(\rho)$ its last state, when ρ is finite.

Strategies. A *strategy* in G is a pair (X, f), where $X \subseteq \{1, \ldots, m\}$ is the *team* to which the strategy belongs, and $f : S^+ \to S$ is a function such that for all $\rho \in S^+$, $(\mathit{last}(\rho), f(\rho)) \in \delta$. Our strategies are deterministic, or, in game-theoretic terms, *pure*. A strategy $\sigma = (X, f)$ is *memoryless* if $f(\rho)$ only depends on the last state of ρ, that is, for all $\rho, \rho' \in S^+$, if $\mathit{last}(\rho) = \mathit{last}(\rho')$ then $f(\rho) = f(\rho')$. We say that an infinite path $s_0 s_1 \ldots$ in G is *consistent* with a strategy $\sigma = (X, f)$ if, for all $i \geq 0$, if $s_i \in S_X$ then $s_{i+1} = f(s_0 s_1 \ldots s_i)$. Observe that the infinite paths in G which start from a state and are consistent with a given strategy of a team X form a tree where the states of team X only have one child. We denote by $\mathrm{Outc}_G(s, \sigma)$ the set of all infinite paths in G which start from s and are consistent with σ (in the following, we omit the subscript G when it is obvious from the context). For two strategies $\sigma = (X, f)$ and $\tau = (\neg X, g)$, and a state s, we denote by $\mathrm{Outc}(s, \sigma, \tau)$ the unique infinite path which starts from s and is consistent with both σ and τ.

3.1 Graded ATL: Definitions

In this subsection we give the definition of graded ATL. We extend ATL, defined in [2], by adding grading capabilities to the team quantifiers.

Syntax. Consider the *path formulas* θ and *state formulas* ψ defined via the inductive clauses below.

$$\theta ::= \bigcirc\psi \mid \psi\, U\, \psi \mid \Box\psi;$$
$$\psi ::= q \mid \neg\psi \mid \psi \vee \psi \mid \langle\!\langle X \rangle\!\rangle^k \theta,$$

where $q \in \Sigma$ is an atomic proposition, $X \subseteq \{1, \ldots, m\}$ is a team, and k is a natural number. Graded ATL is the set of all state formulas.

The operators U (until), $\Box$ (globally) and $\bigcirc$ (next) are the temporal operators. As usual, also the operator $\Diamond$ (eventually) can be introduced using the equivalence $\Diamond\psi \equiv \mathit{true}\, U\psi$. The syntax of ATL is the same as the one of graded ATL, except that the team quantifier $\langle\!\langle \cdot \rangle\!\rangle$ exhibits no natural superscript.

Semantics. We present three alternative semantics for graded ATL, called *off-line semantics*, *memoryless semantics* and *on-line semantics* for reasons explained in the Introduction. Their satisfaction relations are denoted by $\models^{\text{off}}$, $\models^{\text{mless}}$ and $\models^{\text{on}}$, respectively, and they only differ in the interpretation of the team quantifier $\langle\!\langle . \rangle\!\rangle$. We start with the operators whose meaning is invariant in all semantics. Let ρ be an infinite path in the game, s be a state, and ψ_1, ψ_2 be state formulas. For $x \in \{\text{on}, \text{off}, \text{mless}\}$, the satisfaction relations are defined as follows.

$$\begin{aligned}
&\rho \models^x \bigcirc\psi_1 && \text{iff } \rho(1) \models^x \psi_1 \\
&\rho \models^x \Box\psi_1 && \text{iff } \forall i \in \mathbb{N} \,.\, \rho(i) \models^x \psi_1 \\
&\rho \models^x \psi_1\, U\, \psi_2 && \text{iff } \exists j \in \mathbb{N} \,.\, \rho(j) \models^x \psi_2 \text{ and } \forall 0 \leq i < j \,.\, \rho(i) \models^x \psi_1 \qquad (\dagger) \\
&s \models^x q && \text{iff } q \in [s] \\
&s \models^x \neg\psi_1 && \text{iff } s \not\models^x \psi_1 \\
&s \models^x \psi_1 \vee \psi_2 && \text{iff } s \models^x \psi_1 \text{ or } s \models^x \psi_2.
\end{aligned}$$

As explained in the following, graded ATL formulas have the ability to count how many different paths (in the on-line semantics) or strategies (in the off-line semantics) satisfy a certain property. However, it is not obvious when two paths should be considered "different". For instance, consider the formula $p\, U q$, for some atomic propositions p and q, and two infinite paths that start in the same state s, where s satisfies q and not p. Both paths satisfy $p\, U q$, but only due to their initial state (i.e., $j = 0$ is the only witness for the definition (†)). Thus, we claim that these two paths should not be counted as two different ways to satisfy $p\, U q$, because they only become different *after* they have satisfied $p\, U q$. The notion of dissimilar (sets of) paths captures this intuition.

We say that two finite paths ρ and ρ' are *dissimilar* iff there exists $0 \leq i \leq \min\{|\rho|, |\rho'|\}$ such that $\rho(i) \neq \rho'(i)$. Observe that if ρ is a prefix of ρ', then ρ and ρ' are not dissimilar. For a path ρ and an integer i, we denote by $\rho_{\leq i}$ the prefix of ρ comprising $i + 1$ states, i.e. $\rho_{\leq i} = \rho(0), \rho(1), \ldots, \rho(i)$. Given $\varphi = \langle\!\langle X \rangle\!\rangle \theta$, for a path formula θ and a team $X \subseteq \{1, \ldots, m\}$, and $x \in \{\text{on}, \text{off}, \text{mless}\}$, we say that two infinite paths ρ and ρ' are (φ, x)-*dissimilar* iff:

- $\theta = \bigcirc\psi$ and $\rho(1) \neq \rho'(1)$, or
- $\theta = \Box\psi$ and $\rho(i) \neq \rho'(i)$ for some i, or
- $\theta = \psi_1 U \psi_2$ and there are two integers j and j' such that:
  - $\rho(j) \models^x \psi_2$,
  - $\rho'(j') \models^x \psi_2$,
  - for all $0 \leq i < j$, $\rho(i) \models^x \psi_1$ and $\rho(i) \models^x \langle\langle X \rangle\rangle \psi_1 U \psi_2$, and
  - for all $0 \leq i' < j'$, $\rho'(i') \models^x \psi_1$, and $\rho'(i') \models^x \langle\langle X \rangle\rangle \psi_1 U \psi_2$, and
  - $\rho_{\leq j}$ and $\rho'_{\leq j'}$ are dissimilar.

Finally, two sets of infinite paths are (φ, x)-dissimilar iff one set contains a path which is (φ, x)-dissimilar to all the paths in the other set, and, given a state s, two strategies σ_1, σ_2 are (φ, x)-*dissimilar at* s if the sets $\mathrm{Outc}(s, \sigma_1)$ and $\mathrm{Outc}(s, \sigma_2)$ are (φ, x)-dissimilar.

Off-line semantics. The meaning of the team quantifier is defined as follows, for a state s and a path formula θ.

$s \models^{\mathrm{off}} \langle\langle X \rangle\rangle^k \theta$ iff there exist k strategies $\sigma_1 = (X, f_1), \ldots, \sigma_k = (X, f_k)$ s.t. for all i, j such that $i \neq j$, σ_i and σ_j are $(\langle\langle X \rangle\rangle \theta, \mathrm{off})$-dissimilar at s and for all $\rho \in \mathrm{Outc}(s, \sigma_i)$, we have $\rho \models^{\mathrm{off}} \theta$.

Memoryless semantics. In the memoryless semantics, the meaning of the team quantifier is defined as follows, for a state s and a path formula θ.

$s \models^{\mathrm{mless}} \langle\langle X \rangle\rangle^k \theta$ iff there exist k memoryless strategies $\sigma_1 = (X, f_1), \ldots, \sigma_k = (X, f_k)$ s.t. for all i, j such that $i \neq j$, σ_i and σ_j are $(\langle\langle X \rangle\rangle \theta, \mathrm{off})$-dissimilar at s and for all $\rho \in \mathrm{Outc}(s, \sigma_i)$, we have $\rho \models^{\mathrm{mless}} \theta$.

On-line semantics. The meaning of the team quantifier is defined as follows, for a state s and a path formula θ.

$s \models^{\mathrm{on}} \langle\langle X \rangle\rangle^k \theta$ iff for all strategies $\tau = (\neg X, f)$ there exist k pairwise $(\langle\langle X \rangle\rangle \theta, \mathrm{on})$-dissimilar paths $\rho \in \mathrm{Outc}(s, \tau)$ s.t. $\rho \models^{\mathrm{on}} \theta$.

In the following we omit the superscript k of a team quantifier when $k = 1$. If φ is a classical ATL formula, we simply say in the following that a state s *satisfies* φ or, equivalently, we say that φ holds in s. Moreover, we denote by $[\![\varphi]\!] = \{s \in S \mid s \models \varphi\}$ the set of states that satisfy φ. A *simple* formula has the form $\langle\langle X \rangle\rangle \theta$, for $\theta = \Box\psi$ or $\theta = \psi_1 U \psi_2$. For a simple formula $\varphi = \langle\langle X \rangle\rangle \theta$, we denote by δ_φ the restriction of the transition function δ to $[\![\varphi]\!] \times [\![\varphi]\!]$ such that if $(s, s') \in \delta_\varphi$ and $\theta = \psi_1 U \psi_2$ then ψ_1 holds in s. For a simple formula $\varphi = \langle\langle X \rangle\rangle \theta$, a tag $x \in \{\mathrm{on}, \mathrm{off}, \mathrm{mless}\}$, and a state s, we set $\mathit{grade}^x(s, \varphi)$ to be the greatest integer k such that $s \models^x \langle\langle X \rangle\rangle^k \theta$ holds. In particular, we set $\mathit{grade}^x(s, \varphi) = 0$ if $s \not\models^x \varphi$ and $\mathit{grade}^x(s, \varphi) = \infty$ if $s \models^x \langle\langle X \rangle\rangle^k \theta$ for all $k \geq 0$. Finally, we set $\hat{\mathbb{N}} = \mathbb{N} \cup \{\infty\}$, where $\mathbb{N}$ is the set of non-negative integers.

4 Fixpoint Characterization

In this section we provide a fixpoint characterization of the functions $grade^x$, for $x \in \{\text{on}, \text{off}\}$. We start with the off-line semantics. Define the following operator $F^{\text{off}}_{\varphi} : (\llbracket\varphi\rrbracket \to \hat{\mathbb{N}}) \to (\llbracket\varphi\rrbracket \to \hat{\mathbb{N}})$.

$$F^{\text{off}}_{\varphi}(f)(s) = 1 \sqcup \begin{cases} \sum_{(s,s')\in\delta_\varphi} f(s') & \text{if } s \in S_X \\ \prod_{(s,s')\in\delta_\varphi} f(s') & \text{otherwise,} \end{cases} \tag{1}$$

where $x \sqcup y$ denotes $\max\{x, y\}$. The following result motivates the introduction of the F^{off}_{φ} operator. Observe that $grade^{\text{off}}(s, \varphi) = 0$ (and also $grade^{\text{on}}(s, \varphi) = 0$) for all $s \in S \setminus \llbracket\varphi\rrbracket$.

Lemma 1. *Let φ be a simple formula and $f : \llbracket\varphi\rrbracket \to \hat{\mathbb{N}}$ be such that $f(s) = grade^{\text{off}}(s, \varphi)$, for $s \in \llbracket\varphi\rrbracket$. The function f is the least fixpoint of F^{off}_{φ}.*

Proof. First, we prove that f is a fixpoint of F^{off}_{φ}. Let $\varphi = \langle\langle X\rangle\rangle\theta$ and $s \in \llbracket\varphi\rrbracket$, and, for all successors s_i of s such that $(s, s_i) \in \delta_\varphi$, let us set $k_i = grade^{\text{off}}(s_i, \varphi)$. That is, k_i strategies of team X exist which determine k_i (φ, off)-dissimilar sets of paths, consistent with the strategies and satisfying θ. If $s \in S_X$, then the total number of winning strategies for X from s is the sum of the k_i's. Indeed, each winning strategy starting from s_i remains winning if started from s (if $\theta = \psi_1 U \psi_2$, it is essential the hypothesis that $s \models \psi_1$ ensured by the definition of δ_φ). If $s \notin S_X$, then for each s_i, the players of the team X can choose one of the k_i dissimilar winning strategies. Each combination gives rise to a winning strategy from s, that is dissimilar to the one obtained by any other combination. Therefore, the total number of dissimilar winning strategies from s is the product of the k_i's.

Next, we prove that f is the *least* fixpoint of F^{off}_{φ}. Precisely, we prove by induction on n the following statement: Let g be a fixpoint of F^{off}_{φ} and let $s \in \llbracket\varphi\rrbracket$, if $g(s) \leq n$ then $f(s) \leq g(s)$. Assume that $\theta = \Box\psi$ (the other case is similar). If $n = 1$, by hypothesis $g(s) = 1$. Considering the definition of F^{off}_{φ}, there are the following three possibilities: *(i)* s has no successors according to δ_φ; *(ii)* s belongs to S_X and has only one successor in δ_φ; *(iii)* s does not belong to S_X and $g(t) = 1$, for all states t such that $(s, t) \in \delta_\varphi$. Option *(i)* can be discarded because $\llbracket\varphi\rrbracket$ is the set of states where $\langle\langle X\rangle\rangle\Box\psi$ holds, and thus each state in $\llbracket\varphi\rrbracket$ has at least one successor in δ_φ. Given the remaining two options, one can see that $\neg X$ can force the game in a loop where all states x have value $g(x) = 1$, and the players of X cannot exit this loop. Accordingly, we have $f(s) = 1$, as requested. If $n > 1$, by contradiction, let g be a fixpoint of F^{off}_{φ} which is smaller than f. I.e., there is a state $s \in \llbracket\varphi\rrbracket$ such that $g(s) < f(s)$. Clearly, it must be $f(s) > 1$. Assume w.l.o.g. that also $g(s) > 1$, otherwise proceed as in the case for $n = 1$. Starting from s, build a path in the game in the following way. Let t be the current last state of the path (at the beginning, $t = s$): if t has only one successor u according to δ_φ, pick u as the next state (notice that $g(u) = g(t)$ and $f(u) = f(t)$); if $t \notin S_X$ and t has more than one successor according to δ_φ, pick

as the next state of the path a successor u such that $g(u) < f(u)$ (it is a simple matter of algebra to show that such a state exists); finally, if $t \in S_X$ and t has more than one successor according to δ_φ, stop. If the above process continues forever, it means that the adversaries (players not in X) can force the game in a loop from which players in X cannot exit. This means that $f(s) = 1$, which is a contradiction. Otherwise, the above process stops in a state $t \in S_X$, such that $g(t) \leq g(s)$ and $g(t) < f(t)$. Since t has more than one successor, by (1), for all successors u of t we have $g(u) < g(t) \leq g(s) \leq n$ and thus $g(u) \leq n - 1$. Moreover, there is a successor u^* of t such that $g(u^*) < f(u^*)$. On the other hand, by inductive hypothesis $g(u^*) \geq f(u^*)$, which is a contradiction.

Now, we provide a similar characterization for the on-line semantics. Define the following operator $F^{\text{on}}_\varphi : (\llbracket\varphi\rrbracket \to \hat{\mathbb{N}}) \to (\llbracket\varphi\rrbracket \to \hat{\mathbb{N}})$.

$$F^{\text{on}}_\varphi(f)(s) = 1 \sqcup \begin{cases} \sum_{(s,s')\in\delta_\varphi} f(s') & \text{if } s \in S_X \\ \min_{(s,s')\in\delta_\varphi} f(s') & \text{otherwise.} \end{cases} \tag{2}$$

Lemma 2. *Let φ be a simple formula and $f : \llbracket\varphi\rrbracket \to \hat{\mathbb{N}}$ be such that $f(s) = grade^{\text{on}}(s, \varphi)$, for $s \in \llbracket\varphi\rrbracket$. The function f is the least fixpoint of F^{on}_φ.*

Proof. First, we prove that f is a fixpoint of F^{on}_φ. Let $\varphi = \langle\langle X\rangle\rangle\theta$ and $s \in \llbracket\varphi\rrbracket$. Suppose that there is at least one successor of s in δ_φ (otherwise $\theta = \psi_1 U \psi_2$, $s \models^{\text{on}} \psi_2$, and $F^{\text{on}}_\varphi(f)(s) = f(s) = 1$). For all successors s_i of s in δ_φ, let $k_i = grade^{\text{on}}(s_i, \langle\langle X\rangle\rangle\theta)$. For each strategy τ of $\neg X$, there are k_i dissimilar paths starting from s_i, consistent with τ, and satisfying θ. Therefore, if $s \in S_X$, by adding state s in front of each of these paths, we obtain $\sum_i k_i$ dissimilar paths starting from s, consistent with τ, and satisfying θ. In fact, if $\theta = \Box\psi$, since $s \models^{\text{on}} \langle\langle X\rangle\rangle\theta$, we have that $s \models^{\text{on}} \psi$, while, if $\theta = \psi_1 U \psi_2$, then $s \models^{\text{on}} \psi_1$ (if it is not the case, there are no successors s_i of s such that $(s, s_i) \in \delta_\varphi$). If instead $s \notin S_X$, let $i = \arg(\min_j k_j)$. Consider the memoryless strategy τ of $\neg X$ that picks s_i when the game is in s. Under τ, there are k_i dissimilar paths starting from s and satisfying θ. From the choice of i, it follows that all strategies of $\neg X$ have at least as many dissimilar paths from s.

Next, we prove that f is the *least* fixpoint of F^{on}_φ. Similarly to the proof of Lemma 1, we prove by induction on n the following statement: Let g be a fixpoint of F^{on}_φ and let $s \in \llbracket\varphi\rrbracket$, if $g(s) \leq n$ then $f(s) \leq g(s)$. Assume for simplicity that $\theta = \Box\psi$, as the other case can be proved along similar lines. The case for $n = 1$ can be proved similarly to the proof of Lemma 1. If $n > 1$, by contradiction, let g be a fixpoint of F^{on}_φ which is smaller than f. I.e., there is a state $s \in \llbracket\varphi\rrbracket$ such that $g(s) < f(s)$. Clearly, it must be $f(s) > 1$. Starting from s, build a path in the game in the following way. Let t be the current last state of the path (at the beginning, $t = s$): if $t \notin S_X$, pick as the next state of the path a successor u of t such that $g(u) = g(t)$ and $(t, u) \in \delta_\varphi$ (notice that $f(u) \geq f(t)$); if $t \in S_X$ and t has only one successor u with $(t, u) \in \delta_\varphi$, pick u as the next state (notice that $g(u) = g(t)$ and $f(u) = f(t)$); finally, if $t \in S_X$ and t has more than one such successor, stop. If the above process continues forever, team $\neg X$ can force

the game in a loop from which team X cannot exit. This means that $f(s) = 1$, which is a contradiction. Otherwise, the above process stops in a state $t \in S_1$, such that $g(t) = g(s)$ and $f(t) \geq f(s)$. Therefore, $f(t) > g(t)$. Since t has more than one successor according to δ_φ, by (2), for all successors u of t we have $g(u) < g(t) = g(s) \leq n$ and thus $g(u) \leq n - 1$. Moreover, there is a successor u^* of t such that $g(u^*) < f(u^*)$. On the other hand, by inductive hypothesis $g(u^*) \geq f(u^*)$, which is a contradiction.

5 Comparing the Semantics

The example in Figure 1 shows that the three semantics are different in general. In this section we clarify the relationship between the semantics. In the following examples we consider two players, Player 1 and Player 2. As before, states of Player 1 are represented by circles and those of Player 2 by squares.

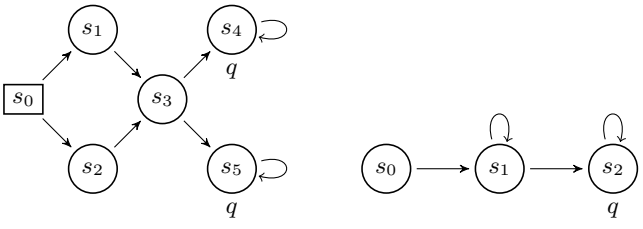


Fig. 2. Two games where the semantics differ

The first example shows that the memoryless semantics can give a smaller grading value than the offline semantics, even when the latter produces a finite value.

Example 1. Consider the game in Figure 2a, where the goal for Player 1 is to reach the proposition q, which is true in states s_4 and s_5. According to the off-line semantics, there are 4 possible strategies to achieve that goal. Namely, for each choice of Player 2 in s_0, Player 1 has two options once the game is in s_3. Thus, we have $s_0 \models^{\text{off}} \langle\langle 1 \rangle\rangle^4 \Diamond q$ and $s_0 \not\models^{\text{off}} \langle\langle 1 \rangle\rangle^5 \Diamond q$. On the other hand, according to the memoryless semantics, there are only two memoryless strategies for Player 1, the one that leads to s_4 and the one leading to s_5. Thus, $s_0 \models^{\text{mless}} \langle\langle 1 \rangle\rangle^2 \Diamond q$ and $s_0 \not\models^{\text{mless}} \langle\langle 1 \rangle\rangle^3 \Diamond q$.

The example in the introduction shows that the memoryless semantics may attribute to a formula a higher grading value than the on-line semantics. The following example shows that the converse is also possible, hence proving that the two semantics are incomparable.

Example 2. Consider the game in Figure 2b, where the goal for Player 1 is again to reach the proposition q, which is true in s_2. According to the on-line

semantics, there are infinitely many strategies to achieve that goal. For all $k > 0$, there is a strategy of Player 1 that makes k visits to s_1 before going to s_2. Thus, we have $s_0 \models^{\text{on}} \langle\langle 1 \rangle\rangle^k \Diamond q$, for all $k > 0$. On the other hand, there is only one memoryless winning strategy, i.e., the one that goes directly from s_1 to s_2. Thus, $s_0 \models^{\text{mless}} \langle\langle 1 \rangle\rangle^1 \Diamond q$ and $s_0 \not\models^{\text{mless}} \langle\langle 1 \rangle\rangle^2 \Diamond q$.

When all quantifiers have grade 1, the semantics coincide. Indeed, the classical quantifiers embedded in each ATL team quantifier (there is a strategy of team X such that for all strategies of team $\neg X$, etc.) can be exchanged, due to the well-known result by Martin on the determinacy of games with Borel objectives [18]. Notice that the languages of infinite words defined by the linear part of ATL are trivially Borel languages. This leads to the following result.

Theorem 1. *For all states s and* ATL *state formulas φ, it holds that*

$$s \models^{\text{off}} \varphi \quad \textit{iff} \quad s \models^{mless} \varphi \quad \textit{iff} \quad s \models^{\text{on}} \varphi.$$

Now we prove that, if a graded ATL formula in which the negation does not occur is satisfied under the on-line semantics, then it is satisfied under the off-line semantics as well. Observe that the same result cannot hold for a general graded ATL formula, since we know that the off-line and the on-line semantics do not coincide. For example, for the game in Figure 1, we have that $s_0 \models^{\text{on}} \neg\langle\langle 1 \rangle\rangle^2 \Diamond s_4$, but, on the contrary, it is false that $s_0 \models^{\text{off}} \neg\langle\langle 1 \rangle\rangle^2 \Diamond s_4$. Let us consider the F_φ^{off} and F_φ^{on} operators, defined in the previous section, and their iteration, starting from the constant function 1: $F_\varphi^{x,0}(s) = 1$ and $F_\varphi^{x,i+1}(s) = F_\varphi^x(F_\varphi^{x,i})(s)$, for $x \in \{\text{on}, \text{off}\}$ and an ATL formula φ. It is easy to see that both the sequences $F_\varphi^{\text{on},i}(s)$ and $F_\varphi^{\text{off},i}(s)$ are nondecreasing, for every state s, and then, by Lemma 1 and Lemma 2, the following proposition follows.

Proposition 1. *Let φ be a simple formula and let $s \in [\![\varphi]\!]$.*

- *The value $grade^x(s, \varphi)$ is the least upper bound of the sequence $\{F_\varphi^{x,i}(s)\}_{i \geq 0}$, for $x \in \{\text{on}, \text{off}\}$.*
- *For every $i \geq 0$, $F_\varphi^{\text{on},i}(s) \leq F_\varphi^{\text{off},i}(s)$.*

Theorem 2. *For all states s and negation-free graded* ATL *formulas ψ,*

$$\textit{if} \quad s \models^{\text{on}} \psi \quad \textit{then} \quad s \models^{\text{off}} \psi.$$

Proof. Let $\psi = \langle\langle X \rangle\rangle^k \theta$, with either $\theta = \Box q$ or $\theta = p\,U\,q$, $p, q \in \Sigma$, and $\varphi = \langle\langle X \rangle\rangle \theta$. From Proposition 1, $grade^{\text{on}}(s, \varphi) \leq grade^{\text{off}}(s, \varphi)$, otherwise $grade^{\text{on}}(s, \varphi)$ would not be the least upper bound of $\{F_\varphi^{\text{on},i}(s)\}_{i \geq 0}$. Thus $s \models^{\text{on}} \psi$ only if $s \models^{\text{off}} \psi$. To complete the proof of our statement, we proceed by structural induction on a generic negation-free graded ATL formula. The proof is trivial for the atomic propositions and for the disjunction operator. Let ψ be a graded ATL formula for which we inductively suppose that if $r \models^{\text{on}} \psi$ then $r \models^{\text{off}} \psi$, for any state r of G. If $s \models^{\text{on}} \langle\langle X \rangle\rangle^k \bigcirc \psi$, the statement trivially follows. Suppose now that $s \models^{\text{on}} \langle\langle X \rangle\rangle^k \Box \psi$ and let $\hat{G}$ be a new game obtained from G by adding a

new atomic proposition q_ψ, holding true in all the states r such that $r \models^{\text{on}} \psi$. Clearly, $s \models^{\text{on}}_{\hat{G}} \langle\langle X \rangle\rangle^k \Box q_\psi$ and, as shown above, $s \models^{\text{off}}_{\hat{G}} \langle\langle X \rangle\rangle^k \Box q_\psi$. This implies that $s \models^{\text{off}} \langle\langle X \rangle\rangle^k \Box \psi$ as well. The proof for the U operator is similar.

6 Model Checking

The model checking problem for the semantics $x \in \{\text{on}, \text{off}, \text{mless}\}$ takes as input a game G, a state s in G and a graded ATL formula ψ, and asks whether $s \models^x \psi$. In this section, we efficiently solve the model checking problem for all the considered semantics, in polynomial time w.r.t. the size of the input game and the number of logical operators in the input formula. Moreover, for the off-line and the on-line semantics, the time complexities do not depend on the constants occurring in the graded team quantifiers.

We say that a state is a *decision point for* X (or simply a *decision point*, when the team X is clear from the context) if it belongs to a player of team X and it has at least two successors. Moreover, a strongly connected component of a graph is a *sink* if there are no outgoing edges from it.

Off-line semantics. We first consider $\psi = \langle\langle X \rangle\rangle^k \theta$ with $\theta = \Box q$ or $\theta = p\, U\, q$, and provide algorithms for solving a stronger form of model checking, that is we compute $grade^x(s, \langle\langle X \rangle\rangle \theta)$.

Algorithm 1. The algorithm computing $grade^{\text{off}}(\cdot, \varphi)$, given $\varphi = \langle\langle X \rangle\rangle \theta$, with $\theta = \Box q$ or $\theta = p\, U\, q$.

---

1. Using standard ATL algorithms, compute the set of states $[\![\varphi]\!]$, and assign 0 to the states in $S \setminus [\![\varphi]\!]$. Then, compute the subgame with state-space $[\![\varphi]\!]$ and transition relation δ_φ.
2. On the sub-game, compute the strongly connected components.
3. Proceed backwards starting from the sink components, according to the following rules:
   (a) Sink components which do not contain decision points are assigned grade 1.
   (b) Sink and non-sink components having more than one state and containing a decision point (of the subgame) are assigned grade ∞.
   (c) Non-sink components having more than one state and do not fall in case 3b are assigned ∞ if they have a successor component with grade greater than 1; otherwise, they are assigned 1.
   (d) Non-sink components containing only one state: if this state belongs to S_X then it is assigned the sum of the grades of the successor components; while if the state does not belong to S_X, then it is assigned the product of the grades of the successor components.

Lemma 3. *For each state s, Algorithm 1 computes $grade^{\text{off}}(s, \varphi)$, for $\varphi = \langle\langle X \rangle\rangle \theta$, with $\theta = \Box q$ or $\theta = p\, U\, q$. The algorithm runs in linear time.*

Proof. The algorithm first computes the states satisfying the ATL formula $\langle\langle X\rangle\rangle\theta$, and removes all other states s, for which it indeed holds $grade^{\text{off}}(s, \langle\langle X\rangle\rangle\theta) = 0$. Then, it computes in the new game the strongly connected components (we assume that there exists at least one such component, otherwise the statement trivially holds). Observe that all the states belonging to the same strongly connected component have the same grade. The algorithm then looks for sink components not containing a decision point, assigning the value 1 to them. Components having more than one state and containing a decision point, get the value ∞. Let us prove that this is correct. Let r be a decision point and suppose that r belongs to Player j, and call r_1, r_2 two of its successors. For all $h > 0$ there is a strategy of team X such that Player j, in state r, chooses to visit r_1 h times, before visiting r_2. Clearly, for each $h > 0$, the strategies σ_i, $i \leq h$, determine pairwise $(\langle\langle X\rangle\rangle\theta, \text{off})$-dissimilar $\text{Outc}(r_2, \sigma_i)$ and thus, $r_2 \models^{\text{off}} \langle\langle X\rangle\rangle^k\theta$, for all $k > 0$. The same reasoning holds for non-sink components and, thus, steps 3a and 3b are correct. Consider now a non-sink component C having more than one state and not containing a decision point (step 3c). Edges outgoing from C are moves of players not belonging to X and thus, if the algorithm has assigned 1 to all the successor components of C, there is only one strategy for the team X. Otherwise, suppose that there is a state r in C having a successor r' in another component and that there exist two strategies of X starting from r'. Then, for any way of alternating between these two strategies, whenever the state r' is entered, there is a strategy of X from r, and thus the algorithm correctly assigns grade infinite. The correctness of case 3d follows from Lemma 1. Finally, observe that the algorithm is complete as all cases have been examined and assuming an adjacency list representation for the game, the above algorithm runs in linear time.

To solve the model checking problem for graded ATL we can use Lemma 3 thus, the following theorem holds. The complexity result assumes that each basic operation on integers is performed in constant time.

Theorem 3. *Given a state s and a graded* ATL *formula ψ, the graded model checking problem, $s \models^{\text{off}} \psi$, can be solved in time $\mathcal{O}(|\delta| \cdot |\psi|)$, where $|\psi|$ is the number of operators occurring in ψ.*

Memoryless semantics. For $\theta \in \{\Box q, p\,U\,q\}$, in order to model-check a graded ATL formula $\langle\langle X\rangle\rangle^k\theta$ on a state s in the memoryless semantics, we call the function count_mless(G, φ, k, s) (Algorithm 2), with $\varphi = \langle\langle X\rangle\rangle\theta$. We have that s satisfies $\langle\langle X\rangle\rangle^k\theta$ if and only if the result of this call is k. To describe Algorithm 2, we need some extra definitions. Given an ATL formula, we say that a state s is *winning* w.r.t. φ if there exists a strategy σ of team X such that for all $\rho \in \text{Outc}(s, \sigma)$, we have $\rho \models \theta$ (i.e., $s \models \varphi$). In that case we also say that σ is *winning from s*. We say that a strategy is *uniformly winning* if it is winning from all winning states. For two states s and u, and a strategy σ, we say that the *distance of u from s according to σ* is the shortest distance between s and u considering only paths consistent with σ.

In general, count_mless$(G, \varphi = \langle\langle X\rangle\rangle\theta, k, s)$ computes the minimum between k and the number of memoryless strategies of team X, (φ, off)-dissimilar at s,

that are winning from s. The idea of the algorithm is the following. We start by computing the set of states W where the ATL formula φ holds, using standard ATL algorithms (line 1). If s does not belong to W, it does not satisfy $\langle\langle X\rangle\rangle^k\theta$, for any $k > 0$. If s belongs to W, we analyze the subgame with state-space W and transition relation δ_φ (see Section 3.1). On this subgame, we compute an arbitrary memoryless uniformly winning strategy. After removing the edge $(u, \pi(u))$ on line 8, every strategy π' in the residual game must be distinct from π, because it cannot use that edge. When $\theta = pUq$, two distinct infinite paths that satisfy θ need not be dissimilar. It is necessary that the paths become distinct before an occurrence of q. This property is ensured by the subgame only containing winning states and by all the computed strategies being uniformly winning.

We then put back the edge $(u, \pi(u))$ and on line 12 we remove all other edges leaving u. This ensures that the strategies computed in the following iterations are dissimilar from the ones computed so far. This structure is inspired by the algorithms for computing the k shortest simple paths in a graph [21,16].

The following result establishes an upper bound on the value returned by count_mless, showing that it cannot return a value higher than the number of mutually dissimilar memoryless winning strategies present in the game. Due to space constraints, we omit the proof and refer the reader to the extended version of this paper.

Lemma 4. *All strategies computed on line 4 by Algorithm 2 are mutually* (φ, off)*-dissimilar at* s*.*

The following result provides a lower bound on the value returned by count_mless. Together with Lemma 4, this result implies the correctness of the algorithm.

Lemma 5. *Given a state* s*, if there are* n *memoryless strategies that are mutually* (φ, off)*-dissimilar at* s*, and that are winning from* s*, then* $\mathit{count\_mless}(G, \varphi, k, s)$ *returns at least* n*, for all* $k \geq n$*.*

The following result characterizes the time complexity of Algorithm 2, in terms of calls to the procedures get_winning_set and get_uniformly_winning_strategy.

Lemma 6. *A call to* $\mathit{count\_mless}(G, \varphi, k, s)$ *which returns value* $n > 0$ *makes at most* $1 + n \cdot |S|$ *calls to get_winning_set and at most* n *calls to get_uniformly_winning_strategy.*

Proof. We proceed by induction on n. For $n = 0$, the statement is trivially true, because the value zero can only be returned on line 2, after one call to get_winning_set.

For $n > 0$, if the algorithm returns on line 5, the statement is trivially true. Otherwise, the algorithm enters the "for all" loop after one call to get_winning_set and one call to get_uniformly_winning_strategy. Let n_i be the value returned by the i-th recursive call on line 9. We have that $n = 1 + \sum_i n_i$ and the number of iterations of the loop is at most $|S|$. By inductive hypothesis, the i-th recursive call is responsible for at most $1 + n_i \cdot |S|$ calls to get_winning_set and at most

n_i calls to get_uniformly_winning_strategy. Hence, the total number of calls to get_winning_set is

$$1 + \sum_i (1 + n_i \cdot |S|) = 1 + \sum_i 1 + |S| \sum_i n_i \leq 1 + |S| + |S| \cdot (n-1) = 1 + n \cdot |S|.$$

The total number of calls to get_uniformly_winning_strategy is instead $1 + \sum_i n_i = n$, as required.

Considering that ATL model checking can be performed in linear time w.r.t. the adjacency list representation of the game, from Lemma 6 we obtain the following.

Corollary 1. *The time complexity of count_mless*(G, φ, k, s) *is* $\mathcal{O}(k \cdot |S| \cdot (|S| + |\delta|)) = \mathcal{O}(k \cdot |S| \cdot |\delta|)$.

Algorithm 2. The procedure count_mless(G, φ, k, s).

Require: $G = (m, S, pl, \delta, [\cdot])$: game, $\varphi \in \{\langle\langle X \rangle\rangle \Box q, \langle\langle X \rangle\rangle p\, U q\}$, k: natural number, s: state of G

1: $W :=$ get_winning_set(G, φ)
2: **if** $s \notin W$ **then return** 0
3: $G' := (m, S, pl, \delta_\varphi, [\cdot])$
4: $\pi :=$ get_uniformly_winning_strategy(G', φ)
5: **if** $k = 1$ **then return** 1
6: $n := 1$
7: **for all** decision points u of team X, reachable from s according to π, in non-decreasing order of distance from s according to π **do**
8: remove_edge$(G', (u, \pi(u)))$
9: $n := n +$ count_mless$(G', \varphi, k - n, s)$
10: add_edge$(G', (u, \pi(u)))$
11: **if** $n = k$ **then return** n
12: remove_edges$(G', \{(u, x) \mid x \neq \pi(u)\})$
13: **end for**
14: **return** n

From the previous lemmas and by using standard arguments, we obtain a solution to the model-checking problem for a graded ATL formula, under the memoryless semantics.

Theorem 4. *Given a state s and a graded* ATL *formula ψ, the graded model checking problem, in the memoryless semantics, can be solved in time $\mathcal{O}(\hat{k} \cdot |S| \cdot |\delta| \cdot |\psi|)$, where $\hat{k}$ is the maximum value of a constant appearing in ψ.*

On-line semantics. Similarly to the case of off-line semantics, we describe an algorithm for computing $grade^{\text{on}}(s, \langle\langle X \rangle\rangle \theta)$ for $\theta = \Box q$ or $\theta = p\, U q$, and for all states $s \in S$. Given a path formula $\theta = \Box q$ or $\theta = p\, U q$, Algorithm 3 computes $grade^{\text{on}}(s, \langle\langle X \rangle\rangle \theta)$, for all states $s \in S$. The complexity of the algorithm

Algorithm 3. The algorithm computing $grade^{\text{on}}(\cdot, \varphi)$, given $\varphi = \langle\langle X \rangle\rangle \theta$, with $\theta = \Box q$ or $\theta = p\, U q$.

1. Using standard ATL algorithms, compute the set of states $[\![\varphi]\!]$. The following steps are performed in the subgame with state-space $[\![\varphi]\!]$ and transition relation δ_φ. Assign grade 0 to the states in $S \setminus [\![\varphi]\!]$.
2. Let d be a new atomic proposition which holds in the decision points (of the subgame). Find the states where $\langle\langle \neg X \rangle\rangle \Box \neg d$ holds, and assign grade 1 to them.
3. Find the states from which team X can enforce infinitely many visits to a decision point (i.e., states where the ATL* formula $\langle\langle X \rangle\rangle \Box \Diamond d$ holds), and assign grade ∞ to them.
4. For the remaining states, compute their value by inductively applying equation (2) to those states whose successors have already been assigned a value.

is dominated by step 3, which involves the solution of a Büchi game [19]. This task can be performed in time $\mathcal{O}(|S| \cdot |\delta|)$, i.e., quadratic in the size of the adjacency-list representation of the game.

It is not obvious that the algorithm assigns a value to each state in the game. Indeed, step 4 assigns a value to a state only if all of its successors have already received a value. If, at some point, each state that does not have a value has a successor that in turn does not have a value, the algorithm stops. For the above situation to arise, there must be a loop of states with no value. The following lemma states that the above situation cannot arise, and therefore that the algorithm ultimately assigns a value to each state.

Lemma 7. *At the end of step 3 of Algorithm 3, there is no loop of states with no value.*

We can now state the correctness and complexity of Algorithm 3.

Lemma 8. *Given a path formula $\theta = \Box q$ or $\theta = p\, U q$, at the end of Algorithm 3, each state s has value $grade^{\text{on}}(s, \langle\langle X \rangle\rangle \theta)$. The algorithm runs in quadratic time.*

Proof. We proceed by examining the four steps of the algorithm. If s receives its value (zero) during step 1, it means that $s \not\models^{\text{on}} \langle\langle X \rangle\rangle \theta$. Therefore, zero is the largest integer k such that $s \models^{\text{on}} \langle\langle X \rangle\rangle^k \theta$ holds.

If s receives its value (one) during step 2, it means that $s \models^{\text{on}} \langle\langle \neg X \rangle\rangle \Box \neg d$. Consider a strategy of team $\neg X$ ensuring the truth of $\Box \neg d$. According to this strategy, a player of the team X can never choose between two different successors. Therefore, there is a unique infinite path consistent with this strategy of $\neg X$. This implies that 1 is the greatest integer k such that $s \models^{\text{on}} \langle\langle X \rangle\rangle^k \theta$ holds.

If s receives its value (infinity) during step 3, it means that $s \models^{\text{on}} \langle\langle X \rangle\rangle \Box \Diamond d$. Consider any strategy τ of $\neg X$, and a strategy σ of X ensuring $\Box \Diamond d$. The resulting infinite path ρ contains infinitely many decision points for X. For each decision point $\rho(i)$, let σ_i be a strategy of X with the following properties: *(i)* σ_i coincides with σ until the prefix $\rho_{\leq i}$ is formed, *(ii)* after $\rho_{\leq i}$, σ_i picks a different successor than σ, and then keeps ensuring θ. It is possible to find such a σ_i

because $\rho(i)$ is a decision point in the subgame. For all $i \neq j$ such that $\rho(i)$ and $\rho(j)$ are decision points, the outcome of τ and σ_i is dissimilar from the outcome of τ and σ_j. Therefore, $s \models^{\text{on}} \langle\langle X \rangle\rangle^k \theta$ holds for all $k > 0$.

Finally, if s receives its value during step 4, the correctness of the value is a consequence of Lemma 2. The complexity of the algorithm is discussed previously in this section.

Due to the above complexity result, and the discussion already made for the off-line semantics, we obtain the following conclusion.

Theorem 5. *Given a state s and a graded* ATL *formula ψ, the graded model checking problem, $s \models^{\text{on}} \psi$, can be solved in time $\mathcal{O}(|S| \cdot |\delta| \cdot |\psi|)$, where $|\psi|$ is the number of operators occurring in ψ.*

As before, under the constant-time assumption for basic integer operations, the above complexity is independent of the integer constants appearing in the formula. Finally, from the PTIME hardness of the reachability problem for AND-OR graphs [14], this corollary follows.

Corollary 2. *The graded* ATL *model checking problem is PTIME-complete, under the off-line, memoryless and on-line semantics, with respect to the size of the game.*

7 Model Checking Concurrent Games

In this section we consider graded ATL when interpreted over concurrent games, i.e., games where at each step all players contribute to the choice of the next move. We show that the model-checking problem for concurrent games can be reduced to the turn-based setting.

Concurrent game structures. A *concurrent* game structure is a tuple $G = (m, S, d, \delta, [\cdot])$ where: $m > 0$ is the number of players; S is a finite set of states; for each player $i \in \{1, \ldots, m\}$ and state $s \in S$, $d_i(s) \geq 1$ is an integer representing the number of moves of player i at state s; and $[\cdot] : S \to 2^{\Sigma}$ is the function assigning to each state s the set of atomic propositions that are true at s. In the following, we use integers from 1 to $d_i(s)$ for the moves of player i in state s. In a state s, the vector $\langle j_1, \ldots, j_m \rangle$ is the *move vector* such that $j_i \leq d_i(s)$. For a state s, we define the *move function* $D(s) = \{1, \ldots, d_1(s)\} \times \ldots \times \{1, \ldots, d_m(s)\}$ that lists all the joint moves available to the players. The transition function assigns to each $s \in S$ and $j \in D(s)$ the state $\delta(s, j)$.

Strategies can be defined naturally as in the case of the turn-based setting. In particular, a strategy of player i assigns a move in the range $1, \ldots, d_i(s)$ to each run ending in state s. As far as the dissimilarity is concerned, we have to decide whether moves are sufficient for distinguishing two strategies (or paths). In the following, we answer the above question in the positive: two strategies (or paths) that only differ in the moves chosen by a player, and not in the sequences of states, are considered different, and hence potentially dissimilar.

This corresponds to the assumption that different moves in the game represent different real-world actions, that we are interested in counting. The satisfaction relations $\models^{\text{off}}$ and $\models^{\text{on}}$ are defined accordingly.

Model-checking complexity. We briefly report the construction of a 2-player turn-based game G_X from a concurrent game G played by the team X [2]. Consider a concurrent game structure $G = (m, S, d, \delta, [\cdot])$ and a team X of players in $\{1, \ldots, m\}$. For a state $s \in S$, an X-*move* is a possible combination of moves of the players in X when the game is in state s. We denote by $C(X, s)$ the set of X-moves in s, and by $C(X) = \cup_{s \in S} C(X, s)$ the set of all X-moves. For an X-move c in s, a state s' is said to be a *c-successor* of s if $s' = \delta(s, j)$, where $j = \langle j_1, \ldots, j_m \rangle$ and each j_a, for $a \in X$, is determined by the X-move c.

The 2-player turn-based game structure $G_X = (2, S', pl, \delta', [\cdot]')$ is defined as follows: the set of atomic propositions is augmented with a special proposition *aux*, the set of states is $S' = S \cup C(X)$, that is it contains S and, for each X-move, a new state which is now labeled with *aux* by the labeling function (the states in S have the same label as in G). Player 1 owns the states $s \in S$, while Player 2 owns the new states, and the behaviour is the following: there is an edge from a state $s \in S$ to each state $c \in C(X, s)$, and there is an edge from $c \in C(X, s)$, for some s, to $s' \in S$ if s' is a c-successor of s. Clearly G_X has $\mathcal{O}(|\delta|)$ states and edges. Moreover it is easy to see that for each strategy σ of team X in G there exists a corresponding strategy σ' in G_X such that every path π' in $\text{Outc}_{G_X}(s, \sigma')$ is of the type $\ldots, s_i, a_i, s_{i+1}, a_{i+1}, \ldots$, where the s states are in S, and the a states are in $C(X)$, and π' uniquely corresponds to a path $(\ldots, s_i, s_{i+1}, \ldots) \in \text{Outc}_G(s, \sigma)$. Consider now a path formula $\theta = \Box p$ or $\theta = p\, U q$, for atomic propositions p and q. The considerations above imply that the number of $(\langle\langle X \rangle\rangle \theta, \text{off})$-dissimilar strategies of X in G is equal to the number of $(\langle\langle 1 \rangle\rangle \theta, \text{off})$-dissimilar strategies of Player 1 in G_X.

Proposition 2. *Let G be a concurrent game, $s \in S$, $p, q \in \Sigma$, and $X \subseteq \{1, \ldots, m\}$. Then the following hold for all $k > 0$:*

$$s \models^{\text{off}}_G \langle\langle X \rangle\rangle^k \Box p \quad \textit{iff} \quad s \models^{\text{off}}_{G_X} \langle\langle 1 \rangle\rangle^k \Box (p \vee \text{aux})$$
$$s \models^{\text{off}}_G \langle\langle X \rangle\rangle^k p\, U q \quad \textit{iff} \quad s \models^{\text{off}}_{G_X} \langle\langle 1 \rangle\rangle^k (p \vee \text{aux})\, U q.$$

For the on-line case, we consider the 2-player turn-based game $G_{\neg X}$ in which Player 1 plays the role of the team $\neg X$ of the concurrent game G. Thus the number of $(\langle\langle \neg X \rangle\rangle \theta, \text{on})$-dissimilar paths in each $\text{Outc}_G(s, \sigma)$, for a strategy σ of team $\neg X$ is equal to the number of $(\langle\langle 1 \rangle\rangle \theta, \text{on})$-dissimilar paths in $\text{Outc}_{G_{\neg X}}(s, \sigma')$ for the corresponding strategy σ' of player 1 in $G_{\neg X}$.

Proposition 3. *Let G be a concurrent game, $s \in S$, $p, q \in \Sigma$ and $X \subseteq \{1, \ldots, m\}$. Then the following hold for all $k > 0$:*

$$s \models^{\text{on}}_G \langle\langle X \rangle\rangle^k \Box p \quad \textit{iff} \quad s \models^{\text{on}}_{G_{\neg X}} \langle\langle 2 \rangle\rangle^k \Box (p \vee \text{ aux})$$
$$s \models^{\text{on}}_G \langle\langle X \rangle\rangle^k p\, U q \quad \textit{iff} \quad s \models^{\text{on}}_{G_{\neg X}} \langle\langle 2 \rangle\rangle^k (p \vee \text{aux})\, U q.$$

Let us remark that the above propositions allow us to model check a formula ψ with one team quantifier $\langle\langle X \rangle\rangle$ by using the algorithms for the turn-based games given in the previuos sections. By applying standard techniques one can model-check formulas with any number of nested team quantifiers. A *PTIME*-completeness result thus follows from Corollary 2 and the construction of the 2-player turn-based game described above.

Theorem 6. *The model-checking problem for graded* ATL *on concurrent games is PTIME-complete with respect to the size of the game. Moreover, it can be solved in time* $\mathcal{O}(|\delta| \cdot |\psi|)$ *in the off-line semantics and in time* $\mathcal{O}(|\delta|^2 \cdot |\psi|)$ *in the on-line semantics, for a concurrent game with transition function* δ *and for a graded* ATL *formula* ψ.

References

1. Ågotnes, T., Goranko, V., Jamroga, W.: Alternating-time temporal logics with irrevocable strategies. In: Proc. of TARK 2007, pp. 15–24. ACM, New York (2007)
2. Alur, R., Henzinger, T.A., Kupferman, O.: Alternating-time temporal logic. J. ACM 49, 672–713 (2002)
3. Alur, R., Henzinger, T.A., Mang, F.Y.C., Qadeer, S., Rajamani, S.K., Tasiran, S.: Mocha: modularity in model checking. In: Y. Vardi, M. (ed.) CAV 1998. LNCS, vol. 1427, pp. 521–525. Springer, Heidelberg (1998)
4. Bianco, A., Mogavero, F., Murano, A.: Graded computation tree logic. In: Proc. of LICS 2009 (2009)
5. Brihaye, T., Da Costa Lopes, A., Laroussinie, F., Markey, N.: Atl with strategy contexts and bounded memory. In: Artemov, S., Nerode, A. (eds.) LFCS 2009. LNCS, vol. 5407, pp. 92–106. Springer, Heidelberg (2008)
6. Ferrante, A., Memoli, M., Napoli, M., Parente, M., Sorrentino, F.: A NuSMV extension for graded-CTL model checking. In: Proc. of CAV 2010 (2010)
7. Ferrante, A., Murano, A., Parente, M.: Enriched μ-calculi module checking. Logical Methods in Computer Science 4(3) (2008)
8. Ferrante, A., Napoli, M., Parente, M.: CTL model-checking with graded quantifiers. In: Cha, S(S.), Choi, J.-Y., Kim, M., Lee, I., Viswanathan, M. (eds.) ATVA 2008. LNCS, vol. 5311, pp. 18–32. Springer, Heidelberg (2008)
9. Ferrante, A., Napoli, M., Parente, M.: Graded-CTL: satisfiability and symbolic model-checking. In: Breitman, K., Cavalcanti, A. (eds.) ICFEM 2009. LNCS, vol. 5885, pp. 306–325. Springer, Heidelberg (2009)
10. Ferrante, A., Napoli, M., Parente, M.: Model-checking for graded CTL. Fundamenta Informaticae 96(3), 323–339 (2009)
11. Fine, K.: In so many possible worlds. Notre Dame Journal of Formal Logic 13(4), 516–520 (1972)
12. Gradel, E., Otto, M., Rosen, E.: Two-variable logic with counting is decidable. In: Proc. of LICS 1997 (1997)
13. Hollunder, B., Baader, F.: Qualifying number restrictions in concept languages. In: 2nd International Conference on Principles of Knowledge Representation and Reasoning, pp. 335–346 (1991)
14. Immerman, N.: Number of quantifiers is better than number of tape cells. J. Comput. Syst. Sci. 22(3), 384–406 (1981)

15. Kupferman, O., Sattler, U., Vardi, M.Y.: The complexity of the graded μ-calculus. In: Voronkov, A. (ed.) CADE 2002. LNCS (LNAI), vol. 2392, p. 423. Springer, Heidelberg (2002)
16. Lawler, E.L.: A procedure for computing the k best solutions to discrete optimization problems and its application to the shortest path problem. Management Science 18(7), 401–405 (1972)
17. Lomuscio, A., Raimondi, F.: Mcmas: A model checker for multi-agent systems. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 450–454. Springer, Heidelberg (2006)
18. Martin, D.A.: Borel determinacy. Annals of Mathematics 102(2), 363–371 (1975)
19. 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)
20. van der Hoek, W., Wooldridge, M.: Cooperation, knowledge, and time: Alternating-time temporal epistemic logic and its applications. Studia Logica 75(1), 125–157 (2003)
21. Yen, J.Y.: Finding the k shortest loopless paths in a network. Management Science 17(11), 712–716 (1971)

Non-oblivious Strategy Improvement

John Fearnley

Department of Computer Science, University of Warwick, UK
john@dcs.warwick.ac.uk

Abstract. We study strategy improvement algorithms for mean-payoff and parity games. We describe a structural property of these games, and we show that these structures can affect the behaviour of strategy improvement. We show how awareness of these structures can be used to accelerate strategy improvement algorithms. We call our algorithms non-oblivious because they remember properties of the game that they have discovered in previous iterations. We show that non-oblivious strategy improvement algorithms perform well on examples that are known to be hard for oblivious strategy improvement. Hence, we argue that previous strategy improvement algorithms fail because they ignore the structural properties of the game that they are solving.

1 Introduction

In this paper we study strategy improvement for two player infinite games played on finite graphs. In this setting the vertices of a graph are divided between two players. A token is placed on one of the vertices, and in each step the owner of the vertex upon which the token is placed must move the token along one of the outgoing edges of that vertex. In this fashion, the two players form an infinite path in the graph. The payoff of the game is then some property of this path, which depends on the type of game that is being played. Strategy improvement is a technique that originated from Markov decision processes [7], and has since been applied many types of games in this setting, including simple stochastic games [3], discounted-payoff games [12], mean-payoff games [2], and parity games [15,1]. In this paper we will focus on the strategy improvement algorithm of Björklund and Vorobyov [2], which is designed to solve mean-payoff games, but can also be applied to parity games.

Algorithms that solve parity and mean-payoff games have received much interest. One reason for this is that the model checking problem for the modal μ-calculus is polynomial time equivalent to the problem of solving a parity game [4,14], and there is a polynomial time reduction from parity games to mean-payoff games [12]. Therefore, faster algorithms for these games lead to faster model checkers for the μ-calculus. Secondly, both of these games lie in NP $\cap$ co-NP, which implies that neither of the two problems are likely to be complete for either class. Despite this, no polynomial time algorithms have been found.

The approach of strategy improvement can be described as follows. The algorithm begins by choosing one of the players to be the strategy improver, and then

E.M. Clarke and A. Voronkov (Eds.): LPAR-16, LNAI 6355, pp. 212–230, 2010.

picks an arbitrary strategy for that player. A strategy for a player consists of a function that picks one edge for each of that player's vertices. Strategy improvement then computes a set of profitable edges for that strategy. If the strategy is switched so that it chooses some subset of the profitable edges, rather than the edges that are currently chosen, then strategy improvement guarantees that the resulting strategy is better in some well-defined measure. So, the algorithm picks some subset of the profitable edges to create a new, improved, strategy to be considered in the next iteration. This process is repeated until a strategy is found that has no profitable edges, and this strategy is guaranteed optimal for the strategy improver. Since any subset of the profitable edges could be used to create an improved strategy in each iteration, some method is needed to determine which subset to choose in each iteration. We call this method a switching policy, and the choice of switching policy can have a dramatic effect on the running time of the algorithm.

A significant amount of research has been dedicated to finding good switching policies. In terms of complexity bounds, the current best switching policies are randomized, and run in an expected $O(2^{\sqrt{n \log n}})$ number of iterations [2]. Another interesting switching policy is the optimal switching policy given by Schewe [13]. An optimal switching policy always picks the subset of profitable edges that yields the best possible successor strategy, according to the measure that strategy improvement uses to compare strategies. It is not difficult to show that such a subset of profitable edges must exist, but computing an optimal subset of profitable edges seemed to be difficult, since there can be exponentially many subsets of profitable edges to check. Nevertheless, Schewe's result is a polynomial time algorithm that computes an optimal subset of edges. Therefore, optimal switching policies can now be realistically implemented. It is important to note that the word "optimal" applies only to the subset of profitable edges that is chosen to be switched in each iteration. It is not the case that a strategy improvement algorithm equipped with an optimal switching policy will have an optimal running time.

Perhaps the most widely studied switching policy is the all-switches policy, which simply selects the entire set of profitable edges in every iteration. Although the best upper bound for this policy is $O(2^n/n)$ iterations [11], it has been found to work extremely well in practice. Indeed, for a period of ten years there were no known examples upon which the all switches policy took significantly more than a linear number of iterations. It was for this reason that the all-switches policy was widely held to be a contender for a proof of polynomial time termination.

However, Friedmann has recently found a family of examples that force a strategy improvement algorithm equipped with the all-switches policy to take an exponential number of steps [5]. Using the standard reductions [12,16], these examples can be generalised to provide exponential lower bounds for all-switches on mean-payoff and discounted-payoff games. Even more surprisingly, Friedmann's example can be generalised to provide an exponential lower bound for strategy improvement algorithms equipped with an optimal switching policy [6]. This recent revelation appears to imply that there is no longer any hope for strategy

improvement, since an exponential number of iterations can be forced even if the best possible improvement is made in every step.

Our contributions. Despite ten years of research into strategy improvement algorithms, and the recent advances in the complexity of some widely studied switching policies, the underlying combinatorial structure of mean-payoff and parity games remains somewhat mysterious. There is no previous work which links the structural properties of a parity or mean-payoff game with the behaviour of strategy improvement on those games. In this paper, we introduce a structural property of these games that we call a snare. We show how the existence of a snare in a parity or mean-payoff game places a restriction on the form that a winning strategy can take for these games.

We argue that snares play a fundamental role in the behaviour of strategy improvement algorithms. We show that there is a certain type of profitable edge, which we call a back edge, that is the mechanism that strategy improvement uses to deal with snares. We show how each profitable back edge encountered by strategy improvement corresponds to some snare that exists in the game. Hence, we argue that the concept of a snare is a new tool that can be used in the analysis of strategy improvement algorithms.

We then go on to show that, in addition to being an analytical tool, awareness of snares can be used to accelerate the process of strategy improvement. We propose that strategy improvement algorithms should remember the snares that they have seen in previous iterations, and we give a procedure that uses a previously recorded snare to improve a strategy. Strategy improvement algorithms can choose to apply this procedure instead of switching a subset of profitable edges. We give one reasonable example of a strategy improvement algorithm that uses these techniques. We call our algorithms non-oblivious strategy improvement algorithms because they remember information about their previous iterations, whereas previous techniques make their decisions based only on the information available in the current iteration.

In order to demonstrate how non-oblivious techniques can be more powerful than traditional strategy improvement, we study Friedmann's family of examples that cause the all-switches and the optimal switching policies to take exponential time. We show that in certain situations non-oblivious strategy improvement makes better progress than even the optimal oblivious switching policy. We go on to show that this behaviour allows our non-oblivious strategy improvement algorithms to terminate in polynomial time on Friedmann's examples. This fact implies that it is ignorance of snares that is a key failing of oblivious strategy improvement.

2 Preliminaries

A mean-payoff game is defined by a tuple $(V, V_{\text{Max}}, V_{\text{Min}}, E, w)$ where V is a set of vertices and E is a set of edges, which together form a finite graph. Every

vertex must have at least one outgoing edge. The sets V_{Max} and V_{Min} partition V into vertices belonging to player Max and vertices belonging to player Min, respectively. The function $w : V \to \mathbb{Z}$ assigns an integer weight to every vertex.

The game begins by placing a token on a starting vertex v_0. In each step, the player that owns the vertex upon which the token is placed must choose one outgoing edge of that vertex and move the token along it. In this fashion, the two players form an infinite path $\pi = \langle v_0, v_1, v_2, \ldots \rangle$, where (v_i, v_{i+1}) is in E for every i in $\mathbb{N}$. The *payoff* of an infinite path is defined to be $\mathcal{M}(\pi) = \liminf_{n\to\infty}(1/n)\sum_{i=0}^{n} w(v_i)$. The objective of Max is to maximize the value of $\mathcal{M}(\pi)$, and the objective of Min is to minimize it.

A *positional strategy* for Max is a function that chooses one outgoing edge for every vertex belonging to Max. A strategy is denoted by $\sigma : V_{\text{Max}} \to V$, with the condition that $(v, \sigma(v))$ is in E, for every Max vertex v. Positional strategies for player Min are defined analogously. The sets of positional strategies for Max and Min are denoted by Π_{Max} and Π_{Min}, respectively. Given two positional strategies, σ and τ for Max and Min respectively, and a starting vertex v_0, there is a unique path $\langle v_0, v_1, v_2 \ldots \rangle$, where $v_{i+1} = \sigma(v_i)$ if v_i is owned by Max and $v_{i+1} = \tau(v_i)$ if v_i is owned by Min. This path is known as the *play* induced by the two strategies σ and τ, and will be denoted by $\text{Play}(v_0, \sigma, \tau)$.

For all v in V we define:

$$\text{Value}_*(v) = \max_{\sigma\in\Pi_{\text{Max}}} \min_{\tau\in\Pi_{\text{Min}}} \mathcal{M}(\text{Play}(v, \sigma, \tau))$$
$$\text{Value}^*(v) = \min_{\tau\in\Pi_{\text{Min}}} \max_{\sigma\in\Pi_{\text{Max}}} \mathcal{M}(\text{Play}(v, \sigma, \tau))$$

These are known as the lower and upper values, respectively. For mean-payoff games we have that the two quantities are equal, a property called determinacy.

Theorem 1 ([10]). *For every starting vertex v in every mean-payoff game we have* $\text{Value}_*(v) = \text{Value}^*(v)$.

For this reason, we define $\text{Value}(v)$ to be the value of the game starting at the vertex v, which is equal to both $\text{Value}_*(v)$ and $\text{Value}^*(v)$. The computational task associated with mean-payoff games is to find $\text{Value}(v)$ for every vertex v.

Computing the 0-mean partition is a decision version of this problem. This requires us to decide whether $\text{Value}(v) > 0$, for every vertex v. Björklund and Vorobyov have shown that only a polynomial number of calls to an algorithm for finding the 0-mean partition are needed to find the value for every vertex in a mean-payoff game [2].

A Max strategy σ is a *winning strategy* for a set of vertices W if $\mathcal{M}(v, \sigma, \tau) > 0$ for every Min strategy τ and every vertex v in W. Similarly, a Min strategy τ is a winning strategy for W if $\mathcal{M}(v, \sigma, \tau) \leq 0$ for every Max strategy σ and every vertex v in W. To solve the 0-mean partition problem we are required to partition the vertices of the graph into the sets $(W_{\text{Max}}, W_{\text{Min}})$, where Max has a winning strategy for W_{Max} and Min has a winning strategy for W_{Min}.

3 Snares

In this section we introduce a structure called that we call a "snare". The dictionary definition[1] of the word snare is "something that serves to entangle the unwary". This is a particularly apt metaphor for these structures since, as we will show, a winning strategy for a player must be careful to avoid being trapped by the snares that are present in that player's winning set.

The definitions in this section could be formalized for either player. We choose to focus on player Max because we will later choose Max to be the strategy improver. For a set of vertices W we define $G \restriction W$ to be the sub-game induced by W, which is G with every vertex not in W removed. A snare for player Max is defined to be a subgame for which player Max can guarantee a win from every vertex.

Definition 2 (Max Snare). *For a game G, a snare is defined to be a tuple (W, χ) where $W \subseteq V$ and $\chi : W \cap V_{Max} \to W$ is a partial strategy for player Max that is winning for every vertex in the subgame $G \restriction W$.*

This should be compared with the concept of a dominion that was introduced by Jurdziński, Paterson, and Zwick [8]. A dominion is also a subgame in which one of the players can guarantee a win, but with the additional constraint that the opponent is unable to leave the dominion. By contrast, the opponent may be capable of leaving a snare. We define an escape edge for Min to be an edge that Min can use to leave a Max snare.

Definition 3 (Escapes). *Let W be a set of vertices. We define the escapes from W as* $\mathrm{Esc}(W) = \{(v, u) \in E \ : \ v \in W \cap V_{Min} \text{ and } u \notin W\}$.

It is in Min's interests to use at least one escape edge from a snare, since if Min stays in a Max snare forever, then Max can use the strategy χ to ensure a positive payoff. In fact, we can prove that if τ is a winning strategy for Min for some subset of vertices then τ must use at least one escape from every Max snare that exists in that subset of vertices.

Theorem 4. *Suppose that τ is a winning strategy for Min on a set of vertices S. If (W, χ) is a Max snare where $W \subset S$, then there is some edge (v, u) in* $\mathrm{Esc}(W)$ *such that $\tau(v) = u$.*

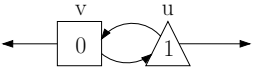


Fig. 1. A simple snare

Figure 1 shows an example of a subgame upon which a snare can be defined. In all of our diagrams, boxes are used to represent Max vertices and triangles are

[1] American Heritage Dictionary of the English Language, Fourth Edition.

used to represent Min vertices. The weight assigned to each vertex is shown on that vertex. If we take $W = \{v, u\}$ and $\chi(v) = u$ then (W, χ) will be a Max snare in every game that contains this structure as a subgame. This is because the cycle is positive, and therefore χ is a winning for Max on the subgame induced by W. There is one escape from this snare, which is the edge Min can use to break the cycle at u.

Since the example is so simple, Theorem 4 gives a particularly strong property for this snare: every winning strategy for Min must use the escape edge at u. If Min uses the edge (u, v) in some strategy, then Max could respond by using the edge (v, u) to guarantee a positive cycle, and therefore the strategy would not be winning for Min. This is a strong property because we can essentially ignore the edge (u, v) in every game into which the example is embedded. This property does not hold for snares that have more than one escape.

4 Strategy Improvement

In this section we will summarise Björklund and Vorobyov's strategy improvement algorithm for finding the 0-mean partition of a mean-payoff game [2]. Their algorithm requires that the game is modified by adding retreat edges from every Max vertex to a special sink vertex.

Definition 5 (Modified Game). *A game $(V, V_{Max}, V_{Min}, E, w)$ will be modified to create $(V \cup \{s\}, V_{Max} \cup \{s\}, V_{Min}, E', w')$, where $E' = E \cup \{(v, s) : v \in V_{Max}\}$, and $w'(v) = w(v)$ for all vertices v in V, and $w'(s) = 0$.*

Strategy improvement always works with the modified game, and for the rest of the paper we will assume that the game has been modified.

Given two strategies, one for each player, the play induced by the two strategies is either a finite path that ends at the sink or a finite initial path followed by an infinitely repeated cycle. This is used to define the valuation of a vertex.

Definition 6 (Valuation). *Let σ be a positional strategy for Max and τ be a positional strategy for Min. If $\mathrm{Play}(v_0, \sigma, \tau) = \langle v_0, v_1, \ldots v_k, \langle c_0, c_1, \ldots c_l \rangle^\omega \rangle$, for some vertex v_0, then we define $\mathrm{Val}^{\sigma,\tau}(v_0) = -\infty$ if $\sum_{i=0}^{l} w(c_i) \leq 0$ and ∞ otherwise. Alternatively, if $\mathrm{Play}(v, \sigma, \tau) = \langle v_0, v_1, \ldots v_k, s \rangle$ then we define $\mathrm{Val}^{\sigma,\tau}(v_0) = \sum_{i=0}^{k} w(v_i)$.*

Strategy improvement algorithms choose one player to be the strategy improver, which we choose to be Max. For a Max strategy σ, we define $\mathrm{br}(\sigma)$ to be the *best response* to σ, which is a Min strategy with the property $\mathrm{Val}^{\sigma,\mathrm{br}(\sigma)}(v) \leq \mathrm{Val}^{\sigma,\tau}(v)$ for every vertex v and every Min strategy τ. Such a strategy always exists, and Björklund and Vorobyov give a method to compute it in polynomial time [2]. We will frequently want to refer to the valuation of a vertex v when the Max strategy σ is played against $\mathrm{br}(\sigma)$, so we define $\mathrm{Val}^{\sigma}(v)$ to be shorthand for $\mathrm{Val}^{\sigma,\mathrm{br}(\sigma)}(v)$. Occasionally, we will need to refer to valuations from multiple games. We use $\mathrm{Val}^{\sigma}_{G}(v)$ to give the valuation of the vertex v when σ is played

against $\text{br}(\sigma)$ in the game G. We extend all of our notations in a similar manner, by placing the game in the subscript.

For a Max strategy σ and an edge (v, u) that is not chosen by σ, we say (v, u) is *profitable* in σ if $\text{Val}^\sigma(\sigma(v)) < \text{Val}^\sigma(u)$. *Switching* an edge (v, u) in σ is denoted by $\sigma[v \mapsto u]$. This operation creates a new strategy where, for a vertex $w \in V_{\text{Max}}$ we have $\sigma[v \mapsto u](w) = u$ if $w = v$, and $\sigma(w)$ otherwise. Let F be a set of edges that contains at most one outgoing edge from each vertex. We define $\sigma[F]$ to be σ with every edge in F switched. The concept of profitability is important because switching profitable edges creates an improved strategy.

Theorem 7 ([2]). *Let σ be a strategy and P be the set of edges that are profitable in σ. Let $F \subseteq P$ be a subset of the profitable edges that contains at most one outgoing edge from each vertex. For every vertex v we have $\text{Val}^\sigma(v) \leq \text{Val}^{\sigma[W]}(v)$, and there is a vertex for which the inequality is strict.*

The second property that can be shown is that a strategy with no profitable edges is optimal. An optimal strategy is a Max strategy σ such that $\text{Val}^\sigma(v) \geq \text{Val}^\chi(v)$ for every Max strategy χ and every vertex v. The 0-mean partition can be derived from an optimal strategy σ: the set W_{Max} contains every vertex v with $\text{Val}^\sigma(v) = \infty$, and W_{Min} contains every vertex v with $\text{Val}^\sigma(v) < \infty$.

Theorem 8 ([2]). *A strategy with no profitable edges is optimal.*

Strategy improvement begins by choosing a strategy σ_0 with the property that $\text{Val}^{\sigma_0}(v) > -\infty$ for every vertex v. One way to achieve this is to set $\sigma_0(v) = s$ for every vertex v in V_{Max}. This guarantees the property unless there is some negative cycle that Min can enforce without passing through a Max vertex. Clearly, for a vertex v on one of these cycles, Max has no strategy σ with $\text{Val}^\sigma(v) > -\infty$. These vertices can therefore be removed in a preprocessing step and placed in W_{Min}.

For every strategy σ_i a new strategy $\sigma_{i+1} = \sigma_i[F]$ will be computed, where F is a subset of the profitable edges in σ_i, which contains at most one outgoing edge from each vertex. Theorem 7 implies that $\text{Val}^{\sigma_{i+1}}(v) \geq \text{Val}^{\sigma_i}(v)$ for every vertex v, and that there is a vertex for which the inequality is strict. This implies that a strategy cannot be visited twice by strategy improvement. The fact that there is a finite number of positional strategies for Max implies that strategy improvement must eventually reach a strategy σ_k in which no edges are profitable. Theorem 8 implies that σ_k is the optimal strategy, and strategy improvement terminates.

Strategy improvement requires a rule that determines which profitable edges are switched in each iteration. We will call this a *switching policy*. Oblivious switching policies are defined as $\alpha : 2^E \to 2^E$, where for every set $P \subseteq E$, we have that $\alpha(P)$ contains at most one outgoing edge for each vertex.

Some of the most widely studied switching policies are all-switches policies. These policies always switch every vertex that has a profitable edge, and when a vertex has more than one profitable edge an additional rule must be given to determine which edge to choose. Traditionally this choice is made by choosing the successor with the highest valuation. We must also be careful to break ties

when there are two or more successors with the highest valuation. Therefore, for the purposes of defining this switching policy we will assume that each vertex v is given a unique index in the range $\{1, 2, \ldots, |V|\}$, which we will denote as $\text{Index}(v)$.

$$\text{All}(F) = \{(v,u) \ : \ \text{There is no edge } (v,w) \in F \text{ with } \text{Val}^\sigma(u) < \text{Val}^\sigma(w) \\ \text{or with } \text{Val}^\sigma(u) = \text{Val}^\sigma(w) \text{ and } \text{Index}(u) < \text{Index}(w)\}.$$

In the introduction we described optimal switching policies, which we can now formally define. A switching policy is optimal if it selects a subset of profitable edges F that satisfies $\text{Val}^{\sigma[H]}(v) \leq \text{Val}^{\sigma[F]}(v)$ for every subset of profitable edges H and every vertex v. Schewe has given a method to compute such a set in polynomial time [13]. We will denote an optimal switching policy as Optimal.

5 Strategy Trees

The purpose of this section is to show how a strategy and its best response can be viewed as a tree, and to classify profitable edges by their position in this tree. We will classify edges as either cross edges or back edges. We will later show how profitable back edges are closely related to snares.

It is technically convenient for us to make the assumption that every vertex has a finite valuation under every strategy. The choice of starting strategy ensures that for every strategy σ considered by strategy improvement, we have $\text{Val}^\sigma(v) > -\infty$ for every vertex v. Obviously, there may be strategies under which some vertices have a valuation of ∞. The first part of this section is dedicated to rephrasing the problem so that our assumption can be made.

We define the *positive cycle* problem to be the problem of finding a strategy σ with $\text{Val}^\sigma(v) = \infty$ for some vertex v, or to prove that there is no strategy with this property. The latter can be done by finding an optimal strategy σ with $\text{Val}^\sigma(v) < \infty$ for every vertex v. We can prove that a strategy improvement algorithm for the positive cycle problem can be adapted to find the 0-mean partition.

Proposition 9. *Let α be a strategy improvement algorithm that solves the positive cycle problem in $O(\kappa)$ time. There is a strategy improvement algorithm which finds the 0-mean partition in $O(|V| \cdot \kappa)$ time.*

We consider switching policies that solve the positive cycle problem, and so we can assume that every vertex has a finite valuation under every strategy that our algorithms consider. Our switching policies will terminate when a vertex with infinite valuation is found. With this assumption we can define the strategy tree.

Definition 10 (Strategy Tree). *Given a Max strategy σ and a Min strategy τ we define the tree $T^{\sigma,\tau} = (V, E')$ where $E' = \{(v,u) \ : \ \sigma(v) = u \text{ or } \tau(v) = u\}$.*

In other words, $T^{\sigma,\tau}$ is a tree rooted at the sink whose edges are those chosen by σ and τ. We define T^σ to be shorthand for $T^{\sigma,\text{br}(\sigma)}$, and $\text{Subtree}^\sigma(v) : V \to 2^V$

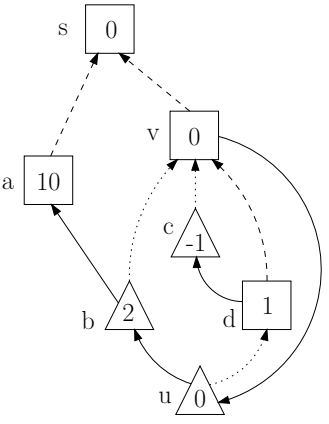


Fig. 2. A strategy tree

to be the function that gives the vertices in the subtree rooted at the vertex v in T^σ.

We can now define our classification for profitable edges. Let (v, u) be a profitable edge in the strategy σ. We call this a profitable *back edge* if u is in $\text{Subtree}^\sigma(v)$, otherwise we call it a profitable *cross edge*.

Figure 2 gives an example of a strategy tree. In all of our diagrams, the dashed lines give a strategy σ for Max, and the dotted lines show $\text{br}(\sigma)$. The strategy tree contains every vertex, and every edge that is either dashed or dotted. The subtree of v is the set $\{v, b, c, d, u\}$. The edge (v, u) is profitable because $\text{Val}^\sigma(v) = 0$ and $\text{Val}^\sigma(u) = 1$. Since u is contained in the subtree of v, the edge (v, u) is a profitable back edge.

6 Profitable Back Edges

In this section we will expose the intimate connection between profitable back edges and snares. We will show how every profitable back edge corresponds to some snare that exists in the game. We will also define the concept of snare consistency, and we will show how this concept is linked with the conditions implied by Theorem 4.

Our first task is to show how each profitable back edge corresponds to some Max snare in the game. Recall that a Max snare consists of a set of vertices, and a strategy for Max that is winning for the subgame induced by those vertices. We will begin by defining the set of vertices for the snare that corresponds to a profitable back edge. For a profitable back edge (v, u) in a strategy σ we define the critical set, which is the vertices in $\text{Subtree}^\sigma(v)$ that Min can reach when Max plays σ.

Definition 11 (Critical Set). *If (v, u) is a profitable back edge in the strategy σ, then we define the critical set as* $\text{Critical}^\sigma(v, u) = \{w \in \text{Subtree}^\sigma(v) \ :$

There is a path $\langle u, u_1, \ldots u_k = w\rangle$ *where for all* i *with* $1 \leq i \leq k$ *we have* $u_i \in \text{Subtree}^\sigma(v)$ *and if* $u_i \in V_{Max}$ *then* $u_{i+1} = \sigma(u_i)\}$.

In the example given in Figure 2, the critical set for the edge (v, u) is $\{v, b, d, u\}$. The vertex b is in the critical set because it is in the subtree of v, and Min can reach it from u when Max plays σ. In contrast, the vertex c is not in the critical set because $\sigma(d) = v$, and therefore Min cannot reach c from u when Max plays σ. The vertex a is not in the critical set because it is not in the subtree of v.

Note that in the example, $\sigma[v \mapsto u]$ is a winning strategy for the subgame induced by critical set. The definition of the critical set is intended to capture the largest connected subset of vertices contained in the subtree of v for which $\sigma[v \mapsto u]$ is guaranteed to be a winning strategy.

Proposition 12. *Let* (v, u) *be a profitable back edge in the strategy* σ *and let* C *be* $\text{Critical}^\sigma(v, u)$. *The strategy* $\sigma[v \mapsto u]$ *is winning for every vertex in* $G \restriction C$.

We can now formally define the snare that is associated with each profitable back edge that is encountered by strategy improvement. For a profitable back edge (v, u) in a strategy σ we define $\text{Snare}^\sigma(v, u) = (\text{Critical}^\sigma(v, u), \chi)$ where $\chi(v) = \sigma[v \mapsto u](v)$ if $v \in \text{Critical}^\sigma(v, u)$, and undefined at other vertices. Proposition 12 confirms that this meets the definition of a snare.

We will now argue that the conditions given by Theorem 4 must be observed in order for strategy improvement to terminate. We begin by defining a concept that we call snare consistency. We say that a Max strategy is consistent with a snare if Min's best response chooses an escape from that snare.

Definition 13 (Snare Consistency). *A strategy* σ *is said to be consistent with the snare* (W, χ) *if* $\text{br}(\sigma)$ *uses some edge in* $\text{Esc}(W)$.

In the example given in Figure 2 we can see that σ is not consistent with $\text{Snare}^\sigma(v, u)$. This is because $\text{br}(\sigma)$ does not choose the edge (b, a). However, once the edge (v, u) is switched we can prove that $\text{br}(\sigma[v \mapsto u])$ must use the edge (b, a). This is because Min has no other way of connecting every vertex in $\text{Subtree}^\sigma(v)$ to the sink, and if some vertex is not connected to the sink then its valuation will rise to ∞.

Proposition 14. *Let* (v, u) *be a profitable back edge in the strategy* σ. *There is some edge* (x, y) *in* $\text{Esc}(\text{Critical}^\sigma(v, u))$ *such that* $\text{br}(\sigma[v \mapsto u])(x) = y$.

We can show that strategy improvement cannot terminate unless the current strategy is consistent with every snare that exists in the game. This is because every strategy that is not consistent with some snare must contain a profitable edge.

Proposition 15. *Let* σ *be a strategy that is not consistent with a snare* (W, χ). *There is a profitable edge* (v, u) *in* σ *such that* $\chi(v) = u$.

These two propositions give us a new tool to study the process of strategy improvement. Instead of viewing strategy improvement as a process that tries to

increase valuations, we can view it as a process that tries to force consistency with Max snares. Proposition 15 implies that this process can only terminate when the current strategy is consistent with every Max snare in the game. Therefore, the behaviour of strategy improvement on an example is strongly related with the snares that exist for the strategy improver in that example.

7 Using Snares to Guide Strategy Improvement

In the previous sections, we have shown the strong link between snares and strategy improvement. In this section we will show how this insight can be used to guide strategy improvement. We will give a procedure that takes a strategy that is inconsistent with some snare, and returns an improved strategy that is consistent with that snare. Since the procedure is guaranteed to produce an improved strategy, it can be used during strategy improvement as an alternative to switching a profitable edge. We call algorithms that make use of this procedure non-oblivious strategy improvement algorithms, and we give a reasonable example of such an algorithm.

To define our procedure we will use Proposition 15. Recall that this proposition implies that if a strategy σ is inconsistent with a snare (W, χ), then there is some profitable edge (v, u) in σ such that $\chi(v) = u$. Our procedure will actually be a strategy improvement switching policy. This policy will always choose to switch an edge that is chosen by χ but not by the current strategy. As long as the current strategy remains inconsistent with (W, χ) such an edge is guaranteed to exist, and the policy terminates once the current strategy is consistent with the snare. This procedure is shown as Algorithm 1.

Algorithm 1. FixSnare$(\sigma, (W, \chi))$

while σ is inconsistent with (W, χ) **do**
  $(v, w) :=$ Some edge where $\chi(v) = w$ and (v, w) is profitable in σ.
  $\sigma := \sigma[v \mapsto u]$
end while
return σ

In each iteration the switching policy switches one vertex v to an edge (v, u) with the property that $\chi(v) = u$, and it never switches a vertex at which the current strategy agrees with χ. It is therefore not difficult to see that if the algorithm has not terminated after $|W|$ iterations then the current strategy will agree with χ on every vertex in W. We can prove that such a strategy must be consistent with (W, χ), and therefore the switching policy must terminate after at most $|W|$ iterations.

Proposition 16. *Let σ be a strategy that is not consistent with a snare (W, χ). Algorithm 1 will arrive at a strategy σ' which is consistent with (W, χ) after at most $|W|$ iterations.*

Since FixSnare is implemented as a strategy improvement switching policy that switches only profitable edges, the strategy that is produced must be an improved strategy. Therefore, at any point during the execution of strategy improvement we can choose not to switch a subset of profitable edges and run FixSnare instead. Note that the strategy produced by FixSnare may not be reachable from the current strategy by switching a subset of profitable edges. This is because FixSnare switches a sequence of profitable edges, some of which may not have been profitable in the original strategy.

We propose a new class of strategy improvement algorithms that are aware of snares. These algorithms will record a snare for every profitable back edge that they encounter during their execution. In each iteration these algorithms can either switch a subset of profitable edges or run the procedure FixSnare on some recorded snare that the current strategy is inconsistent with. We call these algorithms non-oblivious strategy improvement algorithms, and the general schema that these algorithms follow is shown in Algorithm 2.

Algorithm 2. NonOblivious(σ)

$S := \emptyset$
while σ has a profitable edge **do**
  $S := S \cup \{\text{Snare}^{\sigma}(v,u) \ : \ (v,u) \text{ is a profitable back edge in } \sigma\}$
  $\sigma := \text{Policy}(\sigma, S)$
end while
return σ

Recall that oblivious strategy improvement algorithms required a switching policy to specify which profitable edges should be switched in each iteration. Clearly, non-oblivious strategy improvement algorithms require a similar method to decide whether to apply the procedure FixSnare or to pick some subset of profitable edges to switch. Moreover, they must decide which snare should be used when the procedure FixSnare is applied. We do not claim to have the definitive non-oblivious switching policy, but in the rest of this section we will present one reasonable method of constructing a non-oblivious version of an oblivious switching policy. We will later show that our non-oblivious strategy improvement algorithms behave well on the examples that are known to cause exponential time behaviour for oblivious strategy improvement.

We intend to take an oblivious switching policy α as the base of our non-oblivious switching policy. This means that when we do not choose to use the procedure FixSnare, we will switch the subset of profitable edges that would be chosen by α. Our goal is to only use FixSnare when doing so is guaranteed to yield a larger increase in valuation than applying α. Clearly, in order to achieve this we must know how much the valuations increase when α is applied and how much the valuations increase when FixSnare is applied.

Determining the increase in valuation that is produced by applying an oblivious switching policy is easy. Since every iteration of oblivious strategy improvement takes polynomial time, We can simply switch the edges and measure the

difference between the current strategy and the one that would be produced. Let σ be a strategy and let P be the set of edges that are profitable in σ. For an oblivious switching policy α the increase of applying α is defined to be:

$$\text{Increase}(\alpha, \sigma) = \sum_{v \in V} (\text{Val}^{\sigma[\alpha(P)]}(v) - \text{Val}^{\sigma}(v))$$

We now give a lower bound on the increase in valuation that an application of FixSnare produces. Let (W, χ) be a snare and suppose that the current strategy σ is inconsistent with this snare. Our lower bound is based on the fact that FixSnare will produce a strategy that is consistent with the snare. This means that Min's best response is not currently choosing an escape from the snare, but it will be forced to do so after FixSnare has been applied. It is easy to see that forcing the best response to use a different edge will cause an increase in valuation, since otherwise the best response would already be using that edge. Therefore, we can use the increase in valuation that will be obtained when Min is forced to use and escape. We define:

$$\text{SnareIncrease}^{\sigma}(W, \chi) = \min\{(\text{Val}^{\sigma}(y) + w(x)) - \text{Val}^{\sigma}(x) \ : \ (x, y) \in \text{Esc}(W)\}$$

This expression gives the smallest possible increase in valuation that can happen when Min is forced to use an edge in $\text{Esc}(W)$. We can prove that applying FixSnare will cause an increase in valuation of at least this amount.

Proposition 17. *Let σ be a strategy that is not consistent with a snare (W, χ), and let σ' be the result of* $\text{FixSnare}(\sigma, (W, \chi))$. *We have:*

$$\sum_{v \in V} (\text{Val}^{\sigma'}(v) - \text{Val}^{\sigma}(v)) \geq \text{SnareIncrease}^{\sigma}(W, \chi)$$

We now have the tools necessary to construct our proposed augmentation scheme, which is shown as Algorithm 3. The idea is to compare the increase obtained by applying α and the increase obtained by applying FixSnare with the best snare that has been previously recorded, and then to only apply FixSnare when it is guaranteed to yield a larger increase in valuation.

Algorithm 3. $(\text{Augment}(\alpha))(\sigma, S)$

$(W, \chi) := \text{argmax}_{(X,\mu) \in S} \text{SnareIncrease}^{\sigma}(X, \mu)$
if $\text{Increase}(\alpha, \sigma) > \text{SnareIncrease}^{\sigma}(W, \chi)$ **then**
  $P := \{(v, u) \ : \ (v, u) \text{ is profitable in } \sigma\}$
  $\sigma := \sigma[\alpha(P)]$
else
  $\sigma := \text{FixSnare}(\sigma, (W, \chi))$
end if
return σ

8 Comparison with Oblivious Strategy Improvement

In this section we will demonstrate how non-oblivious strategy improvement can behave well in situations where oblivious strategy improvement has exponential time behaviour. Unfortunately, there is only one source of examples with such properties in the literature, and that is the family of examples given by Friedmann. In fact, Friedmann gives two slightly different families of hard examples. The first type is the family that that forces exponential behaviour for the all-switches policy [5], and the second type is the family that forces exponential behaviour for both all-switches and optimal switching policies [6]. Although our algorithm performs well on both families, we will focus on the example that was designed for optimal switching policies because it is the most interesting of the two.

This section is split into two parts. In the first half of this section we will study a component part of Friedmann's example upon which the procedure FixSnare can out perform an optimal switching policy. This implies that there are situations in which our augmentation scheme will choose to use FixSnare. In the second half, we will show how the good performance on the component part is the key property that allows our non-oblivious strategy improvement algorithms to terminate quickly on Friedmann's examples.

8.1 Optimal Switching Policies

We have claimed that the procedure FixSnare can cause a greater increase in valuation than switching any subset of profitable edges. We will now give an example upon which this property holds. The example that we will consider is shown in Figure 3, and it is one of the component parts of Friedmann's family of examples that force optimal policies to take an exponential number of steps [6].

The diagram shows a strategy for Max as a set of dashed edges. It also shows Min's best response to this strategy as a dotted edge. Even though this example

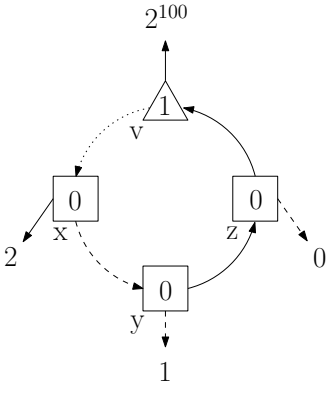


Fig. 3. A component of Friedmann's exponential time example

could be embedded in an arbitrary game, we can reason about the behaviour of strategy improvement by specifying, for each edge that leaves the example, the valuation of the successor vertex that the edge leads to. These valuations are shown as numbers at the end of each edge that leaves the example.

In order to understand how strategy improvement behaves we must determine the set of edges that are profitable for our strategy. There are two edges that are profitable: the edge (z, v) is profitable because the valuation of v is 2 which is greater than 0, and the edge at x that leaves the example is profitable because leaving the example gives a valuation of 2 and the valuation of y is 1. The edge (y, z) is not profitable because the valuation of z is 0, which is smaller than the valuation of 1 obtained by leaving the example at y.

For the purposes of demonstration, we will assume that no other edge is profitable in the game into which the example is embedded. Furthermore, we will assume that no matter what profitable edges are chosen to be switched, the valuation of every vertex not contained in the example will remain constant. Therefore, the all-switches policy will switch the edges (z, v) and the edge leading away from the example at the vertex x. It can easily be verified that this is also the optimal subset of profitable edges, and so the all-switches and the optimal policies make the same decisions for this strategy. After switching the edges chosen by the two policies, the valuation of x will rise to 2, the valuation of z will rise to 3, and the valuation of y remain at 1.

By contrast, we will now argue that non-oblivious strategy improvement would raise the valuations of x, y, and z to $2^{100} + 1$. Firstly, it is critical to note that the example is a snare. If we set $W = \{v, x, y, z\}$ and choose χ to be the partial strategy for Max that chooses the edges (x, y), (y, z), and (z, v), then (W, χ) will be a snare in every game into which the example is embedded. This is because there is only one cycle in the subgame induced by W when Max plays χ, and this cycle has positive weight.

Now, if the non-oblivious strategy improvement algorithm was aware of the snare (W, χ) then the lower bound given by Proposition 17 would be 2^{100}. This is because closing the cycle forces Min's best response to use escape edge to avoid losing the game. Since 2^{100} is much larger than the increase obtained by the optimal switching policy, the policies Augment(All) and Augment(Optimal) will choose to run FixSnare on the snare (W, χ). Once consequence of this is that the policy Optimal is no longer optimal in the non-oblivious setting.

8.2 Friedmann's Exponential Time Examples

The example that we gave in the previous subsection may appear to be trivial. After all, if the valuations outside the example remain constant then both the all-switches and optimal switching policies will close the cycle in two iterations. A problem arises, however, when the valuations can change. Note that when we applied the oblivious policies to the example, no progress was made towards closing the cycle. We started with a strategy that chose to close the cycle at only one vertex, and we produced a strategy that chose to close the cycle at only one vertex. When the assumption that valuations outside the example are constant

is removed, it becomes possible for a well designed game to delay the closing of the cycle for an arbitrarily large number of iterations simply by repeating the pattern of valuations that is shown in Figure 3.

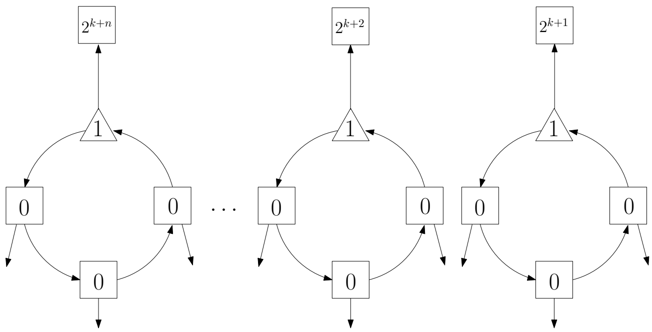


Fig. 4. The bits of a binary counter

Friedmann's family of examples exploits this property to build a binary counter, which uses the subgame shown in Figure 3 to represent the bits. The general idea of this approach is shown in Figure 4. Friedmann's example uses n instances of the cycle, indexed 1 through n. These bits are interconnected in a way that enforces two properties on both the all-switches and the optimal switching policies. Firstly, the ability to prevent a cycle from closing that we have described is used to ensure that the cycle with index i can only be closed after every cycle with index smaller than i has been closed. Secondly, when the cycle with index i is closed, every cycle with index smaller than i is forced to open. Finally, every cycle is closed in the optimal strategy for the example. Now, if the initial strategy is chosen so that every cycle is open, then these three properties are sufficient to force both switching policies to take at least 2^n steps before terminating.

The example works by forcing the oblivious switching policy to make the same mistakes repeatedly. To see this, consider the cycle with index $n-1$. When the cycle with index n is closed for the first time, this cycle is forced open. The oblivious optimal switching policy will then not close it again for at least another 2^{n-1} steps. By contrast, the policies Augment(All) and Augment(Optimal) would close the cycle again after a single iteration. This breaks the exponential time behaviour, and it turns out that both of our policies terminate in polynomial time on Friedmann's examples.

Of course, for Friedmann's examples we can tell by inspection that Max always wants to keep the cycle closed. It is not difficult, however, to imagine an example which replaces the four vertex cycle with a complicated subgame, for which Max had a winning strategy and Min's only escape is to play to the vertex with a large weight. This would still be a snare, but the fact that it is a snare would only become apparent during the execution of strategy improvement. Nevertheless, as long as the subgame can be solved in polynomial time

by non-oblivious strategy improvement, the whole game will also be solved in polynomial time. This holds for exactly the same reason as the polynomial behaviour on Friedmann's examples: once the snare representing the subgame has been recorded then consistency with that snare can be enforced in the future.

9 Conclusions and Further Work

This paper has uncovered and formalized a strong link between the snares that exist in a game and the behaviour of strategy improvement on that game. We have shown how this link can be used to guide the process of strategy improvement. With our augmentation procedure we gave one reasonable method of incorporating non-oblivious techniques into traditional strategy improvement, and we have demonstrated how these techniques give rise to good behaviour on the known exponential time examples.

It must be stressed that we are not claiming that simply terminating in polynomial time on Friedmann's examples is a major step forward. After all, the randomized switching policies of Björklund and Vorobyov [2] have the same property. What is important is that our strategy improvement algorithms are polynomial because they have a better understanding of the underlying structure of strategy improvement. Friedmann's examples provide an excellent cautionary tale that shows how ignorance of this underlying structure can lead to exponential time behaviour.

There are a wide variety of questions that are raised by this work. Firstly, we have the structure of snares in parity and mean-payoff games. Theorem 4 implies that all algorithms that find winning strategies for parity and mean payoff games must, at least implicitly, consider snares. We therefore propose that a thorough and complete understanding of how snares arise in a game is a necessary condition for devising a polynomial time algorithm for these games.

It is not currently clear how the snares in a game affect the difficulty of solving that game. It is not difficult, for example, to construct a game in which there an exponential number of Max snares: in a game in which every weight is positive there will be a snare for every connected subset of vertices. However, games with only positive weights have been shown to be very easy to solve [9]. Clearly, the first challenge is to give a clear formulation of how the structure of the snares in a given game affects the difficulty of solving it.

In our attempts to construct intelligent non-oblivious strategy improvement algorithms we have continually had problems with examples in which Max and Min snares overlap. By this we mean that the set of vertices that define the subgames of the snares have a non empty intersection. We therefore think that studying how complex the overlapping of snares can be in a game may lead to further insight. There are reasons to believe that these overlappings cannot be totally arbitrary, since they arise from the structure of the game graph and the weights assigned to the vertices.

We have presented a non-oblivious strategy improvement algorithm that passively records the snares that are discovered by an oblivious switching policy, and

then uses those snares when doing so is guaranteed to lead to a larger increase in valuations. While we have shown that this approach can clearly outperform traditional strategy improvement, it does not appear to immediately lead to a proof of polynomial time termination. It would be interesting to find an exponential time example for the augmented versions of the all-switches policy or of the optimal policy. This may be significantly more difficult since it is no longer possible to trick strategy improvement into making slow progress by forcing it to repeatedly close a small number of snares.

There is no inherent reason why strategy improvement algorithms should be obsessed with trying to increase valuations as much as possible in each iteration. Friedmann's exponential time example for the optimal policy demonstrates that doing so in no way guarantees that the algorithm will always make good progress. Our work uncovers an alternate objective that strategy improvement algorithms can use to measure their progress. Strategy improvement algorithms could actively try to discover the snares that exist in the game, or they could try and maintain consistency with as many snares as possible, for example. There is much scope for an intelligent snare based strategy improvement algorithm.

We have had some limited success in designing intelligent snare based strategy improvement algorithms for parity games. We have developed a non-oblivious strategy improvement algorithm which, when given a list of known snares in the game, either solves the game or finds a snare that is not in the list of known snares. This gives the rather weak result of a strategy improvement algorithm whose running time is polynomial in $|V|$ and k, where k is the number of Max snares that exist in the game. This is clearly unsatisfactory since we have already argued that k could be exponential in the number of vertices. However, this is one example of how snares can be applied to obtain new bounds for strategy improvement. As an aside, the techniques that we used to obtain this algorithm do not generalize to mean-payoff games. Finding a way to accomplish this task for mean-payoff games is an obvious starting point for designing intelligent snare based algorithms for this type of game.

Acknowledgements. I am indebted to Marcin Jurdziński for his guidance, support, and encouragement during the preparation of this paper.

References

1. Björklund, H., Sandberg, S., Vorobyov, S.: A discrete subexponential algorithm for parity games. In: Alt, H., Habib, M. (eds.) STACS 2003. LNCS, vol. 2607, pp. 663–674. Springer, Heidelberg (2003)
2. Björklund, H., Vorobyov, S.: A combinatorial strongly subexponential strategy improvement algorithm for mean payoff games. Discrete Applied Mathematics 155(2), 210–229 (2007)
3. Condon, A.: On algorithms for simple stochastic games. In: Cai, J.-Y. (ed.) Advances in Computational Complexity Theory. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 13, pp. 51–73. American Mathematical Society, Providence (1993)

4. Emerson, E.A., Jutla, C.S., Sistla, A.P.: On model-checking for fragments of μ-calculus. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 385–396. Springer, Heidelberg (1993)
5. Friedmann, O.: A super-polynomial lower bound for the parity game strategy improvement algorithm as we know it. In: Logic in Computer Science (LICS). IEEE, Los Alamitos (2009)
6. Friedmann, O.: A super-polynomial lower bound for the parity game strategy improvement algorithm as we know it (January 2009) (preprint)
7. Howard, R.: Dynamic Programming and Markov Processes. Technology Press and Wiley (1960)
8. Jurdziński, M., Paterson, M., Zwick, U.: A deterministic subexponential algorithm for solving parity games. In: Proceedings of ACM-SIAM Symposium on Discrete Algorithms, SODA 2006, pp. 117–123. ACM/SIAM (2006)
9. Khachiyan, L., Gurvich, V., Zhao, J.: Extending dijkstras algorithm to maximize the shortest path by node-wise limited arc interdiction. In: Grigoriev, D., Harrison, J., Hirsch, E.A. (eds.) CSR 2006. LNCS, vol. 3967, pp. 221–234. Springer, Heidelberg (2006)
10. Liggett, T.M., Lippman, S.A.: Stochastic games with perfect information and time average payoff. SIAM Review 11(4), 604–607 (1969)
11. Mansour, Y., Singh, S.P.: On the complexity of policy iteration. In: Laskey, K.B., Prade, H. (eds.) UAI 1999: Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence, pp. 401–408. Morgan Kaufmann, San Francisco (1999)
12. Puri, A.: Theory of Hybrid Systems and Discrete Event Systems. PhD thesis, University of California, Berkeley (1995)
13. Schewe, S.: An optimal strategy improvement algorithm for solving parity and payoff games. In: Kaminski, M., Martini, S. (eds.) CSL 2008. LNCS, vol. 5213, pp. 369–384. Springer, Heidelberg (2008)
14. Stirling, C.: Local model checking games (extended abstract). In: Lee, I., Smolka, S.A. (eds.) CONCUR 1995. LNCS, vol. 962, pp. 1–11. Springer, Heidelberg (1995)
15. Vöge, J., Jurdziński, M.: A discrete strategy improvement algorithm for solving parity games (Extended abstract). In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 202–215. Springer, Heidelberg (2000)
16. Zwick, U., Paterson, M.: The complexity of mean payoff games on graphs. Theoretical Computer Science 158(1-2), 343–359 (1996)

A Simple Class of Kripke-Style Models in Which Logic and Computation Have Equal Standing*

Michael Gabbay[1,**] and Murdoch J. Gabbay[2]

[1] http://www.kcl.ac.uk/philosophy/people/fellows/mgabbay.html
[2] www.gabbay.org.uk

Abstract. We present a sound and complete model of lambda-calculus reductions based on structures inspired by modal logic (closely related to Kripke structures). Accordingly we can construct a logic which is sound and complete for the same models, and we identify lambda-terms with certain simple sentences (predicates) in this logic, by direct compositional translation. Reduction then becomes identified with logical entailment.

Thus, the models suggest a new way to identify logic and computation. Both have elementary and concrete representations in our models; where these representations overlap, they coincide.

In a concluding speculation, we note a certain subclass of the models which seems to play a role analogous to that played by the cumulative hierarchy models in axiomatic set theory and the natural numbers in formal arithmetic — there are many models of the respective theories, but only some, characterised by a fully second order interpretation, are the 'intended' ones.

1 Introduction

We try to unify logic and computation by using a class of structures which are (very nearly) Kripke structures. It turns out that these structures allow sound and complete interpretations of both a logic (an extension of second-order propositional logic), and computation (the untyped λ-calculus). Furthermore, we are able to compositionally translate λ-terms and formulae into our logic, and when we do so, the 'computational' reduction $\twoheadrightarrow$ maps to logical entailment, and λ maps to a kind of logical quantifier.

Combining logic and computation is of course not a new idea. The two notions are clearly related and intertwined, and there are good theoretical and practical reasons to be interested in these questions.

A naive combination of logic and computation can lead to some famous contradictions. Consider untyped λ-calculus quotiented by computational equivalence, e.g. β-equivalence. Suppose also the naive addition of some basic logical

* The authors are grateful to the anonymous referees for their comments.
** Supported by British Academy grant PDF/2006/509 & IFCoLog Hilbert-Bernays Project.

E.M. Clarke and A. Voronkov (Eds.): LPAR-16, LNAI 6355, pp. 231–254, 2010.

operations, e.g. negation $\neg$. Then we can form a term encoding the Liar paradox $L = \lambda p.\neg(p{\cdot}p){\cdot}\lambda p.\neg(p{\cdot}p)$. Then $L = \neg L$ and from this a contradiction quickly follows. This is, in a nutshell, the argument in [15].[1]

We can persist with adding logic to λ-terms. This motivated the development of types, as in higher-order logic [21], where the paradox is avoided by restricting which terms can receive an interpretation. Also, with the same intuitions but different design decisions, illative combinatory logics [3,2] can admit an untyped system but restrict which λ-terms count as 'logical'.

Conversely, we can view, e.g. the negation of a λ-term as a category error, and think of computation as an object or vehicle of logical systems. So for example *rewriting logic* has a logic with an oriented equality $\twoheadrightarrow$ representing reduction; whereas *deduction modulo* [4,7] has a background 'computational engine' which may be triggered to efficiently compute equalities between terms, but the logic is modulo this and reasons up to computational equivalence. In both cases the interpretations of the logical and computational parts of the languages are separated, like sentences are separated from terms in first-order languages and semantics.

The model of logic and computation of this paper is like rewriting logic in that we explicitly represent computations (reductions) by an arrow, $\rightarrow$. However, in line with the original intuition equivocating logic and computation, $\rightarrow$ is the same arrow as is used to represent logical implication.

In our models, λ-terms are interpreted as sets on a domain and $\rightarrow$ is interpreted as 'complement union'. Entailment and reduction are both therefore represented in the models as subset inclusion (see Definition 2.13). That is, this arrow $\rightarrow$ *really is* standard implication, just as the reader is used to. We discuss in Section 5.1 how this relates to the paradox above.

The kernel of the ideas in this paper is a class of models, presented in Section 2.1; the rest of the paper can be considered to arise just by considering their structure. It turns out that it is possible to consider λ-abstraction as a kind of quantifier and to consider reduction as subset inclusion (Section 2.2). The models are sets-based: The usual logical connectives such as conjunction, negation, and quantification are interpreted in the models as operations on sets; and logical entailment is interpreted as subset inclusion. We obtain an extension of classical second-order propositional logic with quantifiers (Section 2.3). We make our logic rich enough that it captures the structure we used to interpret λ-abstraction and reduction; because reduction is interpreted as subset inclusion, it maps directly to logical entailment.

The idea of modelling λ-reduction is not new. Models of reduction where terms are interpreted as points in an ordering are discussed by Selinger in [19]; and models, based on *graph models*, where terms are interpreted as sets on a domain of functions, are given in [16]. The models in Section 2 have similarities with these. One significant difference is that our models have a *Boolean structure*, in particular every denotation has a complement and a universal quantification. So

[1] Historically, Church wanted to base maths on the notion of function as opposed to a notion of set. The λ-calculus was invented as a foundation for logic, not a foundation for computation. [15] proved that this foundation was logically inconsistent.

the denotational domains for λ-terms also display the structure of denotational domains of propositions of classical logic.

So: logic and computation are not identical in this story (we do not claim that logic *is* computation, or vice versa) — but the two notions overlap in the models, and in this overlap, they coincide.

One possible use for our models is that they might provide a relatively elementary and systematic framework for extensions of the pure λ-calculus with 'logical' constructs, and indeed the design of other logics. We do not consider that there is anything sacred about the particular languages we use in this paper. However, they do have the virtues of being simple and well-behaved. In particular we can give tight soundness and completeness results for both the logic and the λ-calculus (Theorems 2.27 and 4.13).

The structure and main results of this paper are as follows: We develop a simple model theory and the syntax it motivates in Section 2; in Section 3 we verify that we can faithfully translate a system of λ-reduction into this syntax (and thus obtain a new model theory for λ-reduction); in Section 4 we prove completeness for an axiomatisation of the model theory. We conclude with some comments on the significance of the models presented here both for current and further research.

2 The Models, Computation, Logic

2.1 Frames

Definition 2.1. If W is a set, write $\mathcal{P}(W)$ for the set of subsets of W.

A **frame** F is a 3-tuple $(W, \bullet, H)$ of

- W a set of **worlds**,
- $\bullet$ an **application function** from $W \times W$ to $\mathcal{P}(W)$, and
- $H \subseteq \mathcal{P}(W)$.

Remark 2.2. Frames are not monoids or any other form of applicative structure — an applicative structure would map $W \times W$ to W and not, as we do here, to $\mathcal{P}(W)$. One reasonable way to think of $\bullet$ is as a non-deterministic 'application' operation, although we suggest a better way in the concluding section 5.2 (where we also discuss in more detail the differences between frames and other known structures that 'look' like frames).

Subsets of W will serve as denotations of sentences (Definitions 2.7 and 2.13). We can interpret both computational and logical connectives as elementary operations on sets of worlds (e.g. we interpret logical conjunction as sets intersection).

Remark 2.3. $H \subseteq \mathcal{P}(W)$ ('H' for 'Henkin') plays a similar role to the structure of Henkin models for higher-order logic [1,12,20]. This makes our completeness results possible and is a famous issue for second- and higher-order logics. Powersets are too large; for completeness results to be possible we must cut them down — at least when we quantify. This is why in Definitions 2.7 and 2.13, the binders

restrict quantification from $\mathcal{P}(W)$ down to H. More on this in the concluding section 5.2.

The reader familiar with modal logic can think of $\bullet$ as a ternary 'accessibility relation' R such that $Rw_1w_2w_3$ if and only if $w_3 \in w_1 \bullet w_2$. We can also think of $\bullet$ is as a non-deterministic 'application' operation, but note that frames are not applicative structures — an applicative structure would map $W \times W$ to W, whereas in the case of frames, $W \times W$ maps to $\mathcal{P}(W)$. However, $\bullet$ does induce an applicative structure on $\mathcal{P}(W)$:

Definition 2.4. Suppose $F = (W, \bullet, H)$ is a frame. Suppose $S_1, S_2 \subseteq W$ and $w \in W$.

The function $\bullet$ induces functions from $W \times \mathcal{P}(W)$ and $\mathcal{P}(W) \times \mathcal{P}(W)$ to $\mathcal{P}(W)$ by:

$$w \bullet S_1 = \bigcup\{w \bullet w' \mid w' \in S_1\} \qquad S_1 \bullet S_2 = \bigcup\{w_1 \bullet w_2 \mid w_1 \in S_1,\ w_2 \in S_2\}$$

2.2 λ-Terms

Definition 2.5. Fix a countably infinite set of **variables**. p, q, r will range over distinct variables (we call this a *permutative* convention).

Define a language $\mathcal{L}_\lambda$ of **λ-terms** by:

$$\mathtt{t} ::= p \mid \lambda p.\mathtt{t} \mid \mathtt{t}{\cdot}\mathtt{t}$$

λp binds in $\lambda p.\mathtt{t}$. For example, p is bound (not free) in $\lambda p.p{\cdot}q$. We identify terms up to α-equivalence.

We write $\mathtt{t}[p::=\mathtt{s}]$ for the usual capture-avoiding substitution. For example, $(\lambda p'.q)[q::=p] = \lambda p'.p$, and $(\lambda p.q)[q::=p] = \lambda p'.p$ where p' is a fixed but arbitrary choice of fresh variable.

Definition 2.6. Suppose $F = (W, \bullet, H)$ is a frame. A **valuation** (to F) is a map from variables to sets of worlds (elements of $\mathcal{P}(W)$). v will range over valuations.

If p is a variable, $S \subseteq W$, and v is a valuation, then write $v[p::=S]$ for the valuation mapping q to $v(q)$ and mapping p to S.

Definition 2.7. Define an **denotation** of $\mathtt{t}$ inductively by:

$$[\![p]\!]^v = v(p) \qquad [\![\mathtt{t}{\cdot}\mathtt{s}]\!]^v = [\![\mathtt{t}]\!]^v \bullet [\![\mathtt{s}]\!]^v$$

$$[\![\lambda p.\mathtt{t}]\!]^v = \{w \mid w \bullet h \subseteq [\![\mathtt{t}]\!]^{v[p::=h]} \text{ for all } h \in H\}$$

Reduction on terms is defined in Figure 1 on page 239.

Remark 2.8. We will be particularly interested in models where the denotation of every λ-term is a member of H. This is because Definition 2.7 interprets λ as a kind of quantifier over all members of H. β-reduction then becomes a form of universal instantiation and so requires that all possible instantiations (i.e. the denotation of any term) is a member of H. More on this in Section 5.2.

Lemma 2.9. *β-reduction and η-expansion are sound, if we interpret reduction as subset inclusion:*

$$\begin{array}{lrcll} \beta\text{-reduction} & [\![\lambda p.\mathtt{t}]\!]^v \bullet [\![\mathtt{s}]\!]^v & \subseteq & [\![\mathtt{t}[p::=\mathtt{s}]]\!]^v & (\textit{if } [\![\mathtt{s}]\!]^v \in H) \\ \eta\text{-expansion} & [\![\mathtt{t}]\!]^v & \subseteq & [\![\lambda p.(\mathtt{t}{\cdot}p)]\!]^v & (p \textit{ not free in } \mathtt{t}) \end{array}$$

Proof. By routine calculations from the definitions. We prove a more general result in Theorem 2.23.

Remark 2.10. It may help to give some indication of what the *canonical* frame used in the completeness proof for $\mathcal{L}_\lambda$ (Definition 3.5) looks like: worlds are β-reduction-η-expansion closures of λ-terms $\mathtt{t}$, and for each $h \in H$ there exists some $\mathtt{t}$ such that h is the set of worlds that contain $\mathtt{t}$.

As we emphasised in Remark 2.2, our frames are not applicative structures, and the denotations of λ-terms are not worlds, but sets of worlds. Thus, in the canonical frame, the denotation of a λ-term $\mathtt{t}$ is *not* the set of its reducts (i.e. not some world in the canonical frame). Rather, the denotation of $\mathtt{t}$ is the set of all worlds that *reduce* to $\mathtt{t}$.

We can identify a world with its 'top' term, so roughly speaking, in the canonical model a world $w \in W$ is a term $\mathtt{t}$, and an $h \in H$ (or any denotation) is a set of all terms which reduce to some particular term $\mathtt{s}$.

Remark 2.11. We suggest an intuition why our models 'have to' satisfy β-reduction and η-expansion. Both β-reduction and η-expansion *lose information*: in the case of β we perform the substitution as is usual; in the case of η-expansion $\lambda p.(\mathtt{t}{\cdot}p)$ has lost any intensional information that might reside in $\mathtt{t}$. So we consider validating η-expansion as an interesting feature, and not necessarily a bug.

Others have also noted good properties and justification in models for η-expansion [14]. It *is* possible to refine the models to eliminate η-expansion, at some cost in complexity; see the Conclusions.

We will fill in more details of the semantics of λ-terms in Section 2.4, including the role of H, once we have built the logic in Section 2.3.

2.3 The Logic

Definition 2.12. Define a language $\mathcal{L}$ with **sentences** ϕ by:

$$\phi ::= p, q, r \ldots \mid \phi \to \phi \mid \forall p.\, \phi \mid \Box\phi \mid \phi{\cdot}\phi \mid \phi \rhd \phi \mid \bot$$

$\forall p$ binds in $\forall p.\, \phi$. For example, p is bound in $\forall p.\, (p{\cdot}q)$. We identify sentences up to α-equivalence.

We now give notions of logical entailment and denotation for $\mathcal{L}$. In Section 2.4 we discuss expressive power and in Sections 3 and 4 we sketch proofs of soundness and completeness.

Definition 2.13. Suppose $F = (W, \bullet, H)$ is a frame and v is a valuation to F. Define $[\![\phi]\!]^v$ the **denotation** of ϕ by:

$$\begin{array}{rlrl} [\![p]\!]^v &= v(p) & [\![\bot]\!]^v &= \varnothing \\ [\![\phi{\cdot}\psi]\!]^v &= [\![\phi]\!]^v \bullet [\![\psi]\!]^v & [\![\phi \rhd \psi]\!]^v &= \{w \mid w \bullet [\![\phi]\!]^v \subseteq [\![\psi]\!]^v\} \\ [\![\phi \to \psi]\!]^v &= (W \setminus [\![\phi]\!]^v) \cup [\![\psi]\!]^v & [\![\forall p.\, \phi]\!]^v &= \bigcap_{h \in H} [\![\phi]\!]^{v[p \mapsto h]} \end{array}$$

$$[\![\Box\phi]\!]^v = \begin{cases} W & [\![\phi]\!]^v = W \\ \varnothing & [\![\phi]\!]^v \neq W \end{cases}$$

Remark 2.14. Intuitions are as follows:

– p, q, r, are variables ranging over subsets of W.

– $\phi \to \psi$ is classical implication.

– $\forall p.\, \phi$ is a quantification over elements of H. Think of $\forall p.\, \phi$ as 'the intersection of the denotation of ϕ for all of a pre-selection of possible denotations of p'. The possible denotation of p are subsets of W and not elements of the W; pre-selection is done by H, which identifies those denotations that 'exist' in the sense of being in the ranges of the quantifiers. More on this later.

– $\Box\phi$ is a notion of *necessity*. $\Box\phi$ is either W or $\varnothing$ depending on whether ϕ is itself W or not.
$\Box$ is the modality of S5 [11].

– $\phi{\cdot}\psi$ is a notion of application; the construction in Definition 2.4 ensures that the interpretation of $\cdot$ it is monotonic with respect to subset inclusion.[2]

The maps $\phi{\cdot}$- and -$\cdot\psi$ behave like the box operator of the modal logic K.

– $\rhd$ is the right adjoint to $\cdot$ with respect to $\to$. It is easily verified from Definition 2.13 that $[\![\phi{\cdot}\psi]\!]^v \subseteq [\![\mu]\!]^v$ exactly when $[\![\phi]\!]^v \subseteq [\![\psi \rhd \mu]\!]^v$. So $\phi \rhd \psi$ is interpreted as the largest subset of W that when applied to ϕ, is included in ψ.

$\mathcal{L}$ is a second-order classical propositional logic enriched with the necessity modality $\Box$ from $S5$, and notions of application $\cdot$ and its right adjoint $\rhd$ (with respect to logical implication $\to$).

When we mix all these ingredients, interesting things become expressible, as we now explore.

2.4 Expressivity

Remark 2.15. We can express truth, negation, conjunction, disjunction, if-and-only-if and existential quantification as below. We also unpack Definition 2.13 to see this denotationally:

$$\begin{array}{rl} \top = (\bot \to \bot) & [\![\top]\!]^v = W \\ \neg\phi = \phi \to \bot & [\![\neg\phi]\!]^v = W \setminus [\![\phi]\!]^v \\ \phi \wedge \psi = \neg(\phi \to \neg\psi) & [\![\phi \wedge \psi]\!]^v = [\![\phi]\!]^v \cap [\![\psi]\!]^v \\ \phi \vee \psi = (\neg\phi) \to \psi & [\![\phi \vee \psi]\!]^v = [\![\phi]\!]^v \cup [\![\psi]\!]^v \\ \exists p.\, \phi = \neg(\forall p.\, \neg\phi) & [\![\exists p.\, \phi]\!]^v = \bigcup_{h \in H} [\![\phi]\!]^{v[p \mapsto h]}. \end{array}$$

[2] That is, $[\![\phi]\!]^v \subseteq [\![\phi']\!]^v$ and $[\![\psi]\!]^v \subseteq [\![\psi']\!]^v$ implies $[\![\phi{\cdot}\psi]\!]^v \subseteq [\![\phi'{\cdot}\psi']\!]^v$.

Note that $\exists.$ quantifies over elements of H. This is all standard, which is the point.

Remark 2.16. For the reader familiar with the expression of product and other types in System F [10], note that $[\![\neg(\phi{\to}\neg\psi)]\!]^v \neq [\![\forall p.\,(\phi{\to}\psi{\to}p){\to}p]\!]^v$ in general; H may be too sparse. The equality holds in a frame if $[\![\phi]\!]^v \in H$ for every ϕ. We can specify this condition as an (infinite) theory using an 'existence' predicate E (Definition 2.18).

Definition 2.17. We can express that two predicates *are* equal by: $\phi \approx \psi = \Box(\phi \leftrightarrow \psi)$.

We unpack the denotation of $\phi \approx \psi$ and for comparison also that of $\phi \leftrightarrow \psi$ $((\phi \to \psi) \wedge (\psi \to \phi))$:

$$[\![\phi \approx \psi]\!]^v = \begin{cases} W & [\![\phi]\!]^v = [\![\psi]\!]^v \\ \varnothing & [\![\phi]\!]^v \neq [\![\psi]\!]^v \end{cases}$$

$$[\![\phi \leftrightarrow \psi]\!]^v = \{w \in W \mid (w \in [\![\phi]\!]^v \wedge w \in [\![\psi]\!]^v) \vee (w \notin [\![\phi]\!]^v \wedge w \notin [\![\psi]\!]^v)\}$$

Intuitively, $\phi \leftrightarrow \psi$ holds at the worlds where ϕ and ψ are either both true or both false, whereas $\phi \approx \psi$ represents the statement 'ϕ and ψ are true of the same worlds'.

Definition 2.18. We can express that a predicate is in H by $\mathsf{E}\phi = \exists p.\,(\phi \approx p)$, read '$\phi$ exists'. This is usually called an *existence predicate* [13, Ch.16].

It is not hard to verify that $\mathsf{E}\phi$ has the following denotation:

$$[\![\mathsf{E}\phi]\!]^v = \begin{cases} W & [\![\phi]\!]^v \in H \\ \varnothing & [\![\phi]\!]^v \notin H \end{cases}$$

We are now ready to interpret λ-abstraction in our logic. We also mention a notion of matching, because it comes very naturally out of the logic as a 'dual' to the construction for λ:

Definition 2.19. $\curlywedge p.\,\phi = \forall p.\,(p \rhd \phi) \qquad \mathsf{match}\, p.\,\phi = \forall p.\,(\phi \rhd p).$

Intuitively, $\curlywedge p.\,\phi$ reads as: 'for any p, if p is an argument (of the function instantiated at this world), then we get ϕ'. As a kind of inverse to this, $\mathsf{match}\, p.\,\phi$ reads as: 'for any p, if ϕ is an argument, then we get p'. So $\mathsf{match}\, p.\,\phi$ is a kind of pattern-matching or inverse-λ.

$\curlywedge$ is a logical quantifier, so we name it reversed by analogy with the reversed A and E of universal and existential quantification.

Theorem 2.20. $- [\![\curlywedge p.\,\phi)]\!]^v = \{w \mid w \bullet h \subseteq [\![\phi]\!]^{v[p\mapsto h]}$ *for all* $h \in H\}$
$- [\![\mathsf{match}\, p.\,\phi]\!]^v = \{w \mid w \bullet [\![\phi]\!]^{v[p\mapsto h]} \subseteq h$ *for all* $h \in H\}$

Proof.

$$\begin{array}{lll}
[\![\lambda\!\!\lambda p.\,\phi]\!]^v & = [\![\forall p.\,(p \rhd \phi)]\!]^v & \text{Definition 2.19} \\
& = \bigcap_{h\in H} [\![p \rhd \phi]\!]^{v[p\mapsto h]} & \text{Definition 2.13} \\
& = \bigcap_{h\in H} \{w \mid w \bullet [\![p]\!]^{v[p\mapsto h]} \subseteq [\![\phi]\!]^{v[p\mapsto h]}\} & \text{Definition 2.13} \\
& = \{w \mid w \bullet h \subseteq [\![\phi]\!]^{v[p\mapsto h]} \text{ for all } h \in H\} & \text{Definition 2.4}
\end{array}$$

The case for $\mathsf{match}\, p.$ is similar.

Lemma 2.21. *If p is not free in ϕ, then for any $h \in H$, $[\![\phi]\!]^v = [\![\phi]\!]^{v[p\mapsto h]}$.*

Proof. An easy induction on ϕ.

Lemma 2.22 (Substitution Lemma). *For any v, $[\![\phi[p::=\psi]]\!]^v = [\![\phi]\!]^{v[p\mapsto [\![\psi]\!]^v]}$.*

Proof. By induction on ϕ, we present the cases for $\forall q$ and $\cdot$:

$$\begin{array}{lll}
[\![(\forall q.\,\mu)[p::=\psi]]\!]^v & = \bigcap_{h\in H} [\![\mu[p::=\psi]]\!]^{v[q\mapsto h]} & \text{Definition 2.13} \\
& = \bigcap_{h\in H} [\![\mu]\!]^{v[q\mapsto h, p\mapsto [\![\psi]\!]^{v[q\mapsto h]}]} & \text{Induction Hypothesis} \\
& = \bigcap_{h\in H} [\![\mu]\!]^{v[q\mapsto h, p\mapsto [\![\psi]\!]^{v}]} & \text{Lemma 2.21} \\
& = [\![\forall q.\,\mu]\!]^{v[p\mapsto [\![\psi]\!]^v]} & \text{Definition 2.13}
\end{array}$$

$$\begin{array}{lll}
[\![(\mu_1\cdot\mu_2)[p::=\psi]]\!]^v & = [\![\mu_1[p::=\psi]]\!]^v \bullet [\![\mu_2[p::=\psi]]\!]^v & \text{Definition 2.13} \\
& = [\![\mu_1]\!]^{v[p\mapsto [\![\psi]\!]^v]} \bullet [\![\mu_2]\!]^{v[p\mapsto [\![\psi]\!]^v]} & \text{Induction Hypothesis} \\
& = [\![\mu_1\cdot\mu_2]\!]^{v[p\mapsto [\![\psi]\!]^v]} & \text{Definition 2.13}
\end{array}$$

Theorem 2.23. *The following hold in any frame:*

$$\begin{array}{lrcll}
\textit{(β-reduction)} & [\![(\lambda\!\!\lambda p.\,\phi)\cdot\psi]\!]^v & \subseteq & [\![\phi[p::=\psi]]\!]^v & \textit{(for } [\![\psi]\!]^v \in H\textit{)} \\
\textit{(η-expansion)} & [\![\phi]\!]^v & \subseteq & [\![\lambda\!\!\lambda p.\,(\phi\cdot p)]\!]^v & \textit{(for } p \textit{ not free in } \phi\textit{)} \\
\textit{(matching)} & [\![(\mathsf{match}\, p.\,\phi)\cdot(\phi[p::=\psi])]\!]^v & \subseteq & [\![\psi]\!]^v & \textit{(for } [\![\psi]\!]^v \in H\textit{)}
\end{array}$$

Proof.

$$\begin{array}{lll}
& [\![\forall p.\,(p \rhd \phi)\cdot\psi]\!]^v & \\
& \quad = [\![\forall p.\,(p \rhd \phi)]\!]^v \bullet [\![\psi]\!]^v & \text{Definition 2.13} \\
& \quad = \bigcap_{h\in H}\{w \mid w \bullet h \subseteq [\![\phi]\!]^{v[p\mapsto h]}\} \bullet [\![\psi]\!]^v & \text{Definition 2.13} \\
& \quad \subseteq \{w \mid w \bullet [\![\psi]\!]^v \subseteq [\![\phi]\!]^{v[p\mapsto [\![\psi]\!]^v]}\} \bullet [\![\psi]\!]^v & [\![\psi]\!]^v \in H \\
& \quad \subseteq [\![\phi]\!]^{v[p\mapsto [\![\psi]\!]^v]} & \text{Definition 2.4} \\
& \quad = [\![\phi[p::=\psi]]\!]^v & \text{Lemma 2.22}
\end{array}$$

$$\begin{array}{lll}
[\![\phi]\!]^v \subseteq \{w \mid w \bullet S \subseteq [\![\phi]\!]^v \bullet S \text{ for any } S \in H\} & & \text{Definition 2.4} \\
\quad = \bigcap_{h\in H}\{w \mid w \bullet S \subseteq [\![\phi\cdot p]\!]^{v[p\mapsto h]}\} & & p \text{ not free in } [\![\psi]\!]^v \\
\quad = [\![\lambda\!\!\lambda p.\,(\phi\cdot p)]\!]^v & & \text{Definition 2.19}
\end{array}$$

(matching) follows by a similarly routine calculation.

$$(Eq)\ \mathtt{t} \twoheadrightarrow \mathtt{t} \qquad (\beta)\ (\lambda p.\mathtt{t})\cdot\mathtt{s} \twoheadrightarrow \mathtt{t}[p::=\mathtt{s}] \qquad (\eta)\ \mathtt{t} \twoheadrightarrow \lambda p.(\mathtt{t}\cdot p) \quad (p \text{ not free in } \mathtt{t})$$

$$(\xi)\ \frac{\mathtt{t} \twoheadrightarrow \mathtt{s}}{\lambda p.\mathtt{t} \twoheadrightarrow \lambda p.\mathtt{s}} \qquad (cong)\ \frac{\mathtt{t}_1 \twoheadrightarrow \mathtt{s}_1 \quad \mathtt{t}_2 \twoheadrightarrow \mathtt{s}_2}{\mathtt{t}_1\cdot\mathtt{t}_2 \twoheadrightarrow \mathtt{s}_1\cdot\mathtt{s}_2} \qquad (trans)\ \frac{\mathtt{t}_1 \twoheadrightarrow \mathtt{t}_2 \quad \mathtt{t}_2 \twoheadrightarrow \mathtt{t}_3}{\mathtt{t}_1 \twoheadrightarrow \mathtt{t}_3}$$

Fig. 1. λ-reduction

Corollary 2.24.

- *If p is not free in ϕ and $[\![\bot]\!]^v \in H$ then $[\![(\mathsf{match}\, p.\, \phi)\cdot\phi]\!]^v = \varnothing$.*
- *If $[\![\psi]\!]^v \in H$ then $[\![(\mathsf{match}\, p.\, \phi)\cdot((\not\lambda p.\, \phi)\cdot\psi)]\!]^v \subseteq [\![\psi]\!]^v$.*
- $\mathsf{match}\, p.\, p = \not\lambda p.\, p$.

Proof. The first two parts follow easily from Theorem 2.23. The third part follows unpacking definitions, since both are equal to $\forall p.\, (p \rhd p)$.

Read $(\mathsf{match}\, p.\, \phi)\cdot\mu$ as returning the intersection of all ψ such that μ is equivalent to $\phi[p::=\psi]$. If there are many such ψ, e.g. when p is not free in ϕ, then $(\mathsf{match}\, p.\, \phi)\cdot\mu \to \psi$ for all such ψ and so $(\mathsf{match}\, p.\, \phi)\cdot\mu$ is included in their intersection.

Definition 2.25. Define a translation τ from $\mathcal{L}_\lambda$ (Definition 2.5) to $\mathcal{L}$ (Definition 2.12) by:

$$p^\tau = p \qquad (\mathtt{t}_1\cdot\mathtt{t}_2)^\tau = (\mathtt{t}_1^\tau\cdot\mathtt{t}_2^\tau) \qquad (\lambda p.\mathtt{t})^\tau = \not\lambda p.\, \mathtt{t}^\tau$$

Definition 2.26. Write $\mathtt{t} \twoheadrightarrow \mathtt{s}$ if $\mathtt{t} \twoheadrightarrow \mathtt{s}$ is derivable using the axioms of Figure 1.

Our implementation of λ is *sound* in the following sense:

Theorem 2.27. $\mathtt{t} \twoheadrightarrow \mathtt{s}$ *only if* $[\![\mathtt{t}^\tau \to \mathtt{s}^\tau]\!]^v = W$ *for all* v *and* $F = (W, \bullet, H)$ *such that* $[\![\mathtt{u}^\tau]\!]^v \in H$ *for all* $\mathtt{u}$.[3]

Proof. This follows (mostly) by Theorem 2.23.

3 Completeness for λ-Reduction

In this section we show that the axiomatisation of λ-reduction of Figure 1 is complete for our interpretation of λ-terms in terms of $\not\lambda$. We do this by proving the converse of Theorem 2.27.

To complete Theorem 2.27 we must show that if $\mathtt{t} \not\twoheadrightarrow \mathtt{s}$ then there is a frame and valuation v where $[\![\mathtt{u}^\tau]\!]^v \in H$ for all $\mathtt{u}$ and $[\![\mathtt{t}^\tau \to \mathtt{s}^\tau]\!]^v \neq W$ (where τ is the translation of Definition 2.25).

[3] In other words, if $[\![\mathsf{E}(\mathtt{u}^\tau)]\!]^v = W$ for all terms $\mathtt{u}$.

Definition 3.1. Say a λ-term is **complex** if it contains term formers, i.e. is not simply a variable. The **size** of a λ-term is the number of term formers within it.

Now we can begin the construction of the desired frame. First we add infinitely many new variables $r_1, r_2 \ldots$ to the language $\mathcal{L}_\lambda$. Since the language is countable we can enumerate its complex terms $\mathtt{t}_1, \mathtt{t}_2 \ldots$ and these new variables $r_1, r_2 \ldots$. We describe a one-one function f from terms to variables.

> $f(\mathtt{t}_i) = r_j$ where j is the least number such that $j > i$ and r_j is not free in $\mathtt{t}_i$ nor is the value under f of any $\mathtt{t}_k$ for $k < i$.

Thus f is a one-one function that assigns a distinct 'fresh' variable to each complex term of the language. Thus $f(\mathtt{t})$ is a variable that 'names' $\mathtt{t}$. These play the role of witness constants in the construction of the canonical frame in Theorem 3.7. The $f(\mathtt{t})$ also help us carry out inductions on the size of λ-terms, as $\mathtt{t}[p::=f(\mathtt{s})]$ is smaller than $\lambda p.\mathtt{t}$ even if $\mathtt{t}[p::=\mathtt{s}]$ might not be.

Definition 3.2. Next we add two new axioms of reduction, denote them by (ζ_f):

$$\mathtt{t} \twoheadrightarrow f(\mathtt{t}) \qquad f(\mathtt{t}) \twoheadrightarrow \mathtt{t} \tag{ζ_f}$$

Write $\mathtt{t} \twoheadrightarrow_{\zeta_f} \mathtt{s}$ when $\mathtt{t} \twoheadrightarrow \mathtt{s}$ is derivable using the (ζ_f) rules in addition to the rules of Figure 1.

Lemma 3.3. *If* $\mathtt{t} \twoheadrightarrow_{\zeta_f} \mathtt{s}$ *and neither* $\mathtt{s}$ *or* $\mathtt{t}$ *contain any of the new variables* $r_1, r_2 \ldots$, *then* $\mathtt{t} \twoheadrightarrow \mathtt{s}$.

Proof. By simultaneously substituting each instance of $f(\mathtt{t}_i)$ with $\mathtt{t}_i$ each instance of (ζ_f) becomes an instance of (Eq) without affecting the rest of the derivation.

Definition 3.4. If $\mathtt{t}$ is a term let $w_\mathtt{t} = \{\mathtt{s} \mid \mathtt{t} \twoheadrightarrow_{\zeta_f} \mathtt{s}\}$. Thus $w_\mathtt{t}$ is the closure of $\mathtt{t}$ under $\twoheadrightarrow_{\zeta_f}$.

Definition 3.5. Define the **canonical λ-frame** $F_\lambda = \langle W_\lambda, \bullet_\lambda, H_\lambda \rangle$ as follows:

- $W_\lambda = \{w_\mathtt{t} \mid \mathtt{t} \text{ is a term}\}$
- For any $w_{\mathtt{t}_1}, w_{\mathtt{t}_2} \in W$, $w_{\mathtt{t}_1} \bullet_\lambda w_{\mathtt{t}_2} = \{w \in W_\lambda \mid \mathtt{t}_1 \cdot \mathtt{t}_2 \in w\}$
- $H_\lambda = \big\{\{w \in W_\lambda \mid \mathtt{t} \in w\} \mid \ \mathtt{t} \text{ is a term}\big\}$

Definition 3.6. Given $F_\lambda = \langle W_\lambda, \bullet_\lambda, H_\lambda \rangle$ and a term $\mathtt{t}$. Let $\|\mathtt{t}\| = \{w \in W_\lambda \mid \mathtt{t} \in w\}$.

Theorem 3.7. *Let* F_λ *be the canonical* λ*-frame (Definition 3.5). Let* τ *be the translation from* λ*-terms* $\mathtt{t}$ *to sentences* ϕ *(Definition 2.25). Let* $v(p) = \|p\|$ *for any variable* p. *Then for any term* $\mathtt{t}$, $[\![\mathtt{t}^\tau]\!]^v = \|\mathtt{t}\|$.

Proof. By induction on the size of $\mathtt{t}$ we show that $w \in \|\mathtt{t}\|$ (i.e. $\mathtt{t} \in w$) if and only if $w \in [\![\mathtt{t}^\tau]\!]^v$.

– $\mathtt{t}$ is a variable p. Then $\|p\| = v(p) = [\![p]\!]^v$ by the definition of v.

– $\mathtt{t}$ is $\mathtt{t}_1{\cdot}\mathtt{t}_2$. Then $(\mathtt{t}_1{\cdot}\mathtt{t}_2)^\tau = \mathtt{t}_1^\tau{\cdot}\mathtt{t}_2^\tau$.

Suppose $\mathtt{t}_1{\cdot}\mathtt{t}_2 \in w$, and consider the worlds $w_{\mathtt{t}_1}$ and $w_{\mathtt{t}_2}$ in W_λ. If $\mathtt{s}_1 \in w_{\mathtt{t}_1}$ and $\mathtt{s}_2 \in w_{\mathtt{t}_2}$ then by Definition 3.4, $\mathtt{t}_1 \twoheadrightarrow_{\zeta_f} \mathtt{s}_1$ and $\mathtt{t}_2 \twoheadrightarrow_{\zeta_f} \mathtt{s}_2$. Thus $\mathtt{t}_1{\cdot}\mathtt{t}_2 \twoheadrightarrow_{\zeta_f} \mathtt{s}_1{\cdot}\mathtt{s}_2$ and $\mathtt{s}_1{\cdot}\mathtt{s}_2 \in w$. Then by the definition of $\bullet_\lambda$ we have that $w \in w_{\mathtt{t}_1} \bullet_\lambda w_{\mathtt{t}_2}$. Furthermore, $w_{\mathtt{t}_1} \in \|\mathtt{t}_1\|$ and so by the induction hypothesis, $w_{\mathtt{t}_1} \in [\![\mathtt{t}_1^\tau]\!]^v$. Similarly $w_{\mathtt{t}_2} \in [\![\mathtt{t}_2^\tau]\!]^v$. Hence $w \in [\![\mathtt{t}_1^\tau{\cdot}\mathtt{t}_2^\tau]\!]^v$ by Definition 2.13.

Conversely, suppose that $w \in [\![\mathtt{t}_1^\tau{\cdot}\mathtt{t}_2^\tau]\!]^v$. Then there are $w_{\mathtt{s}_1}, w_{\mathtt{s}_2}$ such that $w_{\mathtt{s}_1} \in [\![\mathtt{t}_1^\tau]\!]^v$ and $w_{\mathtt{s}_2} \in [\![\mathtt{t}_2^\tau]\!]^v$ and $w \in w_{\mathtt{s}_1} \bullet_\lambda w_{\mathtt{s}_2}$. By the induction hypothesis $w_{\mathtt{s}_1} \in \|\mathtt{t}_1\|$ and $w_{\mathtt{s}_2} \in \|\mathtt{t}_2\|$. Then $\vdash \mathtt{s}_1 \twoheadrightarrow_{\zeta_f} \mathtt{t}_1$ and $\vdash \mathtt{s}_2 \twoheadrightarrow_{\zeta_f} \mathtt{t}_2$. Furthermore, by the construction of $\bullet_\lambda$, $\mathtt{s}_1{\cdot}\mathtt{s}_2 \in w$ and hence by (*cong*) $\mathtt{t}_1{\cdot}\mathtt{t}_2 \in w$.

– $\mathtt{t}$ is $\lambda p.\mathtt{s}$. Then $[\![(\lambda p.\mathtt{s})^\tau]\!]^v = [\![\curlywedge p.\,\mathtt{s}^\tau]\!]^v = \{w \mid \forall h \in H_\lambda . w \bullet_\lambda h \subseteq [\![\mathtt{s}^\tau]\!]^{v[p \mapsto h]}\}$.

Suppose $\lambda p.\mathtt{s} \in w_1$. Suppose that $w_3 \in w_1 \bullet_\lambda w_2$, and that $w_2 \in h$ for some $h \in H_\lambda$, then $h = \|\mathtt{u}\|$ for some term $\mathtt{u}$. By (ζ_f) we have that $\mathtt{u} \twoheadrightarrow_{\zeta_f} r$ and $r \twoheadrightarrow_{\zeta_f} \mathtt{u}$ for some r. So $h = \|r\|$ and $r \in w_2$. By the construction of $\bullet_\lambda$, $\lambda p.\mathtt{s}{\cdot}r \in w_3$ and so $\mathtt{s}[p ::= r] \in w_3$, i.e. $w_3 \in \|\mathtt{s}[p ::= r]\|$. By the induction hypothesis $\|\mathtt{s}[p ::= r]\| = [\![\mathtt{s}^\tau[p ::= r]]\!]^v$. Furthermore by Lemma 2.22 $[\![\mathtt{s}^\tau[p ::= r]]\!]^v = [\![\mathtt{s}^\tau]\!]^{v[p \mapsto [\![r]\!]^v]}$. But by the definition of v, $[\![r]\!]^v = \|r\|$, and so $w_3 \in [\![\mathtt{s}^\tau]\!]^{v[p \mapsto \|r\|]}$. But $h = \|r\|$ so $w_3 \in [\![\mathtt{s}^\tau]\!]^{v[p \mapsto h]}$. Thus $w_1 \in \{w \mid \forall h \in H_\lambda . w \bullet_\lambda h \subseteq [\![\mathtt{s}^\tau]\!]^{v[p \mapsto h]}\} = [\![(\lambda p.\mathtt{s})^\tau]\!]^v$. Hence, $\|\lambda p.\mathtt{s}\| \subseteq [\![(\lambda p.\mathtt{s})^\tau]\!]^v$.

Conversely, suppose that $\lambda p.\mathtt{s} \notin w_{\mathtt{u}}$ for some $\mathtt{u}$. Let q be a variable not free in $\mathtt{u}$ or $\mathtt{s}$ and consider the worlds w_q and $w_{\mathtt{u}\cdot q}$. If $\mathtt{s}[p ::= q] \in w_{\mathtt{u}\cdot q}$ then $\mathtt{u}{\cdot}q \twoheadrightarrow_{\zeta_f} \mathtt{s}[p ::= q]$, so $\lambda q.(\mathtt{u}{\cdot}q) \twoheadrightarrow_{\zeta_f} \lambda q(\mathtt{s}[p ::= q])$ by (ξ). But by our choice of q, (η) entails that $\mathtt{u} \twoheadrightarrow_{\zeta_f} \lambda q.(\mathtt{u}{\cdot}q)$. So $\mathtt{u} \twoheadrightarrow_{\zeta_f} \lambda q.\mathtt{s}[p ::= q]$, which contradicts our initial supposition that $\lambda p.\mathtt{s} \notin w_{\mathtt{u}}$, therefore $\mathtt{s}[p ::= q] \notin w_{\mathtt{u}\cdot q}$. In other words $w_{\mathtt{u}\cdot q} \notin \|\mathtt{s}[p ::= q]\|$. Therefore, by the induction hypothesis, $w_{\mathtt{u}\cdot q} \notin [\![\mathtt{s}^\tau[p ::= q]]\!]^v$. Since $[\![q]\!]^v = \|q\|$, it follows by Lemma 2.22 that $w_{\mathtt{u}\cdot q} \notin [\![\mathtt{s}^\tau]\!]^{v[p \mapsto \|q\|]}$. But clearly $w_{\mathtt{u}\cdot q} \in w_{\mathtt{u}} \bullet_\lambda w_q$, so it follows that $w_{\mathtt{u}} \notin \{w \mid \forall h \in H_\lambda . w \bullet_\lambda h \subseteq [\![\mathtt{s}^\tau]\!]^{v[p \mapsto h]}\}$. By the semantics of $(\lambda q.\mathtt{s})^\tau$ (i.e. $\curlywedge q.\,\mathtt{s}^\tau$), this means that $w_{\mathtt{u}} \notin [\![(\lambda q.\mathtt{s})^\tau]\!]^v$. Hence, since every $w \in W_\lambda$ is $w_{\mathtt{u}}$ for some $\mathtt{u}$, $[\![(\lambda p.\mathtt{s})^\tau]\!]^v \subseteq \|\lambda p.\mathtt{s}\|$.

We can now prove the converse of Theorem 2.27:

Theorem 3.8. *$\mathtt{t} \twoheadrightarrow \mathtt{s}$ if and only if $[\![\mathtt{t}^\tau \to \mathtt{s}^\tau]\!]^v = W$ for all v and all frames $F = (W, \bullet, H)$ such that $[\![\mathtt{u}^\tau]\!]^v \in H$ for all $\mathtt{u}$.*[4]

Proof. The left-right direction is Theorem 2.27.

If $\mathtt{t} \not\twoheadrightarrow \mathtt{s}$ then $\mathtt{t} \not\twoheadrightarrow_{\zeta_f} \mathtt{s}$ and so $\mathtt{s} \notin w_{\mathtt{t}}$ in F_λ. Therefore $\|\mathtt{t}\| \not\subseteq \|\mathtt{s}\|$ and so by Theorem 3.7 there is a valuation v such that $[\![\mathtt{t}^\tau]\!]^v \not\subseteq [\![\mathtt{s}^\tau]\!]^v$. Furthermore, $H_\lambda = \{[\![\mathtt{u}^\tau]\!]^v \mid \mathtt{u} \text{ is a } \lambda\text{-term}\}$.

4 The Axioms, Soundness and Completeness

We can axiomatise the interpretation of $\mathcal{L}$ given by Definition 2.13. Axioms are given in Figure 2.

[4] In other words, if $[\![\mathtt{E}(\mathtt{u}^\tau)]\!]^v = W$ for all terms $\mathtt{u}$.

$(\forall R)$ $\phi \to \forall p.\, \phi$ $\quad (p \notin \phi)$

$(\forall L)$ $\forall p.\, \phi \to \mathsf{E}\psi \to \phi[p::=\psi]$

$(\forall A)$ $\forall p.\, (\phi \to \psi) \to (\forall p.\, \phi \to \forall p.\, \psi)$

(Gen) $\dfrac{\mathsf{E}p \to \phi}{\forall p.\, \phi}$

$(\cdot K)$ $\phi{\cdot}(\psi \vee \mu) \to (\phi{\cdot}\psi) \vee (\phi{\cdot}\mu)$
$(\psi \vee \mu){\cdot}\phi \to (\psi{\cdot}\phi) \vee (\mu{\cdot}\phi)$

$(\cdot C)$ $\phi{\cdot}\exists p.\, \psi \to \exists p.\, (\phi{\cdot}\psi)$
$(\exists p.\, \psi){\cdot}\phi \to \exists p.\, (\psi{\cdot}\phi)$ $\quad (p \notin \phi)$

$(\rhd K)$ $(\phi \rhd \psi) \wedge (\phi \rhd \mu) \to \phi \rhd (\psi \wedge \mu)$
$(\psi \rhd \phi) \wedge (\mu \rhd \phi) \to (\psi \vee \mu) \rhd \phi$

$(\rhd C)$ $\forall p.\, (\phi \rhd \psi) \to \phi \rhd \forall p.\, \psi$
$\forall p.\, (\psi \rhd \phi) \to (\exists p.\, \psi \rhd \phi)$ $\quad (p \notin \phi)$

$(Prop)$ Propositional Tautologies and Modus Ponens

$(\rhd L)$ $((\phi \rhd \psi){\cdot}\phi) \to \psi$

$(\rhd R)$ $\phi \to (\psi \rhd \phi{\cdot}\psi)$

$(\bot)$ $(\phi{\cdot}\bot) \to \bot$
$(\bot{\cdot}\phi) \to \bot$

(N) $\dfrac{\phi_1 \to \cdots \to \phi_n \to \psi}{\Box\phi_1 \to \cdots \to \Box\phi_n \to \Box\psi}$ $\quad 0 \leq n$

(T) $\Box\phi \to \phi$

(5) $\neg\Box\phi \to \Box\neg\Box\phi$

$(\Box\cdot)$ $\Box(\phi \to \psi) \to (\phi{\cdot}\mu) \to (\psi{\cdot}\mu)$
$\Box(\phi \to \psi) \to (\mu{\cdot}\phi) \to (\mu{\cdot}\psi)$

$(\Box\rhd)$ $\Box(\phi \to \psi) \to (\psi \rhd \mu) \to (\phi \rhd \mu)$
$\Box(\phi \to \psi) \to (\mu \rhd \phi) \to (\mu \rhd \psi)$

Fig. 2. Axioms for $\mathcal{L}$, we write '$p \notin \phi$' as short 'p is not free in ϕ'

Definition 4.1. Let $\Gamma, \Delta \ldots$ denote sets of sentences. Write $\vdash \phi$ if ϕ is derivable using the rules of Figure 2. Write $\Gamma \vdash A$ when there are $\phi_1 \ldots \phi_n \in \Gamma$ such that $\vdash \phi_1 \to \ldots \phi_n \to \phi$ (associating to the right).

4.1 Theorems and Admissible Rules

Theorem 4.2. *The converses of* $(\cdot K)$,$(\cdot C)$,$(\rhd K)$, *and* $(\rhd C)$ *are all derivable. Also derivable are:*

$\forall p.\, \phi \leftrightarrow \forall p.\, (\mathsf{E}p \to \phi)$	$(\Box\phi{\cdot}\psi) \to \Box\phi$	$\Box\phi \to (\psi{\cdot}\mu) \to (\psi{\cdot}(\Box\phi \wedge \mu))$
$\forall p.\, \Box\phi \leftrightarrow \Box\forall p.\, \phi$	$(\psi{\cdot}\Box\phi) \to \Box\phi$	$\Box\phi \to (\psi{\cdot}\mu) \to ((\Box\phi \wedge \psi){\cdot}\mu)$
$\phi \rhd \top$	$\neg\Box\phi \to (\Box\phi \rhd \psi)$	$\Box\phi \to (\psi \rhd \mu) \to (\psi \rhd (\Box\phi \wedge \mu))$
$\bot \rhd \phi$	$\neg\Box\phi \to (\psi \rhd \neg\Box\phi)$	$\Box\phi \to ((\Box\phi \wedge \psi) \rhd \mu) \to (\psi \rhd \mu)$

Notice that the second sentence of the leftmost column is the Barcan formula [13, Ch.13].

If $n = 0$ then (N) becomes a simple necessitation rule stating that $\vdash A$ implies $\vdash \Box A$. From this we get the following group of inference rules, $(Subs)$:

$$(Subs) \quad \frac{\phi \to \psi}{(\phi{\cdot}\mu) \to (\psi{\cdot}\mu)} \qquad \frac{\phi \to \psi}{(\mu{\cdot}\phi) \to (\mu{\cdot}\psi)}$$

$$\frac{\phi \to \psi}{(\psi \rhd \mu) \to (\phi \rhd \mu)} \qquad \frac{\phi \to \psi}{(\mu \rhd \phi) \to (\mu \rhd \psi)}$$

4.2 Soundness

Theorem 4.3. *Suppose $F = (W, \bullet, H)$ is a frame. Then $\vdash \phi$ implies $[\![\phi]\!]^v = W$ for any v.*

Proof. By induction on derivations. Assume $\vdash \phi$. We consider each axiom and inference rule in turn.

To show that an axiom of the form $\phi \to \psi$ is sound it is enough to show that $[\![\phi]\!]^v \subseteq [\![\psi]\!]^v$ for any v, for that implies that $(W \setminus [\![\phi]\!]^v) \cup [\![\psi]\!]^v = W$.

– ϕ and is an instance of (*Prop*). The soundness of tautological consequence for the chosen interpretation of $\to$ is well known.

– Instances of $(\forall L)$. Given Lemma 2.22 $\bigcap_{h \in H} [\![\phi]\!]^{v[p \mapsto h]} \subseteq \bigcap_{h \in H} [\![\phi]\!]^{v[p \mapsto [\![\psi]\!]^v]}$ provided that $[\![\psi]\!]^v \in H$. But if $[\![\psi]\!]^v \notin H$ then $[\![\mathsf{E}\psi]\!]^v = \varnothing$ and the axiom is validated.

– The cases for $(\forall R)$ and $(\forall A)$ are equally straightforward.

– Instances of (*Gen*). By induction hypothesis $[\![\mathsf{E}p \to \phi]\!]^v = W$ for any v so $[\![\mathsf{E}p]\!]^{v[p \mapsto h]} \subseteq [\![\phi]\!]^{v[p \mapsto h]}$ for any h. But $[\![\mathsf{E}\phi]\!]^{v[p \mapsto h]} = W$ for all h and so $[\![\forall p.\, A]\!]^v = W$.

– Instances of $(\rhd L)$ and $(\rhd R)$. We reason using Definitions 2.13 and 2.4:

$$\begin{aligned}
[\![(\phi \rhd \psi) \cdot \phi]\!]^v &= [\![\phi \rhd \psi]\!]^v \bullet [\![\phi]\!]^v && \text{Definition 2.13} \\
&= \{w \mid w \bullet [\![\phi]\!]^v \subseteq [\![\psi]\!]^v\} \bullet [\![\phi]\!]^v && \text{Definition 2.13} \\
&\subseteq [\![\psi]\!]^v && \text{Definition 2.4}
\end{aligned}$$

$$\begin{aligned}
[\![\phi]\!]^v &\subseteq \{w \mid w \bullet [\![\psi]\!]^v \subseteq [\![\phi]\!]^v \bullet [\![\psi]\!]^v\} && \text{Definition 2.4} \\
&= \{w \mid w \bullet [\![\psi]\!]^v \subseteq [\![\phi \cdot \psi]\!]^v\} && \text{Definition 2.13} \\
&= [\![(\psi \rhd \phi \cdot \psi)]\!]^v && \text{Definition 2.13}
\end{aligned}$$

– Instances of $(\cdot K)$.

$$\begin{aligned}
&[\![(\phi \cdot \psi) \vee (\phi \cdot \mu)]\!]^v \\
&\quad = ([\![\phi]\!]^v \bullet [\![\psi]\!]^v) \cup ([\![\phi]\!]^v \bullet [\![\mu]\!]^v) && \text{Remark 2.15} \\
&\quad = \begin{array}{l} \bigcup\{w_1 \bullet w_2 \mid w_1 \in [\![\phi]\!]^v \ \&\ w_2 \in [\![\psi]\!]^v\} \\ \cup \bigcup\{w_1 \bullet w_2 \mid w_1 \in [\![\phi]\!]^v \& \ w_2 \in [\![\mu]\!]^v\} \end{array} && \text{Definition 2.4} \\
&\quad = \bigcup\{w_1 \bullet w_2 \mid w_1 \in [\![\phi]\!]^v \ \&\ w_2 \in ([\![\psi]\!]^v \cup [\![\mu]\!]^v)\} && \text{Definition 2.4} \\
&\quad = [\![\phi \cdot (\psi \vee \mu)]\!]^v && \text{Remark 2.15}
\end{aligned}$$

The other case for $(\cdot K)$ and the cases for $(\rhd K)$ are similar.

– Instances of $(\cdot C)$ and $(\rhd C)$.

$$\begin{aligned}
[\![\phi \cdot \exists p.\, \psi]\!]^v &= [\![\phi]\!]^v \bullet \textstyle\bigcup_{h \in H} [\![\psi]\!]^{v[p \mapsto h]} && \text{Remark 2.15} \\
&= \textstyle\bigcup_{h \in H} ([\![\phi]\!]^v \bullet [\![\psi]\!]^{v[p \mapsto h]}) && \text{Definition 2.4} \\
&= \textstyle\bigcup_{h \in H} ([\![\phi]\!]^{v[p \mapsto h]} \bullet [\![\psi]\!]^{v[p \mapsto h]}) && \text{Lemma 2.21} \\
&= [\![\exists p.\, (\phi \cdot \psi)]\!]^v && \text{Remark 2.15}
\end{aligned}$$

The other cases are similar.

– Instances of $(\bot)$. $[\![\bot{\cdot}\phi]\!]^v = [\![\phi{\cdot}\bot]\!]^v = [\![\phi]\!]^v \bullet [\![\bot]\!]^v = \varnothing$ etc.
– Instances of $(\dot{\Box})$. We must show that $[\![\Box(\phi \to \psi)]\!]^v \cap [\![\phi{\cdot}\mu]\!]^v \subseteq [\![\psi{\cdot}\mu]\!]^v$. This is trivial if $[\![\phi \to \psi]\!]^v \neq W$, so we may assume that $[\![\phi]\!]^v \subseteq [\![\psi]\!]^v$. But then $[\![\phi{\cdot}\mu]\!]^v \subseteq [\![\psi{\cdot}\mu]\!]^v$. The argument is similar for $(\Box\triangleright)$.
– Instances of (N). By induction hypothesis $\bigcap_i [\![\phi_i]\!]^v \subseteq \psi$ for any v. If for some ϕ_i, $[\![\phi_i]\!]^v \neq W$ then $[\![\Box\phi_i]\!]^v = \varnothing$ and so $\bigcap_i [\![\Box\phi_i]\!]^v \subseteq \Box\psi$. On the other hand, if $[\![\phi_i]\!]^v = W$ then $[\![\psi]\!]^v = [\![\Box\psi]\!]^v = W$ and again $\bigcap_i [\![\Box\phi_i]\!]^v \subseteq \Box\psi$.
– Axioms (T) and (5) are easily seen to be sound from semantic conditions for $\Box$.

4.3 Completeness

Definition 4.4. Say that a set of sentences Γ is **consistent** if $\Gamma \nvdash \bot$.

We will show that given a consistent set of sentences Γ we can construct a frame F and a valuation v such that $\bigcap_{\phi\in\Gamma} [\![\phi]\!]^v \neq \varnothing$.

Definition 4.5. A **maximal** set Δ is a consistent set such that

(1) $\phi \in \Delta$ or $\neg\phi \in \Delta$ for any ϕ,
(2) for every sentence ϕ there is some variable p such that $\phi \approx p \in \Delta$ (see Definition 2.17), and
(3) if $\neg\forall p.\, \phi \in \Delta$ then $\neg\phi[p::=q], \mathsf{E}q \in \Delta$ for some variable symbol q.

Remark 4.6. The second requirement on maximality ensures that every sentence ϕ is 'named' by some atomic variable. The third requirement is the more familiar condition that every existential quantifier have a 'Henkin witness'.

Lemma 4.7. *If Δ consistent then there exists a maximal set Δ' such that $\Delta \subseteq \Delta'$.*

Proof. Add two infinite collections of new propositional variable symbols r_1, $r_2 \ldots$, $c_1, c_2 \ldots$ to $\mathcal{L}$, then enumerate all sentences $\phi_0, \phi_1 \ldots$ and describe two one-one functions f, g from predicates to variables:

$f(\phi_i) = r_j$ where j is the least number such that $j > i$ and r_j is not free in ϕ_i nor in Δ nor is the value under f of any $\phi_k < \phi_i$.

$g(\forall p.\, \phi_i) = c_j$ where j is the least number such that $j > i$ and c_j is not free in ϕ_i nor in Δ nor is the value under f of any $\forall p.\, \phi_k < \forall p.\, \phi_i$. We also write $g(\forall p.\, \psi)$ as $g_{\forall p.\, \psi}$.

We now construct Δ_0, Δ_1,... as follows (using the enumeration $\phi_0, \phi_1 \ldots$ above, or a new one):

$\Delta_0 = \Delta \cup \{\phi \approx f(\phi)\} \cup \{\neg\forall p.\, \phi \to \big(\neg\phi[p::=g_{\forall p.\, \phi}] \wedge \mathsf{E}(g_{\forall p.\, \phi})\big)\}$ for all ϕ.

If $\Delta_n \cup \{\phi_n\}$ is inconsistent then $\Delta_{n+1} = \Delta_n \cup \{\neg\phi_n\}$, otherwise $\Delta_{n+1} = \Delta_n \cup \{\phi_n\}$.

Note that $\Delta_i \subseteq \Delta_j$ if $i \leq j$. Let $\Theta = \bigcup_n \Delta_n$. By construction $\Delta \subseteq \Theta$. We must prove Θ is maximal:

– Δ_0 is consistent: Suppose Δ_0 is inconsistent, then there are $\mu_1 \dots \mu_n \in \Delta_0$ such that $\vdash \mu_1 \to \dots \to \mu_n \to \bot$. Suppose the μ_i that do not occur in Δ are:

$$\phi_1 \approx f(\phi_1) \dots \phi_k \approx f(\phi_k)$$

and

$$\neg\forall p.\, \psi_1 \to \neg\psi_1[p::=g_{\forall p.\, \psi_1}] \dots \neg\forall p.\, \psi_l \to \neg\psi_l[p::=g_{\forall p.\, \psi_l}]$$

First simultaneously substitute $f(\phi_i)$ with ϕ_i. We get that $\vdash \mu'_1 \to \dots \to \mu'_n \to \bot$ where each μ'_i is either in Δ or is $\phi \approx \phi$ or is $\neg\forall p.\, \psi' \to \neg\psi'[p::=g_{\forall p.\, \psi}]$ (where $\psi' = \psi[f(\phi_i)::=\phi_i]$).

Let ρ be $\neg\forall p.\, \psi'_j \to \big(\neg\psi'_j[p::=g_{\forall p.\, \psi_j}] \wedge \mathsf{E}(g_{\forall p.\, \psi_j})\big)$ where ψ'_j is latest in the enumeration of all sentences. We may assume that $\rho = \mu'_n = \mu'_{m+1}$. Since $\vdash \mu'_1 \to \dots \to \mu'_m \to \rho \to \bot$ we have by $(Prop)$ that

$$\vdash \mu'_1 \to \dots \to \mu'_m \to \neg\forall p.\, \psi'_j$$

and

$$\vdash \mu'_1 \to \dots \to \mu'_m \to \mathsf{E}(g_{\forall p.\, \psi'_j}) \to (\forall p.\, \psi'_j \to \psi'_j[p::=g_{\forall p.\, \psi'_j}])$$

but by our choice of $g_{\forall p.\, \psi'_j}$ it follows by the quantifier axioms that $\vdash \mu'_1 \to \dots \to \mu'_m \to \forall p.\, \psi'_j$. So $\vdash \mu'_1 \to \dots \to \mu'_m \to \bot$.

We may conclude from this that $\mu'_1 \to \dots \to \mu'_l \to \bot$ is derivable where each μ'_i $(i < l \leq n)$ is either of the form $\phi \approx \phi$ or is in Δ. But this is impossible by the consistency of Δ.

– For every ϕ, either $\phi \in \Theta$ or $\neg\phi \in \Theta$: By the construction, either ϕ or $\neg\phi$ is added to some Δ_i. By the consistency of Θ, it is also deductively closed.

– By the construction of Δ_0, for every sentence ϕ, there is some variable p such that $\phi \leftrightarrow p \in \Delta_0 \subseteq \Theta$.

– If $\neg\forall p.\, \phi \in \Theta$ then for some c, $\neg\forall p.\, \phi \to (\neg\phi[p::=c] \wedge \mathsf{E}c) \in \Delta_0 \subseteq \Delta'$ and so $\neg\phi[p::=c], \mathsf{E}c \in \Theta$.

Thus Θ is indeed maximal.

Definition 4.8. If Θ is a maximal set then $\mathcal{C}_\Theta = \{\Delta \mid \Delta$ is maximal and $\Box\phi \in \Theta$ implies $\phi \in \Delta\}$.

Definition 4.9. Define the **canonical frame** $F_{\mathcal{C}_\Theta} = \langle \mathcal{C}_\Theta, \bullet_{\mathcal{C}_\Theta}, H_{\mathcal{C}_\Theta} \rangle$:

– For any $w_1, w_2 \in \mathcal{C}_\Theta$, $\quad w_1 \bullet_{\mathcal{C}_\Theta} w_2 = \{w \in \mathcal{C}_\Theta \mid \phi \in w_1$ & $\psi \in w_2$ implies $\phi{\cdot}\psi \in w\}$.

– $H_{\mathcal{C}_\Theta} = \big\{\{w \in \mathcal{C}_\Theta \mid \phi \in w\} \mid \mathsf{E}\phi \in \Theta\big\}$.

It follows by (T) that $\Theta \in \mathcal{C}_\Theta$.

Definition 4.10. Given $F_{\mathcal{C}_\Theta} = (\mathcal{C}_\Theta, \bullet_{\mathcal{C}_\Theta}, H_{\mathcal{C}_\Theta})$ and a sentence ϕ. Let $\|\phi\| = \{w \in \mathcal{C}_\Theta \mid \phi \in w\}$.

Theorem 4.11. *Let $F_{\mathcal{C}_\Theta}$ be the canonical frame, and let $v(p) = \|p\|$ for any (sentential) variable p. Then for any sentence ϕ, $[\![\phi]\!]^v = \|\phi\|$.*

Proof. By induction on ϕ.

$-$ ϕ is p for some variable p. Then $\|p\| = v(p) = [\![p]\!]^v$ by the definition of v.

$-$ ϕ is $\phi_1 \to \phi_2$.

Suppose that $\phi_1 \to \phi_2 \in w$. If $w \in [\![\phi_1]\!]^v$, then by the induction hypothesis $w \in \|\phi_1\|$, i.e. $\phi_1 \in w$. So $\phi_2 \in w$ and $w \in [\![\phi_2]\!]^v$ by the induction hypothesis. Thus $w \in W \setminus [\![\phi_1]\!]^v \cup [\![\phi_2]\!]^v$.

Conversely, suppose that $\phi_1 \to \phi_2 \notin w$. Then by $(Prop)$ $\neg\phi_1 \notin w$ and $\phi_2 \notin w$. By the induction hypothesis, and the maximality of w, we may conclude that $w \notin W \setminus [\![\phi_1]\!]^v$ and $w \notin [\![\phi_2]\!]^v$.

$-$ ϕ is $\bot$. By the consistency of every $w \in \mathcal{C}_\Theta$, $\|\bot\| = \varnothing = [\![\bot]\!]^v$.

$-$ ϕ is $\Box\psi$.

If $\Box\psi \in w$ then by (5) and the construction of $\mathcal{C}_\Theta$, $\neg\Box\psi \notin \Theta$. So $\Box\psi \in \Theta$ and $\psi \in w'$ for all $w' \in \mathcal{C}_\Theta$.

For the converse case suppose that $\psi \in w$ for all $w \in \mathcal{C}_\Theta$. Then since $\mathcal{C}_\Theta$ contains all maximal consistent sets containing $\{\mu \mid \Box\mu \in \Theta\}$ it follows that $\{\mu \mid \Box\mu \in \Theta\} \vdash \psi$. So by (N), $\{\Box\mu \mid \Box\mu \in \Theta\} \vdash \Box\psi$ and so $\Box\psi \in \Theta$. But by (T), (N) and (5), $\vdash \Box\psi \to \Box\Box\psi$, so $\Box\psi \in w'$ for any $w' \in \mathcal{C}_\Theta$.

$-$ ϕ is $\forall p.\,\psi$.

Suppose that $\forall p.\,\psi \in w$ then by $(\forall L)$ $\psi[p::=\mu] \in w$ whenever $\mathsf{E}\mu \in w$. By the cases above for $\Box$, $\psi[p::=\mu] \in w$ whenever $\mathsf{E}\mu \in \Theta$.[5] By the maximality of Θ, for each μ, there is a variable f_μ such that $\mu \approx f_\mu \in \Theta$. Thus $\|\mu\| = \|f_\mu\|$ and every $h \in H_{\mathcal{C}_\Theta}$ is $\|f_\mu\|$ for some μ such that $\mathsf{E}\mu \in \Theta$. By the induction hypothesis and Lemma 2.22, $w \in [\![\psi[p::=f_\mu]]\!]^{v[f_\mu \mapsto h]}$ for all $h \in H_{\mathcal{C}_\Theta}$. Thus $w \in [\![\forall p.\,\mu]\!]^v$.

Conversely, suppose that $\forall p.\,\psi \notin w$. Then by the maximality of w, $\neg\forall p.\,\psi \in w$ and so $\neg\psi'[p::=c] \wedge \mathsf{E}c \in w$ for some c. By the induction hypothesis and Lemma 2.22 we have that $w \notin [\![\psi']\!]^{v[p \mapsto [\![c]\!]^v]}$, we also have $\mathsf{E}c \in \Theta$, so $w \notin [\![\forall p.\,\psi]\!]^v$.

$-$ ϕ is $\phi_1 \cdot \phi_2$.

Suppose $w_3 \in [\![\phi_1 \cdot \phi_2]\!]^v$. Then there are w_1, w_2 such that $w_1 \in [\![\phi_1]\!]^v$ and $w_2 \in [\![\phi_2]\!]^v$ and $w_3 \in w_1 \bullet_{\mathcal{C}_\Theta} w_2$. By the induction hypothesis $w_1 \in \|\phi_1\|$ and $w_2 \in \|\phi_2\|$, so $\phi_1 \in w_1$ and $\phi_2 \in w_2$. This implies $\phi_1 \cdot \phi_2 \in w_3$.

For the converse case suppose that $\phi_1 \cdot \phi_2 \in w_3$, we must show that there are w_1, w_2 such that $w_1 \in [\![\phi_1]\!]^v$, $w_2 \in [\![\phi_2]\!]^v$ and $w_3 \in w_1 \bullet_{\mathcal{C}_\Theta} w_2$. Given the induction hypothesis, it is enough to construct two maximal sets Δ_1, Δ_2 such that $\phi_1 \in \Delta_1, \phi_2 \in \Delta_2$ and $\psi_1 \cdot \psi_2 \in w_3$ for every $\psi_1 \in \Delta_1, \psi_2 \in \Delta_2$. We must then verify that these two sets are in $\mathcal{C}_\Theta$ by showing that $\{\psi \mid \Box\psi \in \Theta\} \subseteq \Delta_1 \cap \Delta_2$. This is done in Lemma 4.12.

[5] $\mathsf{E}\mu$ is short for $\exists p.\,\Box(p \leftrightarrow \mu)$. The axioms for $S5$ (N),(T) and (5) entail that $\vdash \exists p.\,\Box(p \leftrightarrow \mu) \to \Box\exists p.\,\Box(p \leftrightarrow \mu)$. So, by the case for $\Box$, $\mathsf{E}\mu \in w$ iff $\mathsf{E}\mu \in w'$ for any $w, w' \in \mathcal{C}_\Theta$.

– The case where ϕ is $\phi_1 \rhd \phi_2$ is similar to that for $\phi_1 \cdot \phi_2$ and uses a lemma similar to Lemma 4.12.

Lemma 4.12. *If $\Delta \in \mathcal{C}_\Theta$ and $\phi_1 \cdot \phi_2 \in \Delta$, then there are two maximal sets $\Delta_1, \Delta_2 \in \mathcal{C}_\Theta$ such that*

(1) $\phi_1 \in \Delta_1, \phi_2 \in \Delta_2$
(2) $\psi_1 \cdot \psi_2 \in \Delta$ for every $\psi_1 \in \Delta_1, \psi_2 \in \Delta_2$
(3) $\Box\psi \in \Theta$ implies $\psi \in \Delta_1 \cap \Delta_2$ for any ψ.

Proof. Enumerate all sentences $\psi_0, \psi_1 \ldots$ and construct two sequences Φ_0, $\Phi_1, \ldots$ and $\Psi_0, \Psi_1, \ldots$:

$\Phi_0 = \{\neg\neg\phi_1\}$ and $\Psi_0 = \{\neg\neg\phi_2\}$

If ψ_n is not of the form $\forall p.\, \mu$ then:

$$\Phi_{n+1} = \begin{cases} \Phi_n \cup \{\neg\psi_n\} \text{ if } (\bigwedge\Phi_n \wedge \neg\psi_n)\cdot(\bigwedge\Psi_n) \in \Delta \\ \Phi_n \cup \{\psi_n\} \quad \text{otherwise} \end{cases}$$

$$\Psi_{n+1} = \begin{cases} \Psi_n \cup \{\neg\psi_n\} \text{ if } (\bigwedge\Phi_{n+1})\cdot(\bigwedge\Psi_n \wedge \neg\psi_n) \in \Delta \\ \Psi_n \cup \{\psi_n\} \quad \text{otherwise} \end{cases}$$

If ψ_n is of the form $\forall p.\, \mu$ then:

$$\Phi_{n+1} = \begin{cases} \Phi_n \cup \{\neg\forall p.\, \mu, \neg\mu[p::=c], \mathsf{E}c\} \\ \qquad\qquad \text{if } (\bigwedge\Phi_n \wedge \neg\forall p.\, \mu \wedge \neg\mu[p::=c] \wedge \mathsf{E}c)\cdot(\bigwedge\Psi_n) \in \Delta \\ \qquad\qquad \text{for some variable } c \\ \Phi_n \cup \{\forall p.\, \mu\} \text{ otherwise} \end{cases}$$

$$\Psi_{n+1} = \begin{cases} \Psi_n \cup \{\neg\forall p.\, \mu, \neg\mu[x::=c], \mathsf{E}c\} \\ \qquad\qquad \text{if } (\bigwedge\Phi_{n+1})\cdot(\bigwedge\Psi_n \wedge \neg\forall p.\, \mu \wedge \neg\mu[p::=c] \wedge \mathsf{E}c) \in \Delta \\ \qquad\qquad \text{for some variable } c \\ \Psi_n \cup \{\forall p.\, \mu\} \text{ otherwise} \end{cases}$$

Note that $\Phi_i \subseteq \Phi_j$ and $\Psi_i \subseteq \Psi_j$ if $i \leq j$. Let $\Delta_1 = \bigcup_n \Phi_n$ and $\Delta_2 = \bigcup_n \Psi_n$.

– Each $(\bigwedge\Phi_n)\cdot(\bigwedge\Psi_n) \in \Delta$:
By induction on n. If $n = 0$ then since $\phi_1 \cdot \phi_2 \in \Delta$ it follows by (*Prop*) and (*Subs*) that $\neg\neg\phi_1 \cdot \neg\neg\phi_2 \in \Delta$.

Assume that $(\bigwedge\Phi_n)\cdot(\bigwedge\Psi_n) \in \Delta$ but $(\bigwedge\Phi_{n+1})\cdot(\bigwedge\Psi_n) \notin \Delta$. First we must show that

$$(\bigwedge\Phi_n \wedge \neg\psi_n)\cdot(\bigwedge\Psi_n) \notin \Delta \quad \text{implies} \quad (\bigwedge\Phi_n \wedge \psi_n)\cdot(\bigwedge\Psi_n) \in \Delta. \qquad (\dagger)$$

By $(\cdot K)$, (*Prop*) and the consistency of Δ if $(\bigwedge\Phi_n \wedge \neg\psi_n)\cdot(\bigwedge\Psi_n) \notin \Delta$ and $(\bigwedge\Phi_n \wedge \psi_n)\cdot(\bigwedge\Psi_n) \notin \Delta$ then

$$\big(\bigwedge(\Phi_n \wedge \psi_n) \vee (\bigwedge\Phi_n \wedge \neg\psi_n)\big)\cdot(\bigwedge\Psi_n) \notin \Delta.$$

So by (*Prop*) and (*Subs*) $\big((\bigwedge\Phi_n \wedge (\psi_n \vee \neg\psi_n))\cdot(\bigwedge\Psi_n)\big) \notin \Delta$. But this entails that $(\bigwedge\Phi_n)\cdot(\bigwedge\Psi_n) \notin \Delta$ which is contrary to our assumption.

So the lemma holds if ψ_n is not of the form $\forall p.\,\mu$. Suppose ψ_n is of the form $\forall p.\,\mu$. We must show that:

$$(\bigwedge\Phi_n \wedge \neg\forall p.\,\mu \wedge \neg\mu[p::=c] \wedge \mathsf{E}c)\cdot(\bigwedge\Psi_n) \notin \Delta \quad \text{for all } c,$$

implies that

$$(\bigwedge\Phi_n \wedge \forall p.\,\mu)\cdot(\bigwedge\Psi_n) \in \Delta$$

Given †, we need only show that

$$(\bigwedge\Phi_n \wedge \neg\forall p.\,\mu \wedge \neg\mu[p::=c] \wedge \mathsf{E}c)\cdot(\bigwedge\Psi_n) \notin \Delta \quad \text{for all } c,$$

implies that

$$(\bigwedge\Phi_n \wedge \neg\forall p.\,\mu)\cdot(\bigwedge\Psi_n) \notin \Delta$$

If $(\bigwedge\Phi_n \wedge \neg\forall p.\,\mu)\cdot(\bigwedge\Psi_n) \in \Delta$ then by $(\cdot C)$, $\neg\forall p.\,\neg\big((\bigwedge\Phi_n \wedge \neg\forall p.\,\mu \wedge \neg\mu)\cdot(\bigwedge\Psi_n)\big) \in \Delta$.[6] But since Δ is maximal and every negated universal quantification has a witness:

$$\neg\big((\bigwedge\Phi_n \wedge \neg\forall p.\,\mu \wedge \neg\mu[p::=c])\cdot(\bigwedge\Psi_n)\big) \wedge \mathsf{E}c \in \Delta \quad \text{for some } c$$

But then $(\bigwedge\Phi_n \wedge \neg\forall p.\,\mu \wedge \neg\mu[p::=c] \wedge \mathsf{E}c)\cdot(\bigwedge\Psi_n) \notin \Delta$ (for some c).
So we can conclude that $(\bigwedge\Phi_n)\cdot(\bigwedge\Psi_n) \in \Delta$ implies that $(\bigwedge\Phi_{n+1})\cdot(\bigwedge\Psi_n) \in \Delta$. Analogous reasoning shows that this in turn implies that $(\bigwedge\Phi_{n+1})\cdot(\bigwedge\Psi_{n+1}) \in \Delta$

– Δ_1, Δ_2 are consistent:
Suppose Δ_1 is inconsistent, then there are $\mu_1 \ldots \mu_n \in \Delta_1$ such that $\vdash \mu_1 \rightarrow \cdots \rightarrow \mu_n \rightarrow \bot$. The μ_i must all be in some $\Phi_k \subseteq \Delta_1$, but as $(\bigwedge\Phi_k)\cdot(\bigwedge\Psi_k) \in \Delta$ this implies by $(\bot)$ that $\bot \in \Delta$. This is impossible since Δ is consistent. We may conclude analogously that Δ_2 is not inconsistent.

– For any ϕ, either $\phi \in \Delta_1$ or $\neg\phi \in \Delta_1$:
This follows from the fact that every $\phi \in \Delta_1$ is a ψ_i, and so either it or its negation is added to Φ_i. Similarly, $\phi \in \Delta_2$ or $\neg\phi \in \Delta_2$ for any ϕ.

– $\neg\forall p.\,\mu \in \Delta_1$ implies $\neg\mu[p::=c], \mathsf{E}c \in \Delta_1$ for some c:
$\neg\forall p.\,\mu$ is a ψ_{i+1} and so is added to Φ_{i+1}, but $\Phi_{i+2} = \Phi_{i+1} \cup \{\neg\forall p.\,\mu, \neg\mu[p::=c], \mathsf{E}c\}$ for some c.[7] Similarly for Δ_2.

So Δ_1 and Δ_2 are maximal, now we verify that they satisfy the conditions of the lemma.

(1) $\phi_1 \in \Delta_1$ and $\phi_2 \in \Delta_2$: By choice of Φ_0, Ψ_0 we have that $\neg\neg\phi_1 \in \Delta_1$ and $\neg\neg\phi_2 \in \Delta_2$, so by (*Prop*) and the maximality of Δ_1 and Δ_2 it follows that $\phi_1 \in \Delta_1$ and $\phi_2 \in \Delta_2$.

[6] As we may assume that p is not free in $\bigwedge\Phi_n$.

[7] We chose Φ_0 and Ψ_0 to be $\{\neg\neg\phi_1\}$ and $\{\neg\neg\phi_2\}$, to guarantee the that the construction did not begin with a sentence of the form $\neg\forall p.\,\mu$.

(2) $\psi_1 \cdot \psi_2 \in \Delta$ for every $\psi_1 \in \Delta_1, \psi_2 \in \Delta_2$. Choosing some suitable large i, we have that $\psi_1 \in \Phi_i$ and $\psi_2 \in \Psi_i$ and the result follows by (*Prop*), (*Subs*) and the fact that $\bigwedge \Phi_i \cdot \bigwedge \Psi_i \in \Delta$
(3) If $\Box\psi \in \Theta$ then $\Box\Box\psi \in \Theta$ and so, since $\Delta \in \mathcal{C}_\Theta$, $\Box\psi \in \Delta$. Now, if $\neg\Box\psi \in \Delta_1$ or $\neg\Box\psi \in \Delta_2$ then, by (1) and (2), $(\neg\Box\psi)\cdot\phi_2 \in \Delta$ or $\phi_1\cdot(\neg\Box\psi) \in \Delta$. But $\Box\psi \in \Delta$, so by Theorem 4.2 this implies that $(\Box\psi \wedge \neg\Box\psi)\cdot\phi_2 \in \Delta$ or $\phi_1\cdot(\Box\psi \wedge \neg\Box\psi) \in \Delta$. This is impossible since Δ_1 and Δ_2 are consistent. So $\Box\psi \in \Delta_1 \cap \Delta_2$ and by (T) $\psi \in \Delta_1 \cap \Delta_2$.

Theorem 4.13. *If Δ is consistent then $\bigcap_{\phi\in\Delta}[\![\phi]\!]^v \neq \varnothing$ for some frame F and valuation v.*

Proof. We have shown that Δ can be extended to a maximal set Θ which is in the canonical frame $F_{\mathcal{C}_\Theta}$. Then by Theorem 4.11, $\phi \in \Delta$ implies that $\Theta \in [\![\phi]\!]^v \in W_{\mathcal{C}_\Theta} \in F_{\mathcal{C}_\Theta}$, so $\Theta \in \bigcap_{\phi\in\Delta}[\![\phi]\!]^v \neq \varnothing$.

Corollary 4.14. *If $[\![\phi]\!]^v = W$ for all frames F, then $\vdash \phi$.*

Proof. If $\nvdash \phi$ then $\{\neg\phi\}$ is consistent, so there is a frame F such that $[\![\neg\phi]\!]^v \neq \varnothing$. By the semantics of negation it follows that $[\![\phi]\!]^v \neq W$.

We can use this result together with Definition 4.1 to simplify Theorem 3.8.

Corollary 4.15. $\mathtt{t} \twoheadrightarrow \mathtt{s}$ *if and only if* $\{\mathsf{E}\mathtt{u}^\tau \mid \mathtt{u}$ *is a* λ*-term*$\} \vdash \mathtt{t}^\tau \to \mathtt{s}^\tau$.

It is a further issue whether Corollory 4.15 holds if $\{\mathsf{E}\mathtt{t}^\tau \mid \mathtt{t}$ is a λ-term$\}$ is replaced with $\{\mathsf{E}\phi \mid \phi$ is a sentence$\}$, or even $\{\mathsf{E}\phi \mid \phi$ is a closed sentence$\}$. A result equivalent to the fact that the corollory does not hold for the assumptions $\{\mathsf{E}\mathtt{u}^\tau \mid \mathtt{u}$ is a closed λ-term$\}$ was shown by Plotkin in [18].

5 Conclusion

5.1 Negation and the Liar

How does our logic resolve the paradoxes that affected Church's original system? We can extend τ (Definition 2.25) to translate $(\neg\mathtt{t})^\tau$ to either $\neg(\mathtt{t}^\tau)$ or $(\lambda p.\,\neg p)\cdot\mathtt{t}^\tau$. The first case corresponds to negation as a term-former in λ-term syntax; the second case corresponds to treating negation as a constant-symbol.

In the first case, let L^τ be short for $\lambda p.\,\neg(p\cdot p)\cdot\lambda p.\,\neg(p\cdot p)$. Then we may use Theorem 2.27 and Corollary 4.14 to conclude that $\{\mathsf{E}(\mathtt{u}^\tau) \mid \mathtt{u}$ is a λ-term$\} \vdash L^\tau \to \neg L^\tau$. So just as with Church's system we get a sentence L^τ that implies its own negation. Since $[\![\neg L^\tau]\!]^v = W \setminus [\![L^\tau]\!]^v$ there is only one possible interpretation of L^τ: the empty set.

In the second case similar reasoning applies, but more interpretations of L^τ are available. This is because $(\lambda p.\,\neg p)\cdot\mathtt{t}^\tau$ may receive a different interpretation from $\neg\mathtt{t}^\tau$, even if we assume $\mathsf{E}(\mathtt{u}^\tau)$ for all terms $\mathtt{u}$. The reason for this is that $[\![\lambda p.\,\neg p]\!]^v = \bigcap_{h\in H}\{w \mid w \bullet h \subseteq W \setminus h\}$ and so contains only those $w \in W$

that relate, by $\cdot$, members of H to their complements. So although $h \in H$ has a complement in $\mathcal{P}(W)$, there may be no $w \in W$ to serve in the extension of $[\![\lambda p.\, \neg p]\!]^v$.

For example, if F is a frame where $[\![\top]\!]^v = W \in H$ then $w \in [\![\lambda p.\, \neg p]\!]^v$ implies $w \bullet W \subseteq \varnothing$ and so $w \bullet S \subseteq \varnothing$ for any $S \subseteq W$ (as $\bullet$ is monotonic with respect to $\subseteq$). So $w \in [\![\lambda p.\, \neg p]\!]^v$ implies $w \bullet w' = \varnothing$ for all $w' \in W$. So for such a frame F, $[\![(\lambda p.\, \neg p) \cdot \phi]\!]^v = \varnothing = [\![\bot]\!]^v$ for any ϕ!

What moral can we draw from this? The negations of λ-terms can be interpreted perfectly naturally in our models. Paradoxes are averted because they may translate to impossible structural properties on the frames. Our models might help design other extensions of the λ-calculus, by considering how these extensions behave when interpreted in the models.

5.2 Related Work

Multiplicative conjunction. $\mathcal{L}$ with its connective $\cdot$ (Definition 2.12) looks like a (classical) logic with a *multiplicative conjunction* $\otimes$, as in linear logic or bunched implications [8,17]. Multiplicative conjunction $\otimes$ does not have contraction, so that for example $A \otimes A$ is not equivalent to A.

However $\cdot$ is *not* a species of multiplicative conjunction. This is because multiplicative conjunction is usually taken to be associative and commutative, whereas $\cdot$ is neither; it models application, and we do not usually want $f(x)$ to equal $x(f)$, or $f(g(x))$ to equal $(f(g))(x)$.[8]

Phase spaces. On a related point, a frame F based on a set W with its function $\bullet$ from Definition 2.1 looks like a phase space [9]. Indeed the denotation for λ in Definition 2.7 uses the same idea as the denotation of multiplicative implication $\multimap$ (see for example [9, Section 2.1.1]).

However F is unlike a phase space in one very important respect: $\bullet$ does not make W into a commutative monoid because it maps $W \times W$ to $\mathcal{P}(W)$, and not to W. This is also, as we have mentioned, why W is not an applicative structure.

An interesting interpretation of our models. The 'application operation' $\bullet$ returns not worlds but *sets* of worlds. In Section 2.1 we have already suggested that we can read $\bullet$ as a ternary Kripke accessability relation, or as a non-deterministic application operation. We would now like to suggest another reading, which we have found useful.

Think of worlds $w \in W$ as objects, programs, and/or data. What happens when we apply one object to another (e.g. a rock to a nut; a puncher to some tape; a program to an input; a predicate to a datum)? On the one hand, we obtain an output (a nut that is broken; strip of perforated tape; a return value; a truth-value). Yet, on its own, an output is meaningless. What makes the execution of some action *a computation* is not the raw output, but the concept that this output signifies. That is, we apply w_1 to w_2 not to obtain some w_3, but to obtain

[8] There is some interest in non-commutative multiplicative conjunction, but this does not change the basic point.

some meaning that w_3 signifies. A raw output like '42' tells us nothing; it is only significant relative to the meaning we give it.

The output of a computation, rather than a mere action, is a *concept*.

As is standard, we can interpret concepts extensionally as sets of data. So, when $\bullet$ maps $W \times W$ to $\mathcal{P}(W)$ we can read this as follows: $\bullet$ takes two objects and applies one to the other to obtain a concept.

By this reading, when we write $\lambda p.\mathtt{t}$ or $\forall p.\, \phi$, p quantifies over concepts — not over data. Data is certainly there, and resides in W, but when we compute on W this returns us to the world of concepts. It is certainly possible to envisage frames in which $w_1 \bullet w_2$ is always a singleton $\{w_3\}$ — but this is just a very special case (and our completeness proofs do not generate such frames).

The fact that $\bullet$ maps to $\mathcal{P}(W)$ is a key point of the semantics in this paper. It turns out that this is sufficient to unify logic and computation, as we have seen.

Relevance logic. The notion of a function from $W \times W$ to $\mathcal{P}(W)$ does appear in the form of a ternary relation R on W, when giving denotations to *relevant* implication in *relevance logic* [5], and to implication in other provability and substructural logics such as *independence logic* [22]. For example, the clause for $\rhd$ in Definition 2.13 is just like the clause for relevant implication $\rightarrow$ in [5, §3.7, p.69].

However, these logics impose extra structure on R; see for example conditions (1) to (4) in [5, §3.7, p.68]. In the notation of this paper, $\bullet$ for relevance logic and other substructural logics models a form of logical conjunction and has structural properties like associativity and commutativity. In our frames $\bullet$ models function application, which does not have these structural properties.

H and Henkin models for higher-order logic. Another feature of frames is the set $H \subseteq \mathcal{P}(W)$. We mentioned in Remark 2.3 that we use H to restrict quantification and 'cut down the size' of powersets so as to obtain completeness. This idea is standard from Henkin semantics for higher-order logics.

Here, two classes of frame are particularly interesting: **full** frames in which $H = \mathcal{P}(W)$ (which we return to below), and **faithful** frames in which the denotation of every possible sentence is in H (see [20, Section 4.3]). Full frames are simple, and may be represented as a pair $F^{\text{full}} = (W, \bullet)$, but they inherit an overwhelming richness of structure from the full powerset. Henkin models are simpler 'first order' [20, Corollary 3.6] — and therefore completely axiomatisable — approximations to the full models.

Henkin semantics for higher-order logic are actually unsound without the assumption of faithfulness. We do not impose a general condition that models must be faithful because we built the models in the general case without committing to one specific logic. Once we fix a logic, conditions analogous to faithfulness begin to appear. See for example Theorems 2.27 and 3.8, and Corollary 4.15.

Investigating the properties of full models is possible further work.

5.3 Summary, and Further Work

We have presented models in which logic and computation have equal standing. They combine, among other things, the expressive power of untyped λ-calculus

and quantificational logic, in a single package. This has allowed us to give interesting answers to two specific questions:

Q. What is the negation of a λ-term?
A. Its sets complement.

Q. What logical connective corresponds to functional (λ) abstraction?
A. $\curlywedge$ from Definition 2.19.

There are many questions to which we do not have answers.

The logic $\mathcal{L}$ is very expressive; interesting things can be expressed other than the λ-calculus, including encodings of first-order logic and simple types, and also less standard constructs. We note in particular matching (Definition 2.19) as a 'dual' to λ. What other programming constructs do the semantics and the logic $\mathcal{L}$ invite?

We can take inspiration from modal logic, and note how different conditions on accessibility in Kripke models match up with different systems of model logic [13]. It is very interesting to imagine that conditions on H and $\bullet$ might match up with systems of λ-reduction and equality.

Can we tweak the frames so that Theorem 2.27 becomes provable for a reduction relation that does not include (η), or perhaps does include its converse (yes, but there is no space here for details). λ-calculus embeds in $\mathcal{L}$, so can a sequent system be given for $\mathcal{L}$ extending the sequent system for λ-reduction of [6]?

Note that logic-programming and the Curry-Howard correspondence both combine logic and computation, where computation resides in proof-search and proof-normalisation respectively.

Where does our combination of logic and computation fit into this picture, if at all?

We can only speculate on applications of all this.

Models can be computationally very useful. For instance, we may falsify a predicate by building a model that does not satisfy it. Our models might have something specific to offer here, because they are fairly elementary to construct and the tie-in to the languages $\mathcal{L}_\lambda$ and $\mathcal{L}$ is very tight, with completeness results for arbitrary theories (see Definition 4.1 and Corollary 4.14).

We have already touched on possible applications to language design; we might use the models and the logic to design new language constructs. We note in passing that the models have a built-in notion of location, given by reading $w \in W$ as a 'world'. It is not entirely implausible that this might be useful to give semantics to languages with subtle constructs reflecting that not all computation takes place on a single thread. To illustrate what we have in mind, consider a simple 'if... then... else' λ-term $\lambda p.\psi\left\{\begin{smallmatrix}\mathtt{t}_1\\ \mathtt{t}_2\end{smallmatrix}\right.$ which is such that $\lambda p.\psi\left\{\begin{smallmatrix}\mathtt{t}_1\\ \mathtt{t}_2\end{smallmatrix}\right.\cdot\mathtt{s}$ reduces to $\mathtt{t}_1[p::=\mathtt{s}]$ if ψ, and to $\mathtt{t}_2[p::=\mathtt{s}]$ otherwise. But 'where' should ψ be checked? Should it be checked at the world where the function resides, where the argument resides, or where the result resides? Translations into our logic reflect these possibilities:

$$\begin{array}{ll} \text{At the function:} & \forall p.\,\big((\psi \to (p \rhd \phi_1)) \wedge (\neg\psi \to (p \rhd \phi_2))\big) \\ \text{At the argument:} & \forall p.\,\big(((p \wedge \psi) \rhd \phi_1) \wedge ((p \wedge \neg\psi) \rhd \phi_2)\big) \\ \text{At the result:} & \forall p.\,\big(p \rhd ((\psi \to \phi_1) \wedge (\neg\psi \to \phi_2))\big) \end{array}$$

Note that p may be free in ψ.

We conclude the paper with a hypothesis. Consider the full frames where $H = \mathcal{P}(W)$ and using the translation τ of Definition 2.25 consider the relation $\twoheadrightarrow_2$ defined such that $\mathtt{t} \twoheadrightarrow_2 \mathtt{s}$ when $[\![\mathtt{t}^\tau \to \mathtt{s}^\tau]\!]^v = W$ for any valuation v on any full frame. Our hypothesis is this: there are $\mathtt{t}$ and $\mathtt{s}$ such that $\mathtt{t} \twoheadrightarrow_2 \mathtt{s}$ but $\mathtt{t} \not\twoheadrightarrow \mathtt{s}$; furthermore, $\mathtt{t} \twoheadrightarrow_2 \mathtt{s}$ is 'true' in the sense that $\mathtt{t}$ intuitively does compute to $\mathtt{s}$. In other words, we hypothesise that the intuitive concept of computation is captured by the F^{full}, just like our intuitive concept of natural number is captured by the standard model $\mathcal{N}$. We suggest that λ-calculi and axiomatisations of computation are actually first order implementations of $\twoheadrightarrow_2$.

References

1. Benzmüller, C., Brown, C.E., Kohlhase, M.: Higher-order semantics and extensionality. Journal of Symbolic Logic 69, 1027–1088 (2004)
2. Dekkers, W., Bunder, M., Barendregt, H.: Completeness of the propositions-as-types interpretation of intuitionistic logic into illative combinatory logic. Journal of Symbolic Logic 3, 869–890 (1998)
3. Dekkers, W., Bunder, M., Barendregt, H.: Completeness of two systems of illative combinatory logic for first-order propositional and predicate calculus. Archive für Mathematische Logik 37, 327–341 (1998)
4. Dowek, G., Hardin, T., Kirchner, C.: Theorem proving modulo. Journal of Automated Reasoning 31(1), 33–72 (2003)
5. Michael Dunn, J., Restall, G.: Relevance logic. In: Gabbay, D.M. (ed.) Handbook of Philosophical Logic, 2nd edn., vol. 6, pp. 1–136. Kluwer Academic Publishers, Dordrecht (2002)
6. Gabbay, M., Gabbay, M.J.: Term sequent logic. In: Proceedings of the 17th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2008). Electronic Notes in Theoretical Computer Science, vol. 246, pp. 87–106 (August 2009)
7. Werner, B., Dowek, G.: Arithmetic as a theory modulo. In: Proceedings of RTA, pp. 423–437 (2005)
8. Girard, J.-Y.: Linear logic. Theoretical Computer Science 50, 1–102 (1987)
9. Girard, J.-Y.: Linear logic: its syntax and semantics. In: Proceedings of the Workshop on Advances in Linear Logic, pp. 1–42. Cambridge University Press, New York (1995)
10. Girard, J.-Y., Taylor, P., Lafont, Y.: Proofs and types. Cambridge University Press, Cambridge (1989)
11. Goldblatt, R.: Logics of Time and Computation, 2nd edn., Center for the Study of Language and Information. CSLI Lecture Notes, vol. 7 (1992)
12. Henkin, L.: Completeness in the theory of types. Journal of Symbolic Logic 15, 81–91 (1950)
13. Hughes, G.E., Cresswell, M.J.: A New Introduction to Modal Logic. Routledge, New York (1996)

14. Jay, C.B., Ghani, N.: The virtues of eta-expansion. Journal of Functional Programming 5(2), 135–154 (1995)
15. Kleene, S.C., Rosser, J.B.: The inconsistency of certain formal logics. Annals of Mathematics 36(3) (1935)
16. Meyer, R.K., Bunder, M.W., Powers, L.: Implementing the 'fool's model' of combinatory logic. Journal of Automated Reasoning 7(4), 597–630 (1991)
17. O'Hearn, P., Pym, D.: The logic of bunched implications. Bulletin of Symbolic Logic 5(2), 215–244 (1999)
18. Plotkin, G.D.: The lambda-calculus is omega-incomplete. Journal of Symbolic Logic 39(2), 313–317 (1974)
19. Selinger, P.: Order-incompleteness and finite lambda reduction models. Theoretical Computer Science 309(1), 43–63 (2003)
20. Shapiro, S.: Foundations without foundationalism: a case for second-order logic. Oxford logic guides, vol. 17. Oxford University Press, Oxford (2000)
21. van Benthem, J.: Higher-order logic. In: Handbook of Philosophical Logic, 2nd edn., vol. 1, pp. 189–244. Kluwer, Dordrecht (2001)
22. Visser, A.: An overview of interpretability logic. In: Proceedings of the 1988 Heyting Conference on Mathematical Logic, pp. 307–359. Plenum Press, New York (1997)

Label-Free Proof Systems for Intuitionistic Modal Logic IS5

Didier Galmiche and Yakoub Salhi

LORIA – UHP Nancy 1
Campus Scientifique, BP 239
54 506 Vandœuvre-lès-Nancy, France

Abstract. In this paper we propose proof systems without labels for the intuitionistic modal logic IS5 that are based on a new multi-contextual sequent structure appropriate to deal with such a logic. We first give a label-free natural deduction system and thus derive natural deduction systems for the classical modal logic S5 and also for an intermediate logic IM5. Then we define a label-free sequent calculus for IS5 and prove its soundness and completeness. The study of this calculus leads to a decision procedure for IS5 and thus to an alternative syntactic proof of its decidability.

1 Introduction

Intuitionistic modal logics have important applications in computer science, for instance for the formal verification of computer hardware [7] and for the definition of programming languages [6,10]. Here, we focus on the intuitionistic modal logic IS5, introduced by Prior [13] and initially named MIPQ, that is the intuitionistic version of the modal logic S5. It satisfies the requirements given in [16] for the correct intuitionistic analogues of the modal logics. An algebraic semantics for IS5 has been introduced in [3] and the finite model property w.r.t. this semantics and consequently the decidability of this logic have been proved [11,15]. Moreover a translation of this logic into the monadic fragment of the intuitionistic predicate logic has been defined [4] and relations between some extensions of IS5 and intermediate predicate logics have been investigated [12]. In addition a Kripke semantics for IS5 was defined using frames where the accessibility relation is reflexive, transitive and symmetric [16]. As it is an equivalence relation, there exists an equivalent semantics with frames without accessibility relation like in the case of classical modal logic S5.

Here we mainly focus on proof theory for IS5 and on the design of new proof systems without labels for this logic. A Gentzen calculus was proposed in [11], but it does not satisfy the cut-elimination property. A natural deduction and cut-free Gentzen systems for IS5 have been proposed in [16], but a key point is that they are not considered as syntactically pure because of the presence of labels and relations between them corresponding to semantic information. In fact, they were introduced in order to support accessibility relations with arbitrary properties. There exist labelled systems without relations between labels

E.M. Clarke and A. Voronkov (Eds.): LPAR-16, LNAI 6355, pp. 255–271, 2010.

but for the fragment without $\bot$ and $\vee$ [10]. Moreover a hybrid version of IS5 has been introduced in [5,9] in order to reason about places with assertions of the form $A@p$ meaning that A is true at p. We observe that, by restricting the natural deduction system for this hybrid version, we can obtain a labelled natural deduction system for IS5.

In this paper, we aim at studying proof systems without labels for IS5. Then a first contribution is a label-free natural deduction system, called $\mathsf{ND}_{\mathsf{IS5}}$. It is based on a new sequent structure, called MC-sequent, that is multi-contextual and without labels, allowing the distribution of hypotheses in a multi-contextual environment. Compared to the hypersequent structure [1] that is adapted to classical logic the MC-sequent structure is more appropriate to deal with intuitionistic and modal operators. From this system we can deduce label-free natural deduction systems for S5 but also for IM5 [8] that is an intermediate logic between IS5 and S5. These natural deduction systems, without labels, illustrates the appropriateness of the MC-sequent structure for logics defined over IS5.

To complete these results another contribution is the definition of a sequent system for IS5 that is based on the MC-sequent structure and called $\mathsf{G}_{\mathsf{IS5}}$. Its soundness and completeness are proved from its equivalence with the natural deduction system. We also prove that $\mathsf{G}_{\mathsf{IS5}}$ satisfies the cut-elimination property. Moreover from the subformula property satisfied by the cut-free derivations, we introduce a notion of redundancy so that any valid MC-sequent has an irredundant derivation. Therefore we provide a new decision procedure for IS5 and thus obtain an alternative proof of the decidability of IS5 from our label-free sequent calculus.

2 The Intuitionistic Modal Logic IS5

The language of IS5 is obtained from the one of propositional intuitionistic logic IPL by adding the unary operators $\Box$ and $\Diamond$. Let Prop be a countably set of propositional symbols. We use $p, q, r, \ldots$ to range over Prop. The formulas of IS5 are given by the grammar:

$$\mathcal{F} ::= p \mid \bot \mid \mathcal{F} \wedge \mathcal{F} \mid \mathcal{F} \vee \mathcal{F} \mid \mathcal{F} \supset \mathcal{F} \mid \Box\mathcal{F} \mid \Diamond\mathcal{F}$$

The negation is defined by $\neg A \triangleq A \supset \bot$. A Hilbert axiomatic system for IS5 is given in Figure 1 (see [16]).

Note that the interdefinability between $\Box$ and $\Diamond$ given by $\Diamond A \triangleq \neg\Box\neg A$ breaks down in intuitionistic modal logics. That is similar to the fact that $\forall$ and $\exists$ are independent in intuitionistic first-order logic.

Definition 1. *A* Kripke model *is a tuple* $(W, \leqslant, \{D_w\}_{w\in W}, \{V_w\}_{w\in W})$ *where*
- W is a non-empty set (of 'worlds') partially ordered by $\leqslant$;
- for each $w \in W$, D_w is a non-empty set such that $w \leqslant w'$ implies $D_w \subseteq D_{w'}$ and
- for each $w \in W$, V_w is a function that assigns to each $p \in$ Prop a subset of D_w such that $w \leqslant w'$ implies $V_w(p) \subseteq V_{w'}(p)$.

0) All substitution instances of theorems of IPL.
1) $\Box(A \supset B) \supset (\Box A \supset \Box B)$.
2) $\Box(A \supset B) \supset (\Diamond A \supset \Diamond B)$.
3) $\Diamond \bot \supset \bot$.
4) $\Diamond(A \vee B) \supset (\Diamond A \vee \Diamond B)$.
5) $(\Diamond A \supset \Box B) \supset \Box(A \supset B)$.
6) $(\Box A \supset A) \wedge (A \supset \Diamond A)$.
7) $(\Diamond \Box A \supset \Box A) \wedge (\Diamond A \supset \Box \Diamond A)$.

$$\frac{A \supset B \qquad A}{B}\ [MP] \qquad\qquad \frac{A}{\Box A}\ [Nec]$$

Fig. 1. An Axiomatization of IS5

Definition 2. *Let $\mathcal{M} = (W, \leqslant, \{D_w\}_{w \in W}, \{V_w\}_{w \in W})$ be a Kripke model, $w \in W$, $d \in D_w$ and A be a formula, we define $\mathcal{M}, w, d \vDash A$ inductively as follows:*

- $\mathcal{M}, w, d \vDash p$ *iff* $d \in V_w(p)$;
- $\mathcal{M}, w, d \vDash \bot$ *never;*
- $\mathcal{M}, w, d \vDash A \wedge B$ *iff* $\mathcal{M}, w, d \vDash A$ *and* $\mathcal{M}, w, d \vDash B$;
- $\mathcal{M}, w, d \vDash A \vee B$ *iff* $\mathcal{M}, w, d \vDash A$ *or* $\mathcal{M}, w, d \vDash B$;
- $\mathcal{M}, w, d \vDash A \supset B$ *iff for all* $v \geqslant w$, $\mathcal{M}, v, d \vDash A$ *implies* $\mathcal{M}, v, d \vDash B$;
- $\mathcal{M}, w, d \vDash \Box A$ *iff for all* $v \geqslant w$, $e \in D_v$, $\mathcal{M}, v, e \vDash A$;
- $\mathcal{M}, w, d \vDash \Diamond A$ *iff there exists* $e \in D_w$ *such that* $\mathcal{M}, w, e \vDash A$.

A formula A is valid in $\mathcal{M} = (W, \leqslant, \{D_w\}_{w \in W}, \{V_w\}_{w \in W})$, written $\mathcal{M} \vDash A$, if and only if $\mathcal{M}, w, d \vDash A$ for every $w \in W$ and every $d \in D_w$. A formula is valid in IS5, written IS5 $\vDash A$, if and only if $\mathcal{M} \vDash A$ for every Kripke model $\mathcal{M}$.

IS5 has an equivalent Kripke semantics using frames where there is an accessibility relation which is reflexive, transitive and symmetric [16]. A simple way to prove the soundness and the completeness of the Kripke semantics defined here consists in the use of the translation of IS5 in the monadic fragment of the intuitionistic predicate logic [4]. This translation, denoted $(.)^*$, is defined by:

- $(\bot)^* = \bot$; $(p)^* = P(x)$;
- $(A \otimes B)^* = (A)^* \otimes (B)^*$, for $\otimes = \wedge, \vee, \supset$;
- $(\Box A)^* = \forall x.(A)^*$;
- $(\Diamond A)^* = \exists x.(A)^*$.

Proposition 3 (Monotonicity). *If we have $\mathcal{M}, w, d \vDash A$ and $w \leqslant w'$, then we have $\mathcal{M}, w', d \vDash A$.*

Proof. By structural induction on A.

3 Label-Free Natural Deduction for IS5

In this section, we introduce a natural deduction system for IS5, called $\mathsf{ND}_{\mathsf{IS5}}$, based on the definition of a particular sequent structure, called MC-sequent. The

soundness of this system is proved using Kripke semantics and its completeness is proved *via* the axiomatization given in Figure 1.

3.1 The MC-Sequent Structure

Let us recall that a context, denoted by the letters Γ and Δ, is a finite multiset of formulae and that a sequent is a structure of the form $\Gamma \vdash C$ where Γ is a context and C is a formula.

Definition 4 (MC-sequent). *An MC-sequent is a structure* $\Gamma_1; \ldots; \Gamma_k \vdash \Gamma \vdash C$ *where* $\{\Gamma_1, \ldots, \Gamma_k\}$ *is a finite multiset of contexts, called LL-context, and* $\Gamma \vdash C$ *is a sequent, called contextual conclusion.*

Let $G \vdash \Gamma \vdash C$ be a MC-sequent. If Γ is the empty context $\emptyset$, then we write $G \vdash \vdash C$ instead of $G \vdash \emptyset \vdash C$. Concerning the empty contexts in G (LL-context), they are not omitted.

The MC-sequent structure simply captures the fact that all the assumptions are relative and not absolute in the sense that if a formula is true in a given context, it is not necessary true in the other contexts. Intuitively, this can be seen as a spatial distribution of the assumptions. Indeed, each context represents a world in Kripke semantics with the fact that two different contexts do not necessarily represent two different worlds. This fact is highlighted by the corresponding formula of any MC-sequent, namely the MC-sequent $\Gamma_1; \ldots; \Gamma_k \vdash \Gamma \vdash C$ corresponds to the formula $(\Diamond(\bigwedge \Gamma_1) \wedge \ldots \wedge \Diamond(\bigwedge \Gamma_k)) \supset ((\bigwedge \Gamma) \supset C)$. We use the notation $\bigwedge \Gamma$ as a shorthand for $A_1 \wedge \ldots \wedge A_k$ when $\Gamma = A_1, \ldots, A_k$. If Γ is empty, we identify $\bigwedge \Gamma$ with $\top$.

This structure is similar to the hypersequent structure that is a multiset of sequents, called components, separated by a symbol denoting disjunction [1], in the sense that it is a multi-contextual structure. Since IS5 satisfies the disjunction property, namely if $A \vee B$ is a theorem then A is a theorem or B is a theorem, the hypersequent structure does not really enrich the sequent structure in this case and it appears that MC-sequent is more appropriate to deal with intuitionistic and modal operators. However, our approach is similar to the one in [14] where a hypersequent calculus for the classical modal logic S5 was introduced.

3.2 A Natural Deduction System for IS5

The rules of the natural deduction system $\mathsf{ND}_{\mathsf{IS5}}$ are given in Figure 2. Let us remark that if we consider the set of rules obtained from $\mathsf{ND}_{\mathsf{IS5}}$ by replacing any MC-sequent occurring in any rule by its contextual conclusion and by removing all the modal rules and the $[\vee_E^2]$ and $[\perp^2]$ rules then we obtain a set of rules corresponding to the known natural deduction system of IPL [17]. Hence, we obtain the following proposition:

Proposition 5. *If* A *is a substitution instance of a theorem of* IPL *then* $\vdash \vdash A$ *has a proof in* $\mathsf{ND}_{\mathsf{IS5}}$.

$$\frac{}{G \vdash \Gamma, A \vdash A}\ [Id] \qquad \frac{G \vdash \Gamma \vdash \bot}{G \vdash \Gamma \vdash A}\ [\bot^1] \qquad \frac{G; \Gamma \vdash \Gamma' \vdash \bot}{G; \Gamma' \vdash \Gamma \vdash A}\ [\bot^2]$$

$$\frac{G \vdash \Gamma \vdash A_i}{G \vdash \Gamma \vdash A_1 \vee A_2}\ [\vee^i_I] \qquad \frac{G \vdash \Gamma \vdash A \vee B \qquad G \vdash \Gamma, A \vdash C \qquad G \vdash \Gamma, B \vdash C}{G \vdash \Gamma \vdash C}\ [\vee^1_E]$$

$$\frac{G; \Gamma' \vdash \Gamma \vdash A \vee B \qquad G; \Gamma, A \vdash \Gamma' \vdash C \qquad G; \Gamma, B \vdash \Gamma' \vdash C}{G; \Gamma \vdash \Gamma' \vdash C}\ [\vee^2_E]$$

$$\frac{G \vdash \Gamma \vdash A \qquad G \vdash \Gamma \vdash B}{G \vdash \Gamma \vdash A \wedge B}\ [\wedge_I] \qquad \frac{G \vdash \Gamma \vdash A_1 \wedge A_2}{G \vdash \Gamma \vdash A_i}\ [\wedge^i_E]$$

$$\frac{G \vdash \Gamma, A \vdash B}{G \vdash \Gamma \vdash A \supset B}\ [\supset_I] \qquad \frac{G \vdash \Gamma \vdash A \supset B \qquad G \vdash \Gamma \vdash A}{G \vdash \Gamma \vdash B}\ [\supset_E]$$

$$\frac{G \vdash \Gamma \vdash A}{G \vdash \Gamma \vdash \Diamond A}\ [\Diamond^1_I] \qquad \frac{G; \Gamma' \vdash \Gamma \vdash A}{G; \Gamma \vdash \Gamma' \vdash \Diamond A}\ [\Diamond^2_I] \qquad \frac{G \vdash \Gamma \vdash \Diamond A \qquad G; A \vdash \Gamma \vdash C}{G \vdash \Gamma \vdash C}\ [\Diamond^1_E]$$

$$\frac{G; \Gamma' \vdash \Gamma \vdash \Diamond A \qquad G; \Gamma; A \vdash \Gamma' \vdash C}{G; \Gamma \vdash \Gamma' \vdash C}\ [\Diamond^2_E]$$

$$\frac{G; \Gamma \vdash \vdash A}{G \vdash \Gamma \vdash \Box A}\ [\Box_I] \qquad \frac{G \vdash \Gamma \vdash \Box A}{G \vdash \Gamma \vdash A}\ [\Box^1_E] \qquad \frac{G; \Gamma' \vdash \Gamma \vdash \Box A}{G; \Gamma \vdash \Gamma' \vdash A}\ [\Box^2_E]$$

Fig. 2. The Natural Deduction System $\mathsf{ND}_{\mathsf{IS5}}$

Let us comment now the modal rules of $\mathsf{ND}_{\mathsf{IS5}}$. The rule $[\Box_I]$ internalizes the fact that if a formula A is true in a given context without any assumption, then $\Box A$ is true in any context. $[\Box^1_E]$ internalizes the notion that if $\Box A$ is true in a given context then A is true in this context. $[\Box^2_E]$ internalizes the notion that if $\Box A$ is true in a given context then A is true in any other context. Indeed, this rule consists in an elimination of $\Box$ combined with a switch from the current context Γ to an other context Γ'. So $[\Box^1_E]$ and $[\Box^2_E]$ both internalize the fact that if $\Box A$ is true then A is true in any context. The rules of $\Diamond$ are dual to these of $\Box$. $[\Diamond^1_I]$ and $[\Diamond^2_I]$ both internalize the fact that if A is true in a given context then $\Diamond A$ is true in any context. $[\Diamond^1_E]$ and $[\Diamond^2_E]$ both internalize the fact that the assumption "A is true in a context without any other assumption" is equivalent to the assumption "$\Diamond A$ is true". This comes from the fact that if $\Diamond A$ is true then we may not necessary know in what context.

Let us illustrate our system by considering the formula $\Diamond(A \vee B) \supset (\Diamond A \vee \Diamond B)$. A proof of this formula in $\mathsf{ND}_{\mathsf{IS5}}$ is given by:

$$\dfrac{\dfrac{}{\vdash \Diamond(A \vee B) \vdash \Diamond(A \vee B)}\ [Id] \qquad \dfrac{\dfrac{}{\Diamond(A \vee B) \vdash A \vee B \vdash A \vee B}\ [Id] \qquad \mathcal{D}_1 \qquad \mathcal{D}_2}{A \vee B \vdash \Diamond(A \vee B) \vdash \Diamond A \vee \Diamond B}\ [\vee_E 2]}{\vdash \Diamond(A \vee B) \vdash \Diamond A \vee \Diamond B}\ [\Diamond_E 1]$$

with

$$\mathcal{D}_1 = \left\{ \dfrac{\dfrac{\dfrac{}{\Diamond(A \vee B) \vdash A \vee B, A \vdash A}\,[Id]}{A \vee B, A \vdash \Diamond(A \vee B) \vdash \Diamond A}\,[\Diamond_I 2]}{A \vee B, A \vdash \Diamond(A \vee B) \vdash \Diamond A \vee \Diamond B}\,[\vee_I 1] \right. \quad \mathcal{D}_2 = \left\{ \dfrac{\dfrac{\dfrac{}{\Diamond(A \vee B) \vdash A \vee B, B \vdash B}\,[Id]}{A \vee B, B \vdash \Diamond(A \vee B) \vdash \Diamond B}\,[\Diamond_I 2]}{A \vee B, B \vdash \Diamond(A \vee B) \vdash \Diamond A \vee \Diamond B}\,[\vee_I 2] \right.$$

Now we give a proof of the soundness of $\mathsf{ND}_{\mathsf{IS5}}$ using Kripke semantics. It consists in showing that for every rule, if its premise(s) are valid, then its conclusion is valid.

Theorem 6 (Soundness). *ND_{IS5} is sound, i.e., if a MC-sequent of $\mathsf{IS5}$ is provable in ND_{IS5} then it is valid in $\mathsf{IS5}$.*

Proof. Proceeding contrapositively, for every rule, we suppose that its conclusion is not valid and prove that one of its premises is not valid. Here, we only show the cases of $[\Box_I]$, $[\Box^2_E]$ and $[\Diamond^1_E]$.

- Case $[\Box_I]$. Let $\mathcal{M} = (W, \leqslant, \{D_w\}_{w \in W}, \{V_w\}_{w \in W})$ be a countermodel of $G \vdash \Gamma \vdash \Box A$. Then there exist w_0 in W and $d_0 \in D_{w_0}$ such that for all $\Gamma' \in G$, $w_0, d_0 \vDash \Diamond \bigwedge \Gamma'$ and $w_0, d_0 \vDash \bigwedge \Gamma$ and $w_0, d_0 \nvDash \Box A$.

From $w_0, d_0 \nvDash \Box A$, there exist $w_1 \in W$ and $d_1 \in D_{w_1}$ such that $w_0 \leqslant w_1$ and $w_1, d_1 \nvDash A$. Using Kripke monotonicity (Proposition 3), for all $\Gamma' \in G \cup \{\Gamma\}$, $w_1, d_1 \vDash \Diamond \bigwedge \Gamma'$. Thus, we deduce that $\mathcal{M}$ is a countermodel of $G; \Gamma \vdash \vdash A$.

- Case $[\Box^2_E]$. Let $\mathcal{M} = (W, \leqslant, \{D_w\}_{w \in W}, \{V_w\}_{w \in W})$ be a countermodel of $G; \Gamma \vdash \Gamma' \vdash A$. Then there exists w_0 in W and $d_0 \in D_{w_0}$ such that for all $\Gamma'' \in G \cup \{\Gamma\}$, $w_0, d_0 \vDash \Diamond \bigwedge \Gamma''$, $w_0, d_0 \vDash \bigwedge \Gamma'$ and $w_0, d_0 \nvDash A$.

Using $w_0, d_0 \vDash \Diamond \bigwedge \Gamma$, there exists d_1 in D_{w_0} such that $w_0, d_1 \vDash \bigwedge \Gamma$. Using $w_0, d_0 \vDash \bigwedge \Gamma'$, $w_0, d_1 \nvDash \Diamond \bigwedge \Gamma'$ holds. Using $w_0, d_0 \nvDash A$, $w_0, d_1 \nvDash \Box A$ holds. Thus, we deduce that $\mathcal{M}$ is a countermodel of $G; \Gamma' \vdash \Gamma \vdash \Box A$.

- Case $[\Diamond^1_E]$. Let $\mathcal{M} = (W, \leqslant, \{D_w\}_{w \in W}, \{V_w\}_{w \in W})$ be a countermodel of $G \vdash \Gamma \vdash C$. Then there exist w_0 in W and $d_0 \in D_{w_0}$ such that for all $\Gamma' \in G$, $w_0, d_0 \vDash \Diamond \bigwedge \Gamma'$ and $w_0, d_0 \vDash \bigwedge \Gamma$ and $w_0, d_0 \nvDash C$.

If $w_0, d_0 \nvDash \Diamond A$ then $\mathcal{M}$ is a countermodel of $G \vdash \Gamma \vdash \Diamond A$. Otherwise, $w_0, d_0 \vDash \Diamond A$ and then $\mathcal{M}$ is a countermodel of $G; A \vdash \Gamma \vdash C$.

Proposition 7. *The following MC-sequents are provable in ND_{IS5}*

1) $\vdash \vdash \Box(A \supset B) \supset (\Box A \supset \Box B)$ *2)* $\vdash \vdash \Diamond \bot \supset \bot$
3) $\vdash \vdash \Box(A \supset B) \supset (\Diamond A \supset \Diamond B)$ *4)* $\vdash \vdash \Diamond(A \vee B) \supset (\Diamond A \vee \Diamond B)$
5) $\vdash \vdash (\Diamond A \supset \Box B) \supset \Box(A \supset B)$ *6)* $\vdash \vdash (\Box A \supset A) \wedge (A \supset \Diamond A)$
7) $\vdash \vdash (\Diamond \Box A \supset \Box A) \wedge (\Diamond A \supset \Box \Diamond A)$

Proof. For 4) the proof is given as example before.

- For 1) we have

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\emptyset \vdash \Box(A \supset B), \Box A \vdash \Box(A \supset B)}[Id]}{\Box(A \supset B), \Box A \vdash A \supset B}[\Box_E^2] \quad \dfrac{\dfrac{}{\emptyset \vdash \Box(A \supset B), \Box A \vdash \Box A}[Id]}{\Box(A \supset B), \Box A \vdash A}[\Box_E^2]}{\Box(A \supset B), \Box A \vdash B}[\supset_E]}{\vdash \Box(A \supset B), \Box A \vdash \Box B}[\Box_I]}{\vdash \Box(A \supset B) \vdash \Box A \supset \Box B}[\supset_I]}{\vdash \vdash \Box(A \supset B) \supset (\Box A \supset \Box B)}[\supset_I]$$

- For 2) we have

$$\dfrac{\dfrac{\dfrac{}{\vdash \Diamond\bot \vdash \Diamond\bot}[Id] \quad \dfrac{\dfrac{}{\Diamond\bot \vdash \bot \vdash \bot}[Id]}{\bot \vdash \Diamond\bot \vdash \bot}[\bot^2]}{\vdash \Diamond\bot \vdash \bot}[\Diamond_E^1]}{\vdash \vdash \Diamond\bot \supset \bot}[\supset_I]$$

- For 3) we have

$$\dfrac{\dfrac{\dfrac{}{\vdash \Box(A \supset B), \Diamond A \vdash \Diamond A}[Id] \quad \dfrac{\dfrac{\dfrac{\dfrac{}{A \vdash \Box(A \supset B), \Diamond A \vdash \Box(A \supset B)}[Id]}{\Box(A \supset B), \Diamond A \vdash A \supset B}[\Box_E^2] \quad \dfrac{}{\Box(A \supset B), \Diamond A \vdash A \vdash A}[Id]}{\Box(A \supset B), \Diamond A \vdash A \vdash B}[\supset_E]}{A \vdash \Box(A \supset B), \Diamond A \vdash \Diamond B}[\Diamond_I^2]}{\vdash \Box(A \supset B), \Diamond A \vdash \Diamond B}[\Diamond_E^1]}{\vdash \Box(A \supset B) \vdash \Diamond A \supset \Diamond B}[\supset_I]}{\vdash \vdash \Box(A \supset B) \supset (\Diamond A \supset \Diamond B)}[\supset_I]$$

- For 5) we have

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \vdash \Diamond A \supset \Box B \vdash \Diamond A \supset \Box B}[Id] \quad \dfrac{\dfrac{}{\Diamond A \supset \Box B \vdash A \vdash A}[Id]}{A \vdash \Diamond A \supset \Box B \vdash \Diamond A}[\Diamond_I^2]}{A \vdash \Diamond A \supset \Box B \vdash \Box B}[\supset_E]}{\Diamond A \supset \Box B \vdash A \vdash B}[\Box_E^2]}{\Diamond A \supset \Box B \vdash \vdash A \supset B}[\supset_I]}{\vdash \Diamond A \supset \Box B \vdash \Box(A \supset B)}[\Box_I]}{\vdash \vdash (\Diamond A \supset \Box B) \supset \Box(A \supset B)}[\supset_I]$$

- For 6) we have

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \Box A \vdash \Box A}[Id]}{\vdash \Box A \vdash A}[\Box_E^1]}{\vdash \vdash \Box A \supset A}[\supset_I] \quad \dfrac{\dfrac{\dfrac{\dfrac{}{\Diamond A \supset \Box B \vdash A \vdash A}[Id]}{A \vdash \Diamond A \supset \Box B \vdash \Diamond A}[\Diamond_I^2]}{\vdash A \vdash A}[Id]}{\vdash A \vdash \Diamond A}[\Diamond_I^1]}{\vdash \vdash A \supset \Diamond A}[\supset_I]}{\vdash \vdash (\Box A \supset A) \wedge (A \supset \Diamond A)}[\wedge_I]$$

- For 7) we have

$$\dfrac{\dfrac{\dfrac{}{\vdash \Diamond\Box A \vdash \Diamond\Box A}[Id]}{\vdash \Diamond\Box A \vdash \Box A}[\Diamond^1_E]}{\vdash\vdash \Diamond\Box A \supset \Box A}[\supset_I] \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\emptyset; \Diamond\Box A \vdash \Box A \vdash \Box A}[Id]}{\Box A; \Diamond\Box A \vdash\vdash A}[\Box^2_E]}{\Box A \vdash \Diamond\Box A \vdash\vdash \Box A}[\Box_I]}{\vdash\vdash \Diamond\Box A \supset \Box A}[\supset_I] \qquad \dfrac{\dfrac{\dfrac{}{\emptyset; \vdash \Diamond A \vdash \Diamond A}[Id]}{\Diamond A \vdash\vdash \Diamond A}[\Diamond^1_E]}{\vdash \Diamond A \vdash \Box\Diamond A}[\Box_I] \qquad \dfrac{\dfrac{\dfrac{}{\emptyset; \Diamond A \vdash A \vdash A}[Id]}{\Diamond A; A \vdash\vdash \Diamond A}[\Diamond^2_I]}{\Diamond A \vdash\vdash \Diamond A}[\Diamond^2_E]}{\vdash\vdash \Diamond A \supset \Box\Diamond A}[\supset_I]}{\vdash\vdash (\Diamond\Box A \supset \Box A) \wedge (\Diamond A \supset \Box\Diamond A)}[\wedge_I]$$

Proposition 8. *The following properties are satisfied:*

1. *if $G \vdash \Gamma \vdash C$ has a proof in $\mathsf{ND}_{\mathsf{IS5}}$ then $G \vdash \Gamma, A \vdash C$ has a proof in $\mathsf{ND}_{\mathsf{IS5}}$;*
2. *if $G; \Gamma' \vdash \Gamma \vdash C$ has a proof in $\mathsf{ND}_{\mathsf{IS5}}$ then $G; \Gamma', A \vdash \Gamma \vdash C$ has a proof in $\mathsf{ND}_{\mathsf{IS5}}$;*
3. *if $G \vdash \Gamma \vdash C$ has a proof in $\mathsf{ND}_{\mathsf{IS5}}$ then $G; \Gamma' \vdash \Gamma \vdash C$ has a proof in $\mathsf{ND}_{\mathsf{IS5}}$.*

Proof. The first two properties are proved by mutual induction on the proof of their assumptions. The third one is simply proved by induction on the proof of its assumption.

Theorem 9. *If A is valid in $\mathsf{IS5}$ then $\vdash\vdash A$ has a proof in $\mathsf{ND}_{\mathsf{IS5}}$.*

Proof. We identify the validity in $\mathsf{IS5}$ through the axiomatization given in Figure 1 and consider an induction on the proof of A in this axiomatization.

If A is an axiom then $\vdash\vdash A$ is provable in $\mathsf{ND}_{\mathsf{IS5}}$ (Proposition 5 and Proposition 7).

Now, let us consider the last rule applied.

- If it is $[MP]$ then by applying the induction hypothesis, we have $\vdash\vdash A \supset B$ and $\vdash\vdash A$ have proofs in $\mathsf{ND}_{\mathsf{IS5}}$. Using the rule $[\supset_E]$, we show that $\vdash\vdash B$ has also a proof.
- Otherwise, if it is $[Nec]$ then by applying the induction hypothesis, $\vdash\vdash A$ has a proof in $\mathsf{ND}_{\mathsf{IS5}}$. Using Proposition 8, $\emptyset \vdash\vdash A$ has a proof in $\mathsf{ND}_{\mathsf{IS5}}$ and with the rule $[\Box_I]$, we show that $\vdash\vdash \Box A$ has also a proof.

The following two propositions allow us to state that for every MC-sequent, if its corresponding formula has a proof then it has also a proof.

Proposition 10. *$G \vdash \Gamma \vdash A \supset B$ has a proof if and only if $G \vdash \Gamma, A \vdash B$ has a proof.*

Proof. The *if part* comes from the rule $[\supset_I]$. For the *only if part*, using Proposition 8, $G \vdash \Gamma, A \vdash A \supset B$ has a proof. Then $G \vdash \Gamma, A \vdash B$ has a proof using the rule $[\supset_I]$ as follows:

$$\dfrac{G \vdash \Gamma, A \vdash A \supset B \quad \dfrac{}{G \vdash \Gamma, A \vdash A}[Id]}{G \vdash \Gamma, A \vdash B}[\supset_E]$$

Proposition 11. *The following properties are satisfied:*

1. *if* $G \vdash \Gamma, A \wedge B \vdash C$ *has a proof then* $G \vdash \Gamma, A, B \vdash C$ *has a proof;*
2. *if* $G; \Gamma, A \wedge B \vdash \Gamma' \vdash C$ *has a proof then* $G; \Gamma, A, B \vdash \Gamma' \vdash C$ *has a proof;*
3. *if* $G \vdash \Gamma, \Diamond A \vdash C$ *has a proof then* $G; A \vdash \Gamma \vdash C$ *has a proof;*
4. *if* $G; \Gamma, \Diamond A \vdash \Gamma' \vdash C$ *has a proof then* $G; \Gamma; A \vdash \Gamma' \vdash C$ *has a proof.*

Proof. By mutual induction on the proofs of their assumptions.

Theorem 12 (Completeness). $\mathsf{ND}_{\mathsf{IS5}}$ *is complete, i.e., if a MC-sequent of* IS5 *is valid then it has a proof in* $\mathsf{ND}_{\mathsf{IS5}}$.

Proof. Let $\mathcal{S} = \Gamma_1; \ldots; \Gamma_k \vdash \Gamma \vdash C$ be a valid MC-sequent. Then $\mathcal{F}_{\mathcal{S}} = (\Diamond \bigwedge \Gamma_1 \wedge \ldots \wedge \Diamond \bigwedge \Gamma_k \wedge \bigwedge \Gamma) \supset C$ is valid in IS5. Using Theorem 9, $\vdash \vdash \mathcal{F}_{\mathcal{S}}$ has a proof. Using Proposition 10, $\vdash \Diamond \bigwedge \Gamma_1 \wedge \ldots \wedge \Diamond \bigwedge \Gamma_k \wedge \bigwedge \Gamma \vdash C$ has a proof. Finally, using Proposition 11, we deduce that $\mathcal{S}$ has a proof.

Proposition 13 (Admissibility of cut rules).

- *If* $G \vdash \Gamma \vdash A$ *and* $G \vdash \Gamma, A \vdash B$ *have proofs, then* $G \vdash \Gamma \vdash B$ *has also a proof.*
- *If* $G; \Gamma \vdash \Gamma' \vdash A$ *and* $G; \Gamma', A \vdash \Gamma \vdash B$ *have proofs, then* $G; \Gamma' \vdash \Gamma \vdash B$ *has also a proof.*

Proof. By using Kripke semantics similarly to Theorem 6.

3.3 Natural Deduction Systems for S5 and IM5

In this section, we provide two natural deduction systems for S5 and one for IM5. We recall that IM5 is the logic obtained from IS5 by adding the axiom $\neg\Box\neg A \supset \Diamond A$ [8]. It is an intermediate logic in the sense that the set of formulas valid in this logic is between the sets of formulas valid in IS5 and S5 w.r.t. inclusion: $\mathsf{IS5} \subset \mathsf{IM5} \subset \mathsf{S5}$.

A natural deduction system for the classical modal logic S5 is obtained by replacing $[\bot^1]$ and $[\bot^2]$ in $\mathsf{ND}_{\mathsf{IS5}}$ by the following two rules:

$$\frac{G \vdash \Gamma, \neg A \vdash \bot}{G \vdash \Gamma \vdash A}\ [\bot_c^1] \qquad \frac{G; \Gamma, \neg A \vdash \Gamma' \vdash \bot}{G; \Gamma' \vdash \Gamma \vdash A}\ [\bot_c^2]$$

This comes from the fact that the addition of the axiom $A \vee \neg A$ to IS5 yields S5 [16]. As an example, we give a proof of $\neg\Diamond\neg A \supset \Box A$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\neg A \vdash \neg\Diamond\neg A \vdash \neg\Diamond\neg A}\,[Id] \quad \dfrac{\dfrac{}{\neg\Diamond\neg A \vdash \neg A \vdash \neg A}\,[Id]}{\neg A \vdash \neg\Diamond\neg A \vdash \Diamond\neg A}\,[\Diamond_I^2]}{\neg A \vdash \neg\Diamond\neg A \vdash \bot}\,[\supset_E]}{\neg\Diamond\neg A \vdash \vdash A}\,[\bot_c^2]}{\vdash \neg\Diamond\neg A \vdash \Box A}\,[\Box_I]}{\vdash \vdash \neg\Diamond\neg A \supset \Box A}\,[\supset_I]$$

Another natural deduction system for S5 is obtained by replacing the same rules by the following rules

$$\frac{G \vdash \Gamma, \Diamond\neg A \vdash \bot}{G \vdash \Gamma \vdash \Box A}\ [\bot_c^1] \qquad \frac{G; \Gamma, \Diamond\neg A \vdash \Gamma' \vdash \bot}{G; \Gamma' \vdash \Gamma \vdash \Box A}\ [\bot_c^2]$$

This rule internalizes the axiom $\neg\Diamond\neg A \supset \Box A$ and we know that the addition of this axiom to IS5 yields S5 [2].

Now, we consider the intermediate logic IM5 to show that the MC-sequent structure is appropriate to deal with some logics defined over IS5. Similarly to the case of S5, a natural deduction system for this logic is obtained by replacing $[\bot^1]$ and $[\bot^2]$ in $\mathsf{ND}_{\mathsf{IS5}}$ by the following two rules:

$$\frac{G \vdash \Gamma, \Box\neg A \vdash \bot}{G \vdash \Gamma \vdash \Diamond A}\ [\bot_w^1] \qquad \frac{G; \Gamma, \Box\neg A \vdash \Gamma' \vdash \bot}{G; \Gamma' \vdash \Gamma \vdash \Diamond A}\ [\bot_w^2]$$

The soundness of our systems for IM5 and S5 is obtained using the soundness of the rules of $\mathsf{ND}_{\mathsf{IS5}}$ and the axiomatizations of these logics. To prove completeness, we just have to use the axiomatization similarly to the proof of completeness of $\mathsf{ND}_{\mathsf{IS5}}$.

4 A Label-Free Sequent Calculus for IS5

In this section, we introduce a Gentzen calculus, called $\mathsf{G}_{\mathsf{IS5}}$, using the MC-sequent structure. Its soundness and completeness are proved using the natural deduction system $\mathsf{ND}_{\mathsf{IS5}}$. We prove that our calculus satisfies the key property of cut-elimination. Finally, from the subformula property satisfied by the cut-free proofs, we provide a new decision procedure for IS5. The rules of $\mathsf{G}_{\mathsf{IS5}}$ are given in Figure 3.

Note that $\mathsf{G}_{\mathsf{IS5}}$ is sound, complete and satisfies the cut-elimination property without the restriction on $[Id]$ that $p \in \mathsf{Prop}$. However, without this restriction, $\mathsf{G}_{\mathsf{IS5}}$ fails an important property necessary in our approach to prove the cut-elimination property, namely the *depth-preserving admissibility of contraction property.*

Proposition 14. *The MC-sequent $G \vdash \Gamma, A \vdash A$ is provable in $\mathsf{G}_{\mathsf{IS5}}$ for any A.*

Proof. By structural induction on A.

Weakening and contraction rules are not in $\mathsf{G}_{\mathsf{IS5}}$ because they have been absorbed into the rules and axioms. This approach is similar to the one used to obtain the calculus $G3i$ for the intuitionistic logic [17]. For instance, the choice of the axioms $G \vdash \Gamma, p \vdash p$, $G \vdash \Gamma, \bot \vdash C$ and $G; \Gamma', \bot \vdash \Gamma \vdash C$ instead of respectively $\vdash p \vdash p$, $\vdash \bot \vdash C$ and $\bot \vdash \vdash C$ allows us to absorb weakening.

$$\frac{}{G\vdash \Gamma, p\vdash p}\ [Id](p\in \mathsf{Prop}) \qquad \frac{}{G\vdash \Gamma,\bot\vdash C}\ [\bot^1] \qquad \frac{}{G;\Gamma',\bot\vdash \Gamma\vdash C}\ [\bot^2]$$

$$\frac{G\vdash \Gamma, A, B\vdash C}{G\vdash \Gamma, A\wedge B\vdash C}\ [\wedge_L] \qquad \frac{G;\Gamma', A, B\vdash \Gamma\vdash C}{G;\Gamma', A\wedge B\vdash \Gamma\vdash C}\ [\wedge_{LL}] \qquad \frac{G\vdash \Gamma\vdash A \qquad G\vdash \Gamma\vdash B}{G\vdash \Gamma\vdash A\wedge B}\ [\wedge_R]$$

$$\frac{G\vdash \Gamma, A\vdash C \qquad G\vdash \Gamma, B\vdash C}{G\vdash \Gamma, A\vee B\vdash C}\ [\vee_L] \qquad \frac{G;\Gamma', A\vdash \Gamma\vdash C \qquad G;\Gamma', B\vdash \Gamma\vdash C}{G;\Gamma', A\vee B\vdash \Gamma\vdash C}\ [\vee_{LL}]$$

$$\frac{G\vdash \Gamma\vdash A}{G\vdash \Gamma\vdash A\vee B}\ [\vee^1_R] \qquad \frac{G\vdash \Gamma\vdash B}{G\vdash \Gamma\vdash A\vee B}\ [\vee^2_R]$$

$$\frac{G\vdash \Gamma, A\supset B\vdash A \qquad G\vdash \Gamma, B\vdash C}{G\vdash \Gamma, A\supset B\vdash C}\ [\supset_L] \qquad \frac{G;\Gamma\vdash \Gamma', A\supset B\vdash A \qquad G;\Gamma', B\vdash \Gamma\vdash C}{G;\Gamma', A\supset B\vdash \Gamma\vdash C}\ [\supset_{LL}]$$

$$\frac{G\vdash \Gamma, A\vdash B}{G\vdash \Gamma\vdash A\supset B}\ [\supset_R]$$

$$\frac{G\vdash \Gamma, \Box A, A\vdash C}{G\vdash \Gamma, \Box A\vdash C}\ [\Box^1_L] \qquad \frac{G;\Gamma', A\vdash \Gamma, \Box A\vdash C}{G;\Gamma'\vdash \Gamma, \Box A\vdash C}\ [\Box^2_L]$$

$$\frac{G;\Gamma', \Box A\vdash \Gamma, A\vdash C}{G;\Gamma', \Box A\vdash \Gamma\vdash C}\ [\Box^1_{LL}] \qquad \frac{G;\Gamma', \Box A, A\vdash \Gamma\vdash C}{G;\Gamma', \Box A\vdash \Gamma\vdash C}\ [\Box^{2a}_{LL}] \qquad \frac{G;\Gamma'', A;\Gamma', \Box A\vdash \Gamma\vdash C}{G;\Gamma'';\Gamma', \Box A\vdash \Gamma\vdash C}\ [\Box^{2b}_{LL}]$$

$$\frac{G;\Gamma\vdash\vdash A}{G\vdash \Gamma\vdash \Box A}\ [\Box_R] \qquad \frac{G;A\vdash \Gamma\vdash C}{G\vdash \Gamma, \Diamond A\vdash C}\ [\Diamond_L] \qquad \frac{G;A;\Gamma'\vdash \Gamma\vdash C}{G;\Gamma', \Diamond A\vdash \Gamma\vdash C}\ [\Diamond_{LL}]$$

$$\frac{G\vdash \Gamma\vdash A}{G\vdash \Gamma\vdash \Diamond A}\ [\Diamond^1_R] \qquad \frac{G;\Gamma\vdash \Gamma'\vdash A}{G;\Gamma'\vdash \Gamma\vdash \Diamond A}\ [\Diamond^2_R]$$

$$\frac{G\vdash \Gamma\vdash A \qquad G\vdash \Gamma, A\vdash C}{G\vdash \Gamma\vdash C}\ [Cut^1] \qquad \frac{G;\Gamma\vdash \Gamma'\vdash A \qquad G;\Gamma', A\vdash \Gamma\vdash C}{G;\Gamma'\vdash \Gamma\vdash C}\ [Cut^2]$$

Fig. 3. The MC-sequent Calculus $\mathsf{G_{IS5}}$

Theorem 15 (Soundness). *If a MC-sequent is provable in* $\mathsf{G_{IS5}}$ *then it is provable in* $\mathsf{ND_{IS5}}$.

Proof. By induction on the proof of the MC-sequent in $\mathsf{G_{IS5}}$ using Proposition 13. We only have to consider the cases of the last rule of this proof. Here, we only develop the cases of $[\Box^1_L]$, $[\Box^2_L]$ and $[\Diamond_{LL}]$.

- Case of $[\Box^1_L]$: using the induction hypothesis $G\vdash \Gamma, \Box A, A\vdash C$ is provable in $\mathsf{ND_{IS5}}$. A proof of $G\vdash \Gamma, \Box A\vdash C$ in $\mathsf{ND_{IS5}}$ is given by:

$$\dfrac{\dfrac{\dfrac{}{G\vdash \Gamma, \Box A\vdash \Box A}\,[Id]}{G\vdash \Gamma, \Box A\vdash A}\,[\Box^1_E] \qquad G\vdash \Gamma, \Box A, A\vdash C}{G\vdash \Gamma, \Box A\vdash C}\,[Cut^1]$$

- Case of $[\Box^2_L]$: using the induction hypothesis $G;\Gamma', A\vdash \Gamma, \Box A\vdash C$ is provable in $\mathsf{ND_{IS5}}$. A proof of $G;\Gamma'\vdash \Gamma, \Box A\vdash C$ in $\mathsf{ND_{IS5}}$ is given by:

$$\dfrac{\dfrac{\dfrac{}{G;\Gamma' \vdash \Gamma, \Box A \vdash \Box A}\,[Id]}{G;\Gamma;\Box A \vdash \Gamma' \vdash A}\,[\Box^2_E] \qquad G;\Gamma', A \vdash \Gamma, \Box A \vdash C}{G;\Gamma' \vdash \Gamma, \Box A \vdash C}\,[Cut^2]$$

- Case of $[\Diamond_{LL}]$: using the induction hypothesis $G;A;\Gamma' \vdash \Gamma \vdash C$ is provable in $\mathsf{ND_{IS5}}$. Therefore, $G;A;\Gamma', \Diamond A \vdash \Gamma \vdash C$ is also provable in $\mathsf{ND_{IS5}}$. A proof of $G;\Gamma', \Diamond A \vdash \Gamma \vdash C$ in $\mathsf{ND_{IS5}}$ is given by:

$$\dfrac{\dfrac{}{G;\Gamma \vdash \Gamma', \Diamond A \vdash \Diamond A}\,[Id] \qquad G;A;\Gamma', \Diamond A \vdash \Gamma \vdash C}{G;\Gamma', \Diamond A \vdash \Gamma \vdash C}\,[\Diamond^2_E]$$

Theorem 16 (Completeness). *if a MC-sequent is provable in* $\mathsf{ND_{IS5}}$ *then it is provable in* $\mathsf{G_{IS5}}$.

Proof. We proceed by induction on the proof of the MC-sequent in $\mathsf{ND_{IS5}}$. We only have to consider the cases of the last rule applied in this proof. Here, we only develop the cases of $[\Box^2_E]$, $[\Diamond^1_E]$ and $[\Diamond^2_E]$.

- Case of $[\Box^2_E]$: using the induction hypothesis, $G;\Gamma' \vdash \Gamma \vdash \Box A$ is provable in $\mathsf{G_{IS5}}$. Then, a proof of $G;\Gamma \vdash \Gamma' \vdash A$ in $\mathsf{G_{IS5}}$ is given by:

$$\dfrac{G;\Gamma' \vdash \Gamma \vdash \Box A \qquad \dfrac{\dfrac{}{G;\Gamma, \Box A \vdash \Gamma', A \vdash A}\,[Id]}{G;\Gamma, \Box A \vdash \Gamma' \vdash A}\,[\Box^1_{LL}]}{G;\Gamma \vdash \Gamma' \vdash A}\,[Cut^2]$$

- Case of $[\Diamond^1_E]$: using the induction hypothesis, $G \vdash \Gamma \vdash \Diamond A$ and $G;A \vdash \Gamma \vdash C$ are provable in $\mathsf{G_{IS5}}$. Then a proof of $G \vdash \Gamma \vdash C$ is given by:

$$\dfrac{G \vdash \Gamma \vdash \Diamond A \qquad \dfrac{G;A \vdash \Gamma \vdash C}{G \vdash \Gamma, \Diamond A \vdash C}\,[\Diamond_L]}{G \vdash \Gamma \vdash C}\,[Cut^1]$$

- Case of $[\Diamond^2_E]$: using the induction hypothesis, $G;\Gamma' \vdash \Gamma \vdash \Diamond A$ and $G;A;\Gamma \vdash \Gamma' \vdash C$ are provable in $\mathsf{G_{IS5}}$. Then, a proof of $G;\Gamma \vdash \Gamma' \vdash C$ is given by:

$$\dfrac{G;\Gamma' \vdash \Gamma \vdash \Diamond A \qquad \dfrac{G;A;\Gamma \vdash \Gamma' \vdash C}{G;\Gamma, \Diamond A \vdash \Gamma' \vdash C}\,[\Diamond_{LL}]}{G;\Gamma \vdash \Gamma' \vdash C}\,[Cut^2]$$

Let us illustrate $\mathsf{G_{IS5}}$ by considering the MC-sequent $\vdash \vdash (\Diamond\Box A \supset \Box A) \wedge (\Diamond A \supset \Box\Diamond A)$. A proof of this sequent is given by:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Box A;\emptyset \vdash A \vdash A}\,[Id]}{\Box A;\emptyset \vdash \vdash A}\,[\Box^1_{LL}]}{\Diamond\Box A \vdash \vdash A}\,[\Diamond_{LL}]}{\vdash \Diamond\Box A \vdash \Box A}\,[\Box_R]}{\vdash \vdash \Diamond\Box A \supset \Box A}\,[\supset_R] \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\emptyset \vdash A \vdash A}\,[Id]}{A;\emptyset \vdash \vdash \Diamond A}\,[\Diamond^2_R]}{\Diamond A \vdash \vdash \Diamond A}\,[\Diamond_{LL}]}{\vdash \Diamond A \vdash \Box\Diamond A}\,[\Box_R]}{\vdash \vdash \Diamond A \supset \Box\Diamond A}\,[\supset_R]}{\vdash \vdash (\Diamond\Box A \supset \Box A) \wedge (\Diamond A \supset \Box\Diamond A)}\,[\wedge_R]$$

4.1 Depth-Preserving Admissibility of Weakening and Contraction

We write $\rhd_G \mathcal{S}$ if the MC-sequent $\mathcal{S}$ has a proof in a calculus G. Moreover, we write $\rhd^n_G \mathcal{S}$ if $\mathcal{S}$ has a proof in G of depth smaller or equal to n. Let us recall the notion of *depth-preserving admissibility*.

Definition 17. *A rule* $[R]$ *is said to be* admissible *for a calculus* G*, if for all instances* $\dfrac{H_1 \ \dots \ H_k}{C}\ [R]$ *of* $[R]$*, if for all* $i \in [1,k]$ $\rhd_G H_i$ *then* $\rhd_G C$.

A rule $[R]$ *is said to be* depth-preserving admissible *for* G*, if for all* n*, if for all* $i \in [1,k]$ $\rhd^n_G H_i$ *then* $\rhd^n_G C$.

We note $\mathsf{G}^-_{\mathsf{IS5}}$ the sequent calculus $\mathsf{G}_{\mathsf{IS5}}$ without cut rules. The following proposition corresponds to the depth-preserving admissibility property of weakening.

Proposition 18.

1. *If* $\rhd^n_{\mathsf{G}^-_{\mathsf{IS5}}} G \vdash \Gamma \vdash C$ *then* $\rhd^n_{\mathsf{G}^-_{\mathsf{IS5}}} G \vdash \Gamma, A \vdash C$.
2. *If* $\rhd^n_{\mathsf{G}^-_{\mathsf{IS5}}} G; \Gamma' \vdash \Gamma \vdash C$ *then* $\rhd^n_{\mathsf{G}^-_{\mathsf{IS5}}} G; \Gamma', A \vdash \Gamma \vdash C$.
3. *If* $\rhd^n_{\mathsf{G}^-_{\mathsf{IS5}}} G \vdash \Gamma \vdash C$ *then* $\rhd^n_{\mathsf{G}^-_{\mathsf{IS5}}} G; \Gamma' \vdash \Gamma \vdash C$.

Proof. 1. and 2. are proved by mutual induction on n and 3. by induction on n.

The following proposition is used to prove the depth-preserving admissibility of contraction. It is similar to the inversion lemma given in [17]. For some rules of $\mathsf{G}^-_{\mathsf{IS5}}$, if the conclusion has a proof of depth n, then some of its premises has proofs of depth smaller or equal to n.

Proposition 19.

1. a) *If* $\rhd^n_{\mathsf{G}^-_{\mathsf{IS5}}} G \vdash \Gamma, A \wedge B \vdash C$ *then* $\rhd^n_{\mathsf{G}^-_{\mathsf{IS5}}} G \vdash \Gamma, A, B \vdash C$.
   b) *If* $\rhd^n_{\mathsf{G}^-_{\mathsf{IS5}}} G; \Gamma', A \wedge B \vdash \Gamma \vdash C$ *then* $\rhd^n_{\mathsf{G}^-_{\mathsf{IS5}}} G; \Gamma', A, B \vdash \Gamma \vdash C$.
2. a) *If* $\rhd^n_{\mathsf{G}^-_{\mathsf{IS5}}} G \vdash \Gamma, A_1 \vee A_2 \vdash C$ *then* $\rhd^n_{S\mathsf{G}^-_{\mathsf{IS5}}} G \vdash \Gamma, A_i \vdash C$ *for* $i \in \{1,2\}$.
   b) *If* $\rhd^n_{\mathsf{G}^-_{\mathsf{IS5}}} G; \Gamma', A_1 \vee A_2 \vdash \Gamma \vdash C$ *then* $\rhd^n_{S\mathsf{G}^-_{\mathsf{IS5}}} G; \Gamma', A_i \vdash \Gamma \vdash C$ *for* $i \in \{1,2\}$.
3. *If* $\rhd^n_{\mathsf{G}^-_{\mathsf{IS5}}} G \vdash \Gamma \vdash A_1 \wedge A_2$ *then* $\rhd^n_{\mathsf{G}^-_{\mathsf{IS5}}} G \vdash \Gamma \vdash A_i$ *for* $i \in \{1,2\}$.
4. *If* $\rhd^n_{\mathsf{G}^-_{\mathsf{IS5}}} G \vdash \Gamma \vdash A \supset B$ *then* $\rhd^n_{\mathsf{G}^-_{\mathsf{IS5}}} G \vdash \Gamma, A \vdash B$.
5. a) *If* $\rhd^n_{\mathsf{G}^-_{\mathsf{IS5}}} G \vdash \Gamma, A \supset B \vdash C$ *then* $\rhd^n_{\mathsf{G}^-_{\mathsf{IS5}}} G \vdash \Gamma, B \vdash C$.
   b) *If* $\rhd^n_{\mathsf{G}^-_{\mathsf{IS5}}} G; \Gamma', A \supset B \vdash \Gamma \vdash C$ *then* $\rhd^n_{\mathsf{G}^-_{\mathsf{IS5}}} G; \Gamma, B \vdash \Gamma \vdash C$.
6. *If* $\rhd^n_{\mathsf{G}^-_{\mathsf{IS5}}} G \vdash \Gamma \vdash \Box A$ *then* $\rhd^n_{\mathsf{G}^-_{\mathsf{IS5}}} G; \Gamma \vdash \vdash A$.
7. a) *If* $\rhd^n_{\mathsf{G}^-_{\mathsf{IS5}}} G \vdash \Gamma, \Diamond A \vdash C$ *then* $\rhd^n_{\mathsf{G}^-_{\mathsf{IS5}}} G; A \vdash \Gamma \vdash C$.
   b) *If* $\rhd^n_{\mathsf{G}^-_{\mathsf{IS5}}} G; \Gamma', \Diamond A \vdash \Gamma \vdash C$ *then* $\rhd^n_{\mathsf{G}^-_{\mathsf{IS5}}} G; \Gamma'; A \vdash \Gamma \vdash C$.

Proof. 3., 4. and 6. are proved by induction on n. The other cases are proved by mutual induction. Here we only develop the proof of 6.

- If $n = 0$ then $G \vdash \Gamma \vdash \Box A$ is an instance of $[\bot^1]$ or $[\bot^2]$. Indeed, this sequent is not an instance of $[Id]$ because of $\Box A \notin \mathsf{Prop}$. Therefore, $\rhd^0_{\mathsf{G}^-_{\mathsf{IS5}}} G; \Gamma \vdash \vdash A$ holds.

- We assume that $\rhd^{n+1}_{\mathsf{G}^-_{\mathsf{IS5}}} G \vdash \Gamma \vdash \Box A$ by a proof $\mathcal{D}$. If $\Box A$ is not principal in the last rule applied in $\mathcal{D}$, then by applying induction hypothesis to the premise(s) and using the same rule, $\rhd^{n+1}_{\mathsf{G}^-_{\mathsf{IS5}}} G; \Gamma \vdash \vdash A$ holds. Otherwise, $\Box A$ is principal and $\mathcal{D}$ ends with

$$\frac{G; \Gamma \vdash \vdash A}{G \vdash \Gamma \vdash \Box A}\ [\Box_R]$$

By taking the immediate subdeduction of the premise, $\rhd^{n+1}_{\mathsf{G}^-_{\mathsf{IS5}}} G; \Gamma \vdash \vdash A$ holds.

The following proposition corresponds to the depth-preserving admissibility property of contraction.

Proposition 20.

1. *If* $\rhd^n_{\mathsf{G}^-_{\mathsf{IS5}}} G \vdash \Gamma, A, A \vdash C$ *then* $\rhd^n_{\mathsf{G}^-_{\mathsf{IS5}}} G \vdash \Gamma, A \vdash C$.
2. *If* $\rhd^n_{\mathsf{G}^-_{\mathsf{IS5}}} G; \Gamma', A, A \vdash \Gamma \vdash C$ *then* $\rhd^n_{\mathsf{G}^-_{\mathsf{IS5}}} G; \Gamma', A \vdash \Gamma \vdash C$.
3. *If* $\rhd^n_{\mathsf{G}^-_{\mathsf{IS5}}} G; \Gamma \vdash \Gamma \vdash C$ *then* $\rhd^n_{\mathsf{G}^-_{\mathsf{IS5}}} G \vdash \Gamma \vdash C$.
4. *If* $\rhd^n_{\mathsf{G}^-_{\mathsf{IS5}}} G; \Gamma'; \Gamma' \vdash \Gamma \vdash C$ *then* $\rhd^n_{\mathsf{G}^-_{\mathsf{IS5}}} G; \Gamma' \vdash \Gamma \vdash C$.

Proof. By mutual induction on n using Proposition 19.

4.2 Cut-Elimination in $\mathsf{G}_{\mathsf{IS5}}$

In order to prove the cut-elimination property, we use a variant of Gentzen's original proof of this property for classical and intuitionistic logic [17].

Theorem 21 (Cut-elimination). *The cut-elimination property holds for* $\mathsf{G}_{\mathsf{IS5}}$.

Proof. It consists in transforming the applications of cut rules to applications of cut rules on smaller formulae or applications of less height. For our calculus, because of the presence of two cut rules, the cut-elimination is proved by mutual induction. Here we only consider some cases.

If we have

$$\dfrac{\dfrac{G; \Gamma \vdash \vdash A}{G \vdash \Gamma \vdash \Box A}\ [\Box_R] \quad \dfrac{G \vdash \Gamma, \Box A, A \vdash C}{G \vdash \Gamma, \Box A \vdash C}\ [\Box^1_L]}{G \vdash \Gamma \vdash C}\ [Cut^1]$$

Then, to apply the induction hypothesis, we transform this derivation as follows:

$$\dfrac{\dfrac{G; \Gamma \vdash \vdash A}{G; \Gamma \vdash \Gamma \vdash A}\ [Prop\ 18] \quad \dfrac{\dfrac{G \vdash \Gamma \vdash \Box A}{G; \Gamma \vdash \Gamma, A \vdash \Box A}\ [Prop\ 18] \quad \dfrac{G \vdash \Gamma, \Box A, A \vdash C}{G; \Gamma \vdash \Gamma, \Box A, A \vdash C}\ [Prop\ 18]}{G; \Gamma \vdash \Gamma, A \vdash C}\ [Cut^1]}{\dfrac{G; \Gamma \vdash \Gamma \vdash C}{G \vdash \Gamma \vdash C}\ [Prop\ 20]}\ [Cut^1]$$

If we have

$$\dfrac{\dfrac{G';\Gamma';\Gamma\vdash\vdash A}{G';\Gamma'\vdash\Gamma\vdash\Box A}\,[\Box_R] \quad \dfrac{G';\Gamma',A\vdash\Gamma,\Box A\vdash C}{G';\Gamma'\vdash\Gamma,\Box A\vdash C}\,[\Box^2_L]}{G';\Gamma'\vdash\Gamma\vdash C}\,[cut^1]$$

Then, to apply the induction hypothesis, we transform this derivation as follows:

$$\dfrac{\dfrac{G';\Gamma';\Gamma\vdash\vdash A}{G';\Gamma;\Gamma'\vdash\Gamma'\vdash A}\,[Prop\ 18] \quad \dfrac{\dfrac{G';\Gamma'\vdash\Gamma\vdash\Box A}{G';\Gamma';\Gamma',A\vdash\Gamma\vdash\Box A}\,[Prop\ 18] \quad \dfrac{G';\Gamma',A\vdash\Gamma,\Box A\vdash C}{G';\Gamma';\Gamma',A\vdash\Gamma,\Box A\vdash C}\,[Prop\ 18]}{G;\Gamma';\Gamma',A\vdash\Gamma\vdash C}\,[cut^1]}{\dfrac{G;\Gamma';\Gamma'\vdash\Gamma\vdash C}{G;\Gamma'\vdash\Gamma\vdash C}\,[Prop\ 20]}\,[cut^2]$$

Corollary 22 (Subformula Property). *Any formula in any cut-free proof in* $\mathsf{G}_{\mathsf{IS5}}$ *of a MC-sequent* $\mathcal{S}$ *is a subformula of a formula appearing in* $\mathcal{S}$*.*

Proof. Each rule of $\mathsf{G}_{\mathsf{IS5}}$ except $[Cut^1]$ and $[Cut^2]$ has the property that every subformula of the formulae in the premise(s) is also a subformula of a formula in the conclusion.

5 A New Decision Procedure for IS5

In this section we provide a decision procedure for IS5 based on the use of $\mathsf{G}_{\mathsf{IS5}}$. The key point is the introduction of a notion of redundancy on the cut-free proof in $\mathsf{G}_{\mathsf{IS5}}$ satisfying the fact that any MC-sequent valid has an irredundant proof. And then using the subformula property, we prove that there is no infinite proof which is not irredundant. Finally, by an exhaustive search for an irredundant proof, we can decide any sequent.

We are interested in the size of proofs, i.e, the number of nodes. Previously, we proved that weakening and contraction are depth-preserving admissible for $\mathsf{G}^{-}_{\mathsf{IS5}}$. Weakening and contraction are also size-preserving admissible for $\mathsf{G}^{-}_{\mathsf{IS5}}$. We can prove this similarly to the proofs of Proposition 18 and Proposition 20. We use $set(\Gamma)$ to denote the set underlying the multiset Γ (the set of the formulas of Γ). We define a preorder, denoted $\lesssim$, on MC-sequent as follows: $\Gamma_1;\ldots;\Gamma_k\vdash\Gamma\vdash A \lesssim \Delta_1;\ldots;\Delta_l\vdash\Delta\vdash B$ iff $A=B$, $set(\Gamma)\subseteq set(\Delta)$ and for all $i\in[1,k]$ there exists $j\in[1,l]$ such that $set(\Gamma_i)\subseteq set(\Delta_j)$.

Proposition 23. *Let* $\mathcal{S}_1$ *and* $\mathcal{S}_2$ *be two MC-sequents. If* $\mathcal{S}_1\lesssim\mathcal{S}_2$ *then if* $\mathcal{S}_1$ *has a proof of size* n *then* $\mathcal{S}_2$ *has a proof of size smaller or equal to* n*.*

Proof. This follows directly from the size-preserving admissibility of weakening and contraction.

Definition 24. *A derivation is said to be* redundant *if it contains two MC-sequents* $\mathcal{S}_1$ *and* $\mathcal{S}_2$*, with* $\mathcal{S}_1$ *occurring strictly above* $\mathcal{S}_2$ *in the same branch, such that* $\mathcal{S}_1\lesssim\mathcal{S}_2$*. A derivation is irredundant if it is not redundant.*

Now, let us give our decision procedure for the MC-sequents in IS5.

Let $\mathcal{S}$ be a MC-sequent.

- Step 1. We start with the derivation containing only $\mathcal{S}$ which is the unique irredundant derivation of size 1. If this derivation is a proof then we return it. Otherwise we move to the next step.

- Step $i+1$. We construct the set of all the irredundant derivations of size $i+1$. If this set contains a proof of $\mathcal{S}$ then we return it. Otherwise if this set is empty then the decision algorithm fails, else we move to the next step.

There are only a finite number of possible rule applications. Thus, the set of the irredundant derivations of size $i+1$ is finite. Moreover, this set can be built in a finite time because the $\lesssim$ relation is decidable.

Theorem 25. IS5 *is decidable.*

Proof. Using Corollary 22, we know that there is no infinite irredundant derivation. Thus, we deduce that our algorithm terminates. Therefore, IS5 is decidable.

6 Conclusion and Perspectives

In this work, we introduce a new multi-contextual structure in order to deal with the intuitionistic modal logic IS5. An important contribution is the definition of a label-free natural deduction system for IS5 based on this structure. Then we deduce natural deduction systems for the modal logic S5 and the intermediate logic IM5. Another important contribution is the definition of a label-free sequent calculus satisfying the cut-elimination property. Then we define a new decision procedure from the subformula property satisfied by the cut-free derivation in this calculus. In further works, we will define natural deduction systems and sequent calculi for logics defined over IS5, for instance the ones in [12].

References

1. Avron, A.: The method of hypersequents in the proof theory of propositional non-classical logics. In: Hodges, W., Hyland, M., Steinhorn, C., Truss, J. (eds.) Logic: From Foundations to Applications, pp. 1–32. Oxford University Press, Oxford (1996)
2. Bezhanishvili, G., Zakharyaschev, M.: Logics over MIPC. In: Proceedings of Sequent Calculi and Kripke Semantics for Non-Classical Logics, pp. 86–95. Kyoto University (1997)
3. Bull, R.A.: A modal extension of intuitionistic logic. Notre Dame Journal of Formal Logic 6, 142–145 (1965)
4. Bull, R.A.: MIPC as the formalization of an intuitionistic concept of modality. Journal of Symbolic Logic 31, 609–616 (1966)
5. Chadha, R., Macedonio, D., Sassone, V.: A hybrid intuitionistic logic: Semantics and decidability. Journal of Logic and Computation 16(1), 27–59 (2006)
6. Davies, R., Pfenning, F.: A modal analysis of staged computation. Journal of ACM 48(3), 555–604 (2001)

7. Fairtlough, M., Mendler, M.: An intuitionistic modal logic with applications to the formal verification of hardware. In: Kleine Büning, H. (ed.) CSL 1995. LNCS, vol. 1092, pp. 354–368. Springer, Heidelberg (1996)
8. Font, J.M.: Modality and possibility in some intuitionistic modal logics. Notre Dame Journal of Formal Logic 27, 533–546 (1986)
9. Jia, L., Walker, D.: Modal proofs as distributed programs (extended abstract). In: Schmidt, D. (ed.) ESOP 2004. LNCS, vol. 2986, pp. 219–233. Springer, Heidelberg (2004)
10. Murphy VII, T., Crary, K., Harper, R., Pfenning, F.: A symmetric modal lambda calculus for distributed computing. In: Proceedings of the 19th IEEE Symposium on Logic in Computer Science (LICS), pp. 286–295. IEEE Press, Los Alamitos (2004)
11. Ono, H.: On some intuitionistic modal logics. Publications of the Research Institute for Mathematical Science 13, 55–67 (1977)
12. Ono, H., Suzuki, N.: Relations between intuitionistic modal logics and intermediate predicate logics. Reports on Mathematical Logic 22, 65–87 (1988)
13. Prior, A.: Time and Modality. Clarendon Press, Oxford (1957)
14. Restall, G.: Proofnets for S5: sequents and circuits for modal logic. In: Dimitracopoulos, C., Newelski, L., Normann, D. (eds.) Logic Colloquium 2005. Lecture Notes in Logic, vol. 28, pp. 151–172. Cambridge University Press, Cambridge (2007)
15. Fischer Servi, G.: The finite model property for MIPQ and some consequences. Notre Dame Journal of Formal Logic XIX, 687–692 (1978)
16. Simpson, A.: The proof theory and semantics of intuitionistic modal logic. PhD thesis, University of Edinburgh (1994)
17. Troelstra, A.S., Schwichtenberg, H.: Basic Proof Theory. Cambridge Tracts in Theoretical Computer Science, vol. 43. Cambridge University Press, Cambridge (1996)

An Intuitionistic Epistemic Logic for Sequential Consistency on Shared Memory

Yoichi Hirai

University of Tokyo, Dept. of Computer Science, 7-3-1 Hongo, Tokyo 113-0033, Japan
yh@lyon.is.s.u-tokyo.ac.jp

Abstract. In the celebrated Gödel Prize winning papers, Herlihy, Shavit, Saks and Zaharoglou gave topological characterization of waitfree computation. In this paper, we characterize waitfree communication logically. First, we give an intuitionistic epistemic logic $\mathbf{K}\vee$ for asynchronous communication. The semantics for the logic $\mathbf{K}\vee$ is an abstraction of Herlihy and Shavit's topological model. In the same way Kripke model for intuitionistic logic informally describes an agent increasing its knowledge over time, the semantics of $\mathbf{K}\vee$ describes multiple agents passing proofs around and developing their knowledge together. On top of the logic $\mathbf{K}\vee$, we give an axiom type that characterizes sequential consistency on shared memory. The advantage of intuitionistic logic over classical logic then becomes apparent as the axioms for sequential consistency are meaningless for classical logic because they are classical tautologies. The axioms are similar to the axiom type for prelinearity $(\varphi \supset \psi) \vee (\psi \supset \varphi)$. This similarity reflects the analogy between sequential consistency for shared memory scheduling and linearity for Kripke frames: both require total order on schedules or models. Finally, under sequential consistency, we give soundness and completeness between a set of logical formulas called waitfree assertions and a set of models called waitfree schedule models.

1 Introduction

Waitfree Computation. The main purpose of this paper is to characterize waitfree communication logically (Theorem 7) in a language as simple as possible. Waitfreedom [10] is a restriction on distributed programs over shared memory. It forbids any process to wait for another process. Some tasks can be solved by a well-chosen waitfree protocol while the others cannot.

For example, it is waitfreely impossible for each one of two processes to attain the input value of the other process. On the other hand, it is waitfreely possible for either one of two processes to attain the input value of the other process. A waitfree protocol that solves this task is:

- process p tells the memory m that φ holds, and then m replies back to p,
- process q tells the memory m that ψ holds, and then m replies back to q.

After this protocol finishes, either φ has been communicated from p to q or ψ has been communicated from q to p. In the logic $\mathbf{K}\vee$, this fact is represented by

E.M. Clarke and A. Voronkov (Eds.): LPAR-16, LNAI 6355, pp. 272–289, 2010.

a formula $(K_pK_{\mathsf{m}}K_p\varphi \wedge K_qK_{\mathsf{m}}K_q\psi) \supset (K_pK_q\psi \vee K_qK_p\varphi)$, which is deducible in $\mathbf{K}\vee$ with sequential consistency (Figure 2).

Herlihy and Shavit [11] characterized waitfree computation using simplicial topology (See Section 6). Using their characterization, Gafni and Koutsoupias [8] showed that it is undecidable whether a task is waitfreely solvable or not. In this paper we show that, when tasks are restricted to communication defined by a class of logical formulas we call waitfree assertions, it is decidable whether a task is waitfreely solvable or not (Subsection 4.1).

Sequential Consistency. The topological characterization by Herlihy and Shavit [11] implicitly assumes sequential consistency [17] for shared memory. Since we seek to use a simple language, we state sequential consistency explicitly in the language. We characterize sequential consistency with an axiom type $(K_{\mathsf{m}}\varphi \supset K_{\mathsf{m}}\psi) \vee (K_{\mathsf{m}}\psi \supset K_{\mathsf{m}}\varphi)$ in the logic $\mathbf{K}\vee$ for asynchronous computation. The axiom type informally states that for any two propositions φ and ψ, either one become known to the memory no later than the other. The axiom type is sound (Theorem 4) and strongly complete (Theorem 6) for a class of models called sequential models where memory states are temporally lined up in a total order.

Asynchronous Communication. We define an intuitionistic modal propositional logic that we call $\mathbf{K}\vee$ and show soundness (Theorem 4) and strong completeness (Theorem 5) for Kripke semantics. The syntax of $\mathbf{K}\vee$ is the same as that of the classical epistemic logic [13] although the informal reading of the epistemic modality K_a is new. In the classical epistemic logic, the formula $K_a\varphi$ informally reads "φ is valid in any possible worlds of agent a." while in $\mathbf{K}\vee$ the same formula informally reads "agent a has received a proof of φ." The semantics of $\mathbf{K}\vee$ is simple: it has only one function for each agent in addition to the Kripke model for intuitionistic propositional logic. We deliberately identify the partial order in Kripke frame with the temporal relation. Intuitionistic logic can be seen as a logic describing an agent whose knowledge increases over time. The logic $\mathbf{K}\vee$ can be seen as a logic describing multiple agents that asynchronously communicate with each other and increase their knowledge. Although $\mathbf{K}\vee$ deals with communication, the logic has only epistemic modalities so that it has simpler syntax than many other logics for communication.

There are other choices: there have been proposed a huge number of epistemic logics for communication [3–6, 9, 14, 18, 21, 22, 28] and a huge number of intuitionistic modal logics [1, 7, 20, 21, 23]. In both cases, when considered under Kripke semantics, the huge variety of logics comes from the diversity of relationships between two binary relations on the state space. In intuitionistic modal logic, the two relations are: (a) which state is prior to which state with regard to Kripke monotonicity and (b) the modality in which state refers to which state. In logics for communication, the two relations are: (a') which state is temporally prior to which state and (b') from which state to which state communication occurs.

The semantics of $\mathbf{K}\vee$ uses a binary relation and *functions* on possible worlds instead of additional binary relations. This choice dramatically limits the room for design choice. Also, we identify relations (a) with (a') and (b) with (b') in order to make the language of $\mathbf{K}\vee$ simpler.

Structure of Paper. Although this introduction so far is organized in the top-to-bottom order, the rest of this paper is in the opposite bottom-to-top order. Sections 2–4 respectively treat asynchronous computation in general, sequential consistency and waitfree communication.

2 Intuitionistic Epistemic Logic for Asynchronous Communication

2.1 Syntax

We fix a countably infinite set of propositional variables $PVar$ and a set of agents A. We use the meta-variables $P, Q, \ldots$ running over $PVar$ and $a, b, \ldots, p, q, \ldots$ running over A.

Definition 1. *We define a formula φ by the BNF:*

$$\varphi ::= \bot \mid P \mid (K_a\varphi) \mid (\varphi \vee \varphi) \mid (\varphi \wedge \varphi) \mid (\varphi \supset \varphi).$$

The unary operators connect more strongly than the binary operators. We sometimes omit the parentheses when no confusion occurs. We use $=$ for syntactic equality of formulas. The notation $(\neg\varphi)$ stands for $(\varphi \supset \bot)$. For a sequence of formulas $\Gamma = (\varphi_i)_{i\in I}$ or a set of formulas Γ, the notation $K_a\Gamma$ stands for the sequence $(K_a\varphi_i)_{i\in I}$ or the set $\{K_a\varphi \mid \varphi \in \Gamma\}$ respectively.

Definition 2. *We define the proof system of* $\mathbf{K}\vee$ *by Figure 7.*

For a set of formula Γ and a formula φ, notation $\Gamma \vdash \varphi$ denotes a relation where there is such a finite sequence Γ_0 that $\Gamma_0 \vdash \varphi$ is deducible and that Γ_0 contains only formulas in Γ.

2.2 Semantics

We define validity of a formula on a state in a model. A model is a Kripke model for propositional intuitionistic logic equipped with an additional mapping $f_a : W \to W$ for each agent $a \in A$ where W is the set of possible states. Informally[1], the function f_a represents the "view" of agent a. When the current state is $w \in W$, agent a sees that the current state is $f_a(w) \in W$, in other words, agent a knows everything valid in $f_a(w)$. Agent a also sees that agent b sees that the current state is $f_b(f_a(w)) \in W$ because we assume that all agents know the frame structure and the functions f_x explicitly or implicitly. This is in contrast to the classical epistemic logic, where an agent's view is represented by not a state but a set of states. This model is an abstraction of Herlihy and Shavit's model of waitfree computation [11]. See Section 6 for details.

[1] This account is informal in that we do not attempt to define the terms "view" and "current state".

$$\text{(ax)}\ \frac{}{\varphi \vdash \varphi} \qquad \text{(w)}\ \frac{\Gamma \vdash \varphi}{\psi, \Gamma \vdash \varphi} \qquad \text{(c)}\ \frac{\varphi, \varphi, \Gamma \vdash \varphi'}{\varphi, \Gamma \vdash \varphi'}$$

$$\text{(ex)}\ \frac{\Gamma, \varphi, \psi, \Gamma' \vdash \varphi'}{\Gamma, \psi, \varphi, \Gamma' \vdash \varphi'} \qquad \text{(}\wedge\text{-I)}\ \frac{\Gamma \vdash \varphi \qquad \Gamma' \vdash \psi}{\Gamma, \Gamma' \vdash \varphi \wedge \psi} \qquad \text{(}\vee\text{-I}_0\text{)}\ \frac{\Gamma \vdash \varphi}{\Gamma \vdash \varphi \vee \psi}$$

$$\text{(}\vee\text{-I}_1\text{)}\ \frac{\Gamma \vdash \varphi}{\Gamma \vdash \psi \vee \varphi}$$

$$\text{(}\wedge\text{-E}_0\text{)}\ \frac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \varphi} \qquad \text{(}\wedge\text{-E}_1\text{)}\ \frac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \psi}$$

$$\text{(}\vee\text{-E)}\ \frac{\Gamma \vdash \psi_0 \vee \psi_1 \qquad \Gamma, \psi_0 \vdash \varphi \qquad \Gamma, \psi_1 \vdash \varphi}{\Gamma \vdash \varphi}$$

$$\text{(}\supset\text{-I)}\ \frac{\varphi, \Gamma \vdash \psi}{\Gamma \vdash \varphi \supset \psi} \qquad \text{(}\supset\text{-E)}\ \frac{\Gamma \vdash \psi_0 \supset \psi_1 \qquad \Gamma \vdash \psi_0}{\Gamma \vdash \psi_1} \qquad \text{(}\bot\text{-E)}\ \frac{\Gamma \vdash \bot}{\Gamma \vdash \varphi} \qquad \text{(T)}\ \frac{}{K_a\varphi \vdash \varphi}$$

$$\text{(introspection)}\ \frac{}{K_a\varphi \vdash K_aK_a\varphi} \qquad \text{(nec)}\ \frac{\Gamma \vdash \varphi}{K_a\Gamma \vdash K_a\varphi}$$

$$\text{(}\vee K\text{)}\ \frac{}{K_a(\varphi \vee \psi) \vdash (K_a\varphi) \vee K_a\psi}$$

Fig. 1. Deduction rules of $\mathbf{K}\vee$

Definition 3. *A model $\langle W, \preceq, (f_a)_{a\in A}, \rho\rangle$ is a tuple of following things:*

1. *$\langle W, \preceq\rangle$ is a partial order,*
2. *$f_a\colon W \to W$ is a function satisfying all of the following conditions for any $w \in W$:*
   (a) (descending) $f_a(w) \preceq w$,
   (b) (idempotency) $f_a(f_a(w)) = f_a(w)$,
   (c) (monotonicity) $w \preceq v$ implies $f_a(w) \preceq f_a(v)$,
3. *$\rho\colon PVar \to \mathcal{P}(W)$ is a function such that each $\rho(P)$ is upward-closed with respect to $\preceq$, i.e., $w' \succeq w \in \rho(P)$ implies $w' \in \rho(P)$.*

With the informal account in mind, the conditions on f_a have rationales: descending condition says an agent a recognizes only truth, idempotency says an agent a recognizes that a recognizes something whenever the agent a recognizes that thing, and monotonicity says an agent a does not forget things once they recognized.

Definition 4. *We define the validity relation $\models$ of a model $\langle W, \preceq, (f_a)_{a\in A}, \rho\rangle$, a state $w \in W$ and a formula φ. Let us fix a model $M = \langle W, \preceq, f, \rho\rangle$ and abbreviate $M, w \models \varphi$ into $w \models \varphi$. The definition of $\models$ is inductive on the structure of φ.*

(Case $\varphi = \bot$) *$w \models \bot$ never holds.*
(Case $\varphi = P$) *$w \models P$ if and only if $w \in \rho(P)$.*
(Case $\varphi = K_a\psi$) *$w \models K_a\psi$ if and only if $f_a(w) \models \psi$.*

(Case $\varphi = \psi_0 \wedge \psi_1$) $w \models \psi_0 \wedge \psi_1$ *if and only if both* $w \models \psi_0$ *and* $w \models \psi_1$ *hold.*
(Case $\varphi = \psi_0 \vee \psi_1$) $w \models \psi_0 \vee \psi_1$ *if and only if either* $w \models \psi_0$ *or* $w \models \psi_1$ *holds.*
(Case $\varphi = \psi_0 \supset \psi_1$) $w \models \psi_0 \supset \psi_1$ *if and only if for any* $w' \in W$, $w' \succeq w$ *and* $w' \models \psi_0$ *imply* $w' \models \psi_1$.

Theorem 1 (Kripke monotonicity). $M, w \models \varphi$ *and* $w \preceq v$ *imply* $M, v \models \varphi$.

Proof. By simple structural induction on φ.

Notation 2. *For a model* M, *a state* w *of* M *and a set of formulas* Γ, *we write* $M, w \models \Gamma$ *when* $M, w \models \varphi$ *holds for any formula* $\varphi \in \Gamma$.

Notation 3. $\Gamma \models \varphi$ *stands for the relation of a set of a formula* Γ *and a formula* φ *where* $M, w \models \Gamma$ *implies* $M, w \models \varphi$ *for any model* M *and a state* $w \in M$.

2.3 Soundness

Theorem 4 (Soundness). $\Gamma \vdash \varphi$ *implies* $\Gamma \models \varphi$.

Proof. We prove soundness with induction on the definition of $\vdash$. We fix a model M and we abbreviate $M, w \models \varphi$ into $w \models \varphi$.

($\supset$-I). Assume $\Gamma, \varphi \models \psi$. Assume $w \models \Gamma$. Also assume that there is such a state w' in M that $w' \succeq w$ and $w' \models \varphi$ hold. By Lemma 1, $w' \models \Gamma$ holds. Since $\Gamma, \varphi \models \psi$, the relation $\Gamma, w' \models \psi$ holds.
($\supset$-E). Assume $\Gamma \models \varphi \supset \psi$ and $\Gamma \models \varphi$. By the second assumption, $w \models \varphi$ holds. The first assumption says $w \models \varphi \supset \psi$. Since $w \succeq w$, the relation $w \models \psi$ holds.
(T). By induction hypothesis, Lemma 1 and the descending property of f_a.
(introspection). By induction hypothesis and the idempotency of f_a.
(nec). Immediately by the induction hypothesis.
($\vee K_a$). Immediately by the induction hypothesis and the fact that the semantics of the modality K_a is defined in terms of a function.
(axiom)(weakening)(contraction)(exchange)
($\wedge$-I)($\vee$-I$_0$)($\vee$-I$_1$)($\vee$-E)($\wedge$-E$_0$)($\wedge$-E$_1$) Trivial.

2.4 Strong Completeness

We show strong completeness for $\mathbf{K}\vee$ with a canonical model construction as in [27, Ch. 2].

Definition 5. *A set of formulas* Γ *is saturated if and only if all of these conditions are satisfied:*

1. Γ *is deductively closed, i.e.,* $\Gamma \vdash \varphi \Rightarrow \varphi \in \Gamma$,
2. $\varphi \vee \psi \in \Gamma \Rightarrow \varphi \in \Gamma$ *or* $\psi \in \Gamma$,
3. $\Gamma \nvdash \bot$.

Proposition 1. *For any set Γ of formulas and two formulas φ, ψ, conditions $\Gamma \vdash \varphi$ and $\Gamma, \varphi \vdash \psi$ imply $\Gamma \vdash \psi$.*

Proof. By simple induction on the definition of $\Gamma, \varphi \vdash \psi$. Note that the induction goes without fixing Γ, φ or ψ.

Lemma 1 (Saturation lemma). *For a set of formulas Γ with $\Gamma \nvdash \varphi$, there exists a saturated set Γ^ω with $\Gamma^\omega \nvdash \varphi$ and $\Gamma \subseteq \Gamma^\omega$.*

Proof. We can enumerate all formulas in a sequence $(\varphi_i)_{i \in \mathbb{N}^+}$. We define Γ^i inductively on $i \in \mathbb{N}$:

(Case $i = 0$) $\Gamma^0 = \Gamma$,
(Case $i > 0$) if $\{\varphi_i\} \cup \Gamma^{i-1} \nvdash \varphi$, $\Gamma^i = \{\varphi_i\} \cup \Gamma^{i-1}$; otherwise, $\Gamma^i = \Gamma^{i-1} \cup \{\varphi_i \supset \varphi\}$.

Using these Γ^i, we define $\Gamma^\omega = \bigcup_{i \in \omega} \Gamma^i$.

Claim: $\Gamma^\omega \nvdash \varphi$. Seeking contradiction, assume $\Gamma^\omega \vdash \varphi$. Since only finite number of formulas in Γ are used to prove φ, there exists a minimal i with $\Gamma^i \vdash \varphi$. Since $\Gamma \nvdash \varphi$, $i \neq 0$. Either $\Gamma^i = \{\varphi_i\} \cup \Gamma^{i-1}$ or $\Gamma^i = \{\varphi_i \supset \varphi\} \cup \Gamma^{i-1}$. The first case is explicitly forbidden. In the second case, $\Gamma^{i-1}, \varphi_i \supset \varphi \vdash \varphi$ holds. That means $\Gamma^{i-1} \vdash (\varphi_i \supset \varphi) \supset \varphi$. Also, since we could not take the first case, $\Gamma^{i-1}, \varphi_i \vdash \varphi$ holds. That means $\Gamma^{i-1} \vdash \varphi_i \supset \varphi$. These combined with Proposition 1 prove $\Gamma^{i-1} \vdash \varphi$, which contradicts to the minimality of i.

Claim: Γ^ω is a saturated set. We check each condition for a saturated set (Definition 5).

1. Assume $\Gamma^\omega \vdash \psi$. There is $i \in \mathbb{N}^+$ with $\varphi_i = \psi$. We know that $\Gamma^{i-1} \cup \{\varphi_i\} \nvdash \varphi$. It means $\psi \in \Gamma^\omega$.
2. Assume $\psi_0 \vee \psi_1 \in \Gamma^\omega$. Seeking contradiction, assume $\psi_0 \notin \Gamma^\omega$ and $\psi_1 \notin \Gamma^\omega$. By construction, $\Gamma^\omega \vdash \psi_0 \supset \varphi$ and $\Gamma^\omega \vdash \psi_1 \supset \varphi$. Since Γ^ω is deductively closed, by ($\vee$-E) rule, we have $\Gamma^\omega \vdash \varphi$, which contradicts to the previous fact.
3. Since $\Gamma^\omega \nvdash \varphi$, $\Gamma^\omega \nvdash \bot$.

Since $\Gamma = \Gamma^0$, Γ^ω contains Γ^0. The lemma is now proved.

Definition 6 (Canonical model candidate). *We define a tuple $M^c = \langle W^c, \preceq^c, (f_a^c)_{a \in A}, \rho^c \rangle$ where*

- *W^c is the set of saturated sets of formulas,*
- *$\Gamma \preceq^c \Delta$ if and only if $\Gamma \subseteq \Delta$,*
- *$f_a^c(\Gamma) = \{\varphi \mid K_a \varphi \in \Gamma\}$,*
- *$\rho^c(P) = \{\Gamma \mid P \in \Gamma\}$.*

Lemma 2 (Canonical model). *The tuple $M^c = \langle W^c, \preceq^c, (f_a^c)_{a \in A}, \rho^c \rangle$ is a model.*

Proof. First, let us check f_a^c is actually a function $W^c \to W^c$. Assume $\Gamma \in W^c$.

To prove that $f_a(\Gamma)$ is a saturated set of formulas, we check each condition on the Definition 5 of saturated sets.

1. Assume $f_a^{\mathbf{C}}(\Gamma) \vdash \varphi$. By rule (nec),$K_a(f_a(\Gamma)) \vdash K_a\varphi$. Since $K_a(f_a^{\mathbf{C}}(\Gamma)) \subseteq \Gamma$, the relation $\Gamma \vdash K_a\varphi$ holds. Since Γ is deductively closed, $K_a\varphi \in \Gamma$. By definition of $f_a^{\mathbf{C}}$, $\varphi \in f_a^{\mathbf{C}}(\Gamma)$.
2. Assume $\varphi \vee \psi \in f_a^{\mathbf{C}}(\Gamma)$. By definition of $f_a^{\mathbf{C}}$, $K_a(\varphi \vee \psi) \in \Gamma$. By rule $(\vee K_a)$, $K_a(\varphi \vee \psi) \vdash K_a\varphi \vee K_a\psi$. Since Γ is deductively closed, $K_a\varphi \vee K_a\psi \in \Gamma$. Since Γ is saturated, either $K_a\varphi \in \Gamma$ or $K_a\psi \in \Gamma$. By definition of $f_a^{\mathbf{C}}$, either $\varphi \in f_a^{\mathbf{C}}(\Gamma)$ or $\psi \in f_a^{\mathbf{C}}(\Gamma)$.
3. Seeking contradiction, assume $f_a^{\mathbf{C}}(\Gamma) \vdash \bot$. Since $f_a^{\mathbf{C}}(\Gamma)$ is deductively closed, $\bot \in f_a^{\mathbf{C}}(\Gamma)$. By definition of $f_a^{\mathbf{C}}$, $K_a\bot \in \Gamma$. Because of the rule (T), $\Gamma \vdash \bot$. This contradicts to the assumption of Γ being a saturated set.

Now, let us check each condition in Definition 3 to make sure the tuple $M^{\mathbf{C}}$ is actually a model:

1. $\preceq^{\mathbf{C}}$ is a partial order because the set theoretic inclusion $\subseteq$ is a partial order.
2. (a) $f_a^{\mathbf{C}}(\Gamma) \preceq^{\mathbf{C}} \Gamma$ by the rule (T).
   (b) $f_a^{\mathbf{C}}(f_a^{\mathbf{C}}(\Gamma)) \preceq^{\mathbf{C}} f_a^{\mathbf{C}}(\Gamma)$ is now obvious from the previous line. Let us show the opposite. Assume $\varphi \in f_a^{\mathbf{C}}(\Gamma)$. By the definition of $f_a^{\mathbf{C}}$, $K_a\varphi \in \Gamma$. By rule (introspection), $\Gamma \vdash K_aK_a\varphi$. Since Γ is deductively closed, $K_aK_a\varphi \in \Gamma$. Thus $\varphi \in f_a^{\mathbf{C}}(f_a^{\mathbf{C}}(\Gamma))$.
   (c) Assume $\Gamma \preceq \Delta$. Every $K_a\varphi \in \Delta$ is also in Γ. Thus $f_a^{\mathbf{C}}(\Gamma) \preceq f_a^{\mathbf{C}}(\Delta)$.
3. Assume $\Gamma' \succeq \Gamma \in \rho^{\mathbf{C}}(P)$. $P \in \Gamma$. So $P \in \Gamma'$. Thus $\Gamma' \in \rho^{\mathbf{C}}(P)$.

Lemma 3. *For a saturated set of formula Γ and the canonical model $M^{\mathbf{C}}$, an equivalence $\varphi \in \Gamma \Leftrightarrow M^{\mathbf{C}}, \Gamma \vdash \varphi$ holds.*

Proof. By induction on φ.

(Case $\varphi = \bot$). Neither side ever holds.
(Case $\varphi = P$). By the definition of $\rho^{\mathbf{C}}$, $\varphi \in \Gamma \Leftrightarrow \Gamma \in \rho(P) \Leftrightarrow M^{\mathbf{C}}, \Gamma \models P$.
(Case $\varphi = \psi_0 \wedge \psi_1$)(Case $\varphi = \psi_0 \vee \psi_1$)(Case $\varphi = K_a\psi$). Directly from the induction hypotheses.
(Case $\varphi = \psi_0 \supset \psi_1$). ($\Rightarrow$) Assume $M^{\mathbf{C}}, \Gamma \models \psi_0 \supset \psi_1$. Seeking contradiction, assume $\psi_0 \supset \psi_1 \notin \Gamma$. Since Γ is deductively closed, $\Gamma, \psi_0 \nvdash \psi_1$. By Lemma 1, there exists a saturated set Γ' with $\Gamma' \supseteq \Gamma \cup \{\psi_0\}$ and $\Gamma' \nvdash \psi_1$. By induction hypothesis, $M^{\mathbf{C}}, \Gamma' \models \psi_0$ but not $M^{\mathbf{C}}, \Gamma' \models \psi_1$. Since $\Gamma' \succeq \Gamma$, this contradicts to $M^{\mathbf{C}}, \Gamma \models \psi_0 \supset \psi_1$.
($\Leftarrow$) Assume $\psi_0 \supset \psi_1 \in \Delta$, $\Delta' \succeq \Delta$ and $M^{\mathbf{C}}, \Delta' \models \psi_0$. Showing $M^{\mathbf{C}}, \Delta' \models \psi_1$ is enough. By induction hypothesis, $\psi_0 \in \Delta'$. Since Δ' is deductively closed and $\psi_0 \supset \psi_1 \in \Delta'$, $\psi_1 \in \Delta'$. By induction hypothesis, $M^{\mathbf{C}}, \Delta' \models \psi_1$.

Theorem 5 (Strong completeness). *$\Gamma \models \varphi$ implies $\Gamma \vdash \varphi$.*

Proof. We show the contraposition: assuming $\Gamma \nvdash \varphi$, we show $\Gamma \not\models \varphi$. By Lemma 1, there is a saturated set of formula Γ' with $\Gamma' \nvdash \varphi$ and $\Gamma' \supseteq \Gamma$. By Lemma 3, $M^{\mathbf{C}}, \Gamma' \models \Gamma$ but not $M^{\mathbf{C}}, \Gamma' \models \varphi$. This denies $\Gamma \models \varphi$.

3 Axiom Type for Sequential Consistency

A schedule determines a temporal partial order of events such as message sending and receiving. A correct program must behave correctly under every schedule. Shared memory consistency is a restriction on schedules. When a stronger memory consistency is posed, it is easier for programs to behave correctly. This is analogous to the fact that when a stronger condition on models implies more valid formulas.

In this section, we characterize sequential consistency with a set of axioms. Sequential consistency defined by Lamport [17] is essentially a condition requiring the states of memory to be lined up in a total order. We define a deduction system $\vdash_{SC}$ by adding an axiom type to $\mathbf{K}\vee$ and characterize sequential consistency.

Henceforth, we assume $A = \{\mathsf{m}\} \cup P$ ($\mathsf{m} \notin P$), where P is the set of processes and m represents the shared memory. A memory state is a state w with $f_{\mathsf{m}}(w) = w$.

Definition 7. *We let SC be the set of formula of the form* $(K_{\mathsf{m}}\varphi \supset K_{\mathsf{m}}\psi) \vee (K_{\mathsf{m}}\psi \supset K_{\mathsf{m}}\varphi)$.

We add a rule (SC) to the previous calculus $\vdash$*:* $(SC)\ \dfrac{}{\vdash \varphi}$ *where* $\varphi \in SC$. *Except this additional rule, we define* $\Gamma \vdash_{SC} \varphi$ *in the same way as* $\Gamma \vdash \varphi$.

Note that all axioms in the set SC are classical tautologies so that adding these axioms to classical logic is meaningless. In other words, if we force the intuitionistic epistemic logic to be classical by adding the double negation elimination rule, we obtain an epistemic logic where every truth is known to every agent, where shared memory consistency is meaningless. This clarifies the merit of using intuitionistic logic instead of classical logic.

Definition 8. *A sequential model is a model where any two memory states* w *and* w' *satisfy either* $w \preceq w'$ *or* $w' \preceq w$.

3.1 Soundness

Lemma 4. $\vdash_{SC} \varphi \Rightarrow M \models \varphi$ *for any sequential model* M.

Proof. We extend the inductive proof of Lemma 4 with a clause for the rule (SC).

(SC) Seeking contradiction, assume $M, w \not\models (K_{\mathsf{m}}\varphi \supset K_{\mathsf{m}}\psi) \vee (K_{\mathsf{m}}\psi \supset K_{\mathsf{m}}\varphi)$. The definition for $\models$ says that there exist states $w_0, w_1 \succeq w$ with $M, w_0 \models K_{\mathsf{m}}\varphi$, $M, w_1 \models K_{\mathsf{m}}\psi$, $M, w_1 \not\models K_{\mathsf{m}}\psi$ and $M, w_0 \not\models K_{\mathsf{m}}\varphi$. These and Kripke monotonicity (Lemma 1) show M is not a pre-sequential model. Thus M is not a sequential model.

Other cases are the same as Lemma 4.

3.2 Strong Completeness

Definition 9. *A pre-sequential model is a model where any two memory states w and w' with a common ancestor satisfy either $w \preceq w'$ or $w' \preceq w$.*

In this definition, a common ancestor x of w and w' is a state x with both $x \preceq w$ and $x \preceq w'$.

Definition 10. *A set of formulas Γ is SC-saturated if and only if all of these conditions are satisfied:*

1. *Γ is SC-deductively closed, i.e., $\Gamma \vdash_{\mathrm{SC}} \varphi \Rightarrow \varphi \in \Gamma$,*
2. *$\varphi \vee \psi \in \Gamma \Rightarrow \varphi \in \Gamma$ or $\psi \in \Gamma$,*
3. *$\Gamma \nvdash_{\mathrm{SC}} \bot$.*

Lemma 5 (Saturation lemma). *For a set of formulas Γ with $\Gamma \nvdash_{\mathrm{SC}} \varphi$, there exists a saturated set of formulas Γ^{ω} with $\Gamma^{\omega} \nvdash_{\mathrm{SC}} \varphi$ and $\Gamma \subset \Gamma^{\omega}$.*

Proof. The same as Lemma 1 where each $\vdash$ is replaced by $\vdash_{\mathrm{SC}}$.

Definition 11 (Canonical model candidate for sequential consistency). *We define a tuple $M^{sc} = \langle W^{\mathrm{SC}}, \preceq^{\mathrm{SC}}, (f_a^{\mathrm{SC}})_{a \in A}, \rho^{\mathrm{SC}} \rangle$ in the same way as Definition 6 of M^{c} except that W^{sc} is the set of SC-saturated sets of formulas.*

Lemma 6 (Canonical model for sequential consistency). *The tuple M^{sc} is a pre-sequential model.*

Proof. First, we can show, in the same way as before, that checking f_a^{SC} is actually a function $W^{\mathrm{SC}} \to W^{\mathrm{SC}}$. Also, checking each condition in Definition 3 is similar so that we see M^{SC} is actually a model. Finally, to see that the model M^{SC} is sequential, let Γ, Δ and Θ be states of M^{SC} and assume all of $\Theta \preceq^{\mathrm{SC}} \Gamma$, $\Theta \preceq^{\mathrm{SC}} \Delta$, $f_{\mathsf{m}}^{\mathrm{SC}}(\Gamma) = \Gamma$ and $f_{\mathsf{m}}^{\mathrm{SC}}(\Delta) = \Delta$. We claim that either $\Delta \preceq^{\mathrm{SC}} \Gamma$ or $\Gamma \preceq^{\mathrm{SC}} \Delta$ holds. Seeking contradiction, deny the claim. Since the relation $\preceq^{\mathrm{SC}}$ is actually the set theoretic inclusion, there exist formulas φ and ψ with $\varphi \in \Gamma$, $\varphi \notin \Delta$, $\psi \in \Delta$ and $\psi \notin \Gamma$. Since $f_{\mathsf{m}}^{\mathrm{SC}}(\Gamma) = \Gamma$, $K_a\psi \notin \Gamma$ and $K_a\varphi \in \Gamma$ hold. Similarly, $K_a\varphi \notin \Delta$ and $K_a\psi \in \Delta$ hold. Since Θ is SC-saturated, $(K_a\varphi \supset K_a\psi) \vee (K_a\varphi \supset K_a\psi)$ is in Θ. The definition of saturation says either $K_a\varphi \supset K_a\psi \in \Theta$ or $K_a\psi \supset K_a\varphi \in \Theta$. Consequently, either $K_a\varphi \supset K_a\psi \in \Gamma$ or $K_a\psi \supset K_a\varphi \in \Delta$ holds. Each case leads to contradiction by deductive closedness of Γ and Δ.

Lemma 7. *For an SC-saturated set of formulas Γ and the canonical model for sequential consistency M^{sc}, an equivalence $\varphi \in \Gamma \iff M^{sc}, \Gamma \vdash_{\mathrm{SC}} \varphi$ holds.*

Proof. This lemma can be proved in the same way as Lemma 3.

Lemma 8. *For a pre-sequential model M and a state w of M, there exists a sequential model M' and a state w' of M' such that $M, w \models \psi$ if and only if $M', w' \models \psi$ for any formula ψ.*

Proof. Let M be $\langle W, \preceq (f_a)_{a\in A}, \rho\rangle$. We define W' to be the set $\{v \in W \mid$ there exists $x \in W$ with $x \preceq v$ and $x \preceq w\}$. Also, we define $\preceq', f'_a$ and ρ' to be $\preceq, f_a$ and ρ respectively restricted on W'.

To check $M' = \langle W', \preceq', (f'_a)_{a\in A}, \rho'\rangle$ is actually a sequential model, the only non-trivial part is showing $f_a(W') \subseteq W'$. Let us take $v \in W'$ arbitrarily. There exists $x \in W$ with $x \preceq v$ and $x \preceq w$. Since f_a is monotonic, $f_a(x) \preceq f_a(v)$. Since f_a is descending, $f_a(x) \preceq x \preceq w$. These combined show $f_a(v) \in W$.

Showing the equivalence $M, w \models \psi \iff M', w \models \psi$ is a straightforward induction on ψ.

Theorem 6 (Strong completeness for sequential consistency). *$\Gamma \vdash_{\mathrm{SC}} \varphi$ holds if $M \models \Gamma$ implies $M \models \varphi$ for every sequential model M.*

Proof. We show the contraposition: assuming $\Gamma \nvdash_{\mathrm{SC}} \varphi$, we show that there exists a sequential model M that satisfies $M \models \Gamma$ but not $M \models \varphi$. By Lemma 5, there is an SC-saturated set of formula Γ' with $\Gamma' \nvdash \varphi$ and $\Gamma' \supset \Gamma$. By Lemma 7, $M^{\mathrm{SC}}, \Gamma' \models \Gamma$ but not $M^{\mathrm{SC}}, \Gamma' \models \varphi$. By Lemma 6 and Lemma 8, there exists a sequential model M' with a state $w' \in M'$ such that $M', w' \models \Gamma$ but not $M', w' \models \varphi$.

Example Theorem. In the introduction, we gave an example of theorems of $\vdash_{\mathrm{SC}}$: $(K_p K_{\mathsf{m}} K_p \varphi \wedge K_q K_{\mathsf{m}} K_q \psi) \supset (K_p K_q \psi \vee K_q K_p \varphi)$. We give a proof for this theorem in Figure 2.

4 Waitfree Computation

We define a class of formulas called waitfree assertions, which have a special finite model property (Theorem 7): if a waitfree assertion is consistent[2], there is a finite model of a special shape where the assertion is valid. The special shape mimics the scheduling of shared memory defined by Saks and Zaharoglou [24].

Definition 12. *Assume there is a vector of atomic formulas $(I_p)_{p\in P}$.*

A waitfree protocol description φ is a formula of the form

$$\varphi = \bigwedge_{p\in P} K_p K_{\mathsf{m}} K_p \cdots K_{\mathsf{m}} K_p I_p$$

where K_p and K_{m} appear alternatively in and around "$\cdots$".

A waitfree task specification ψ is defined with the BNF:

$$\psi ::= K_p \psi \mid \psi \wedge \psi \mid \psi \vee \psi \mid I_p$$

where p stands for a process in P.

A waitfree assertion is a formula $\varphi \supset \psi$ where φ is a waitfree protocol description and ψ is a waitfree task specification.

[2] A formula φ is consistent if and only if $\bot$ cannot be proved even if φ is added as an axiom.

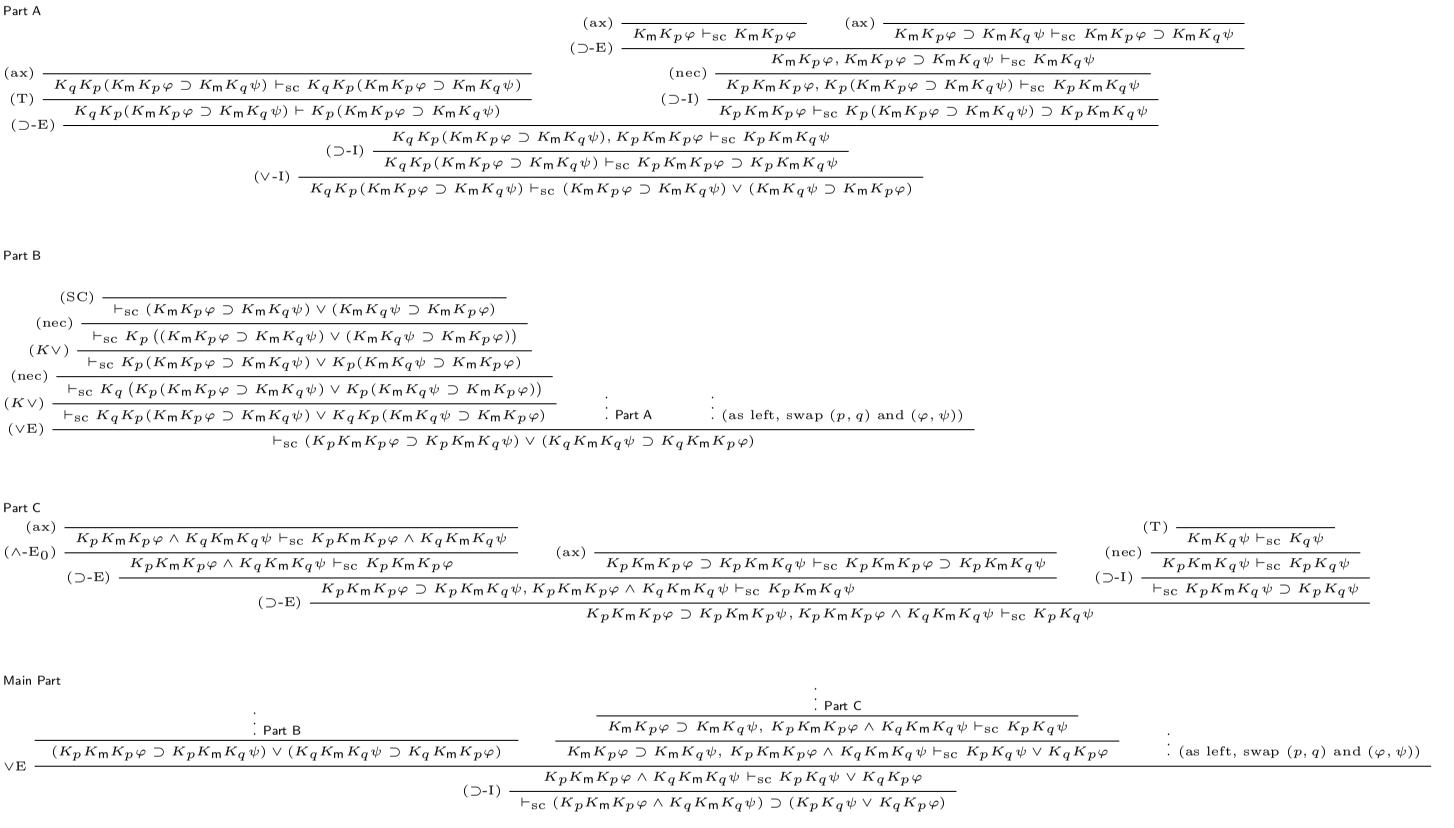


Fig. 2. A proof diagram for an example theorem $(K_pK_{\mathsf{m}}K_p\varphi \wedge K_qK_{\mathsf{m}}K_q\psi) \supset (K_pK_q\psi \vee K_qK_p\varphi)$ in $\vdash_{\mathrm{sc}}$

We are only interested in reasoning about a fixed protocol so that each process interacts with the memory for only finite times. In addition to this restriction, there is no process–process communication although there is process–memory communication so that a protocol can be described by a formula without nesting of different process modalities such as K_pK_q. Finally, we forcefully decide that we are only interested in existence of knowledge at the end of protocols so that the requirement of a task can be represented in a positive formula. The formula $(K_pK_{\mathsf{m}}K_p\varphi \wedge K_qK_{\mathsf{m}}K_q\psi) \supset (K_pK_q\psi \vee K_qK_p\varphi)$ proved in Figure 2 is a waitfree assertion.

Definition 13. *A partial schedule $(\sigma_i)_{i\in I}$ is a finite sequence of subsets of P.*

Definition 14. *For a process $p \in P$ and a partial schedule σ, $count_p(\sigma)$ is the cardinality $|\{i \in I \mid p \in \sigma_i\}|$.*

For a waitfree protocol description $\varphi = \bigwedge_{p\in P} K_pK_{\mathsf{m}} \cdots K_pI_p$, $count_p(\varphi)$ is the number of K_{m} occurrences in $K_pK_{\mathsf{m}} \cdots K_pI_p$.

A partial schedule σ is compatible to a waitfree protocol description φ if $count_p(\varphi) = count_p(\sigma)$ for any process $p \in P$.

Definition 15. *For a waitfree protocol description φ and a compatible partial schedule $(\sigma_i)_{i\in I}$, we define a waitfree schedule model $R(\varphi,\sigma) = \langle W, \preceq, (f_a)_{a\in A}, \rho\rangle$ as:*

- $W = \{(p,i) \in P \times \mathbb{N} \mid p \in \sigma_i\} \cup \{(p,i)' \in P \times \mathbb{N} \mid p \in \sigma_i\} \cup \{(\mathsf{m},i) \mid i \in I\} \cup \{(o,i) \mid i \in I\} \cup \{\bot\}$
- $(p,i) \preceq (m,i+1) \preceq (p,i)'$ *for* $p \in P$; $(a,j) \preceq (o,i)$ *if and only if* $j \le i$ *for* $a \in A$; $\bot \preceq w$ *for all* $w \in W$; *and* $(a,j)' \preceq (o,i)$ *if and only if* $j \le i$ *for* $a \in A$.
- $f_a(w) = \begin{cases} \text{the least } (a,j) \text{ with } (a,j) \preceq w \text{ (if there exists such } (a,j)) \\ \quad \text{(the definition of } \preceq \text{ assures there is the least such } (a,j)), \\ \bot \text{ (if such } (a,j) \text{ does not exist).} \end{cases}$
- $\rho(I_p) = \{w \in W \mid (p,0) \preceq w\}$.

An example of a model induced by a partial schedule is shown in Figure 3.

Using the definitions above, we can state the logical characterization of waitfree communication.

Theorem 7 (Completeness for waitfree communication). *Assume $\varphi \supset \psi$ is a waitfree assertion. The relation $\vdash_{SC} \varphi \supset \psi$ holds if the relation $R(\varphi,\sigma),(o,n) \models \psi$ holds for any compatible partial schedule σ where the state (o,n) is the last state of the waitfree model $R(\varphi,\sigma)$.*

To prove completeness, we only use special models called singleton models induced by a permutation of processes.

Definition 16. *For a set of processes P, we define $\mathsf{S}(P)$ to be the set of the permutations of P.*

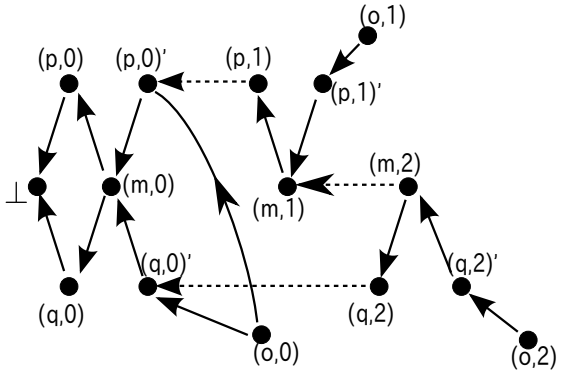


Fig. 3. A model induced by the partial schedule $\{p,q\},\{p\},\{q\}$. A solid arrow pointing to (a,n) shows an f_a mapping. Dotted arrows show $\preceq$ relations. We omit inferable arrows and the valuation.

Definition 17. *For $\pi \in \mathsf{S}(P)$ and $0 \le k \le |P|$, we define $SC(\pi,k)$ to be the set $\{K_{\mathsf{m}}K_pI_p \supset K_{\mathsf{m}}K_qI_q \mid$ there exist i and j with $j \le i \le k$ such that $\pi_i = a$ and $\pi_j = b\}$.*

Lemma 9. $\vdash_{\mathrm{sc}} \bigvee_{\pi\in\mathsf{S}(P)} SC(\pi,|P|)$ *holds.*

Proof. It suffices to use rule (SC) many times.

Definition 18. *For a permutation π of P and a waitfree protocol description φ, we define a partial schedule $\sigma(\varphi,\pi)$ as*

$$\sigma(\varphi,\pi) = \overbrace{\pi_0,\cdots,\pi_0}^{count_{\pi_0}(\varphi)},\overbrace{\pi_1,\cdots,\pi_1}^{count_{\pi_1}(\varphi)},\cdots\cdots\cdots,\overbrace{\pi_n,\cdots,\pi_n}^{count_{\pi_n}(\varphi)}.$$

Definition 19. *A singleton model is a model of the form $R(\varphi,\sigma(\varphi,\pi))$. We abbreviate this to $R(\varphi,\pi)$.*

For a singleton model and an index $k \in I$, w_k denotes the minimum external observer state above all π_j states for $j < k$.

Definition 20. *For a waitfree protocol description $\varphi = \bigwedge_{p\in P} \overbrace{K_pK_{\mathsf{m}}K_p\cdots K_p}^{n_p} I_p$, we define the restriction*

$\varphi \restriction_{p,k} = \bigwedge_{p\in P\restriction_{p,k}} \overbrace{K_pK_{\mathsf{m}}K_p\cdots K_p}^{n_p} I_p$ *where* $P \restriction_{p,k} = \{p \in P \mid p_j = p$ *for some* $j < k\}$.

Lemma 10. $R(\varphi,\pi),(o,k) \models \psi \Longrightarrow SC(\pi,k) \vdash \varphi \restriction_{\pi,k} \supset \psi$.

Proof (Proof of Lemma 10). By induction on k.

(Case $k = 0$). We show a stronger proposition: $(o, 0) \models \psi$ implies $f_{p_0}(o, 0) \models \psi$, $\vdash \varphi \restriction_{p,0} \supset \psi$ and
$\vdash \varphi \restriction_{p,0} \supset K_a\psi$ by inner induction on ψ.

(When ψ is an atomic formula P). $P = I_{\pi_0}$ holds.
Since $\varphi \restriction_{\pi,0} = K_{\pi_0} K_{\mathsf{m}} K_{\pi_0} \cdots K_{\mathsf{m}} K_{\pi_0} I_{\pi_0}$, $\vdash \varphi \restriction_{\pi,0} \supset K_{\pi_0} P$ holds. So, $SC(\pi, 0) \vdash \varphi \restriction_{\pi,0} \supset K_{\pi_0} P$ holds. Consequently, $SC(\pi, 0) \vdash \varphi \restriction_{\pi,0} \supset P$ also holds.

(When $\psi = \psi_0 \wedge \psi_1$ or $\psi_0 \vee \psi_1$). Induction goes smoothly.

(When $\psi = K_a\psi'$). Assume $(o, 0) \models K_a\psi'$. Claim: $a = \pi_0$ holds. Seeking contradiction, assume $a \neq \pi_0$. That means $f_a((o, 0)) = \bot$. However, no waitfree task specification is satisfied at the state $\bot$. Contradiction. We have proved $a = \pi_0$. Using this, we can show that $f_a((o, 0)) \models \psi'$ holds. By idempotency of f_a, $f_a(f_a((o, 0))) \models \psi'$ holds. This means $f_a((o, 0)) \models K_a\psi'$. Since $(o, 0) \models \psi'$, by inner induction hypothesis, $\vdash \varphi \restriction_{\pi,0} \supset K_a\psi'_a$. By proof theoretic consideration, $\vdash \varphi \restriction_{\pi,0} \supset K_a K_a \psi'$ holds.

(Case $k = k' + 1$). Like the base case, we show a stronger proposition $(o, k) \models \psi \Leftrightarrow f_{\pi_k}((o, k)) \models \psi \Rightarrow SC(\pi, k) \vdash \varphi \restriction_{\pi,k} \supset \psi$ and $SC(\pi, k) \vdash \varphi \restriction_{\pi,k} \supset K_{\pi_k}\psi$, using inner induction on ψ.

(When $\psi = P$, an atomic formula). Either $R(\varphi, \pi), w_{k'} \models P$ or $I_{\pi_k} = P$ holds. In the former case, by induction hypothesis. In the latter case, similarly as the base case.

(When $\psi = \psi_0 \wedge \psi_1$ or $\psi_0 \vee \psi_1$). Induction goes smoothly.

(When $\psi = K_a\psi'$). If $\pi_k \neq a$, $f_{\pi_k}((o, k)) \models K_a\psi'$ implies $(o, k') \models K_a\psi'$. By outer induction hypothesis, $SC(\pi, k') \vdash \varphi \restriction_{\pi,k'} \supset K_a\psi'$ and $SC(\pi, k') \vdash \varphi \restriction_{\pi,k'} \vdash \varphi \restriction_{\pi,k'} \supset K_a\psi'$ hold. Here, we can safely replace k' with k. If $\pi_k = a$, $(o, k) \models K_a\psi'$ imply $(o, k) \models \psi'$. By inner induction hypothesis, we obtain $SC(\pi, k) \vdash \varphi \restriction_{\pi,k} \supset K_a\psi'$. This also implies $SC(\pi, k) \vdash \varphi \restriction_{\pi,k} \supset K_a K_a \psi'$.

After showing this generalized lemma, proving Theorem 7 is easy.

Proof (Proof of Theorem 7). Since $R(\varphi, p), w_{|P|} \models \psi$, $SC(p, |P|) \vdash \varphi \supset \psi$. By Lemma 9, $\vdash_{\mathrm{SC}} \varphi \supset \psi$.

Any model induced by a partial schedule is finite. For a waitfree assertion φ, it is decidable whether $\vdash_{\mathrm{SC}} \varphi$ holds or not.

4.1 Decidability of Solvability of Waitfree Task Specification

Definition 21. *A waitfree task specification ψ is solvable if there is such a waitfree protocol description φ that the relation $R(\varphi, \sigma), (o, n) \models \psi$ holds for any compatible partial schedule σ where the state (o, n) is the last state of the model $R(\varphi, \sigma)$.*

Fact. By Theorem 7, the set of solvable waitfree task specifications are recursively enumerable because the relation $\vdash_{\mathrm{SC}}$ is axiomatized.

Fact. The set of unsolvable waitfree task specifications are recursively enumerable because schedule-induced models are recursively enumerable.

These two facts imply that it is decidable whether a waitfree task specification is solvable or not. This does not contradict the undecidability of waitfreely solvable tasks by Gafni and Koutsoupias [8] because the undecidability proof utilizes tasks that cannot be expressed by waitfree task specifications. They use tasks involving consensus: the tasks involving making agreements among processes, where whether an output value is allowed or not depends on other processes' *output* values. Waitfree tasks specifications cannot describe such tasks.

5 Related Work

Ondrej Majer's Epistemic Logic with Relevant Agents [19] is similar to $\mathbf{K}\vee$ in that both logics have epistemic modalities and that both logics are not classical. However, the logic given in [19] contains only one modality K for knowledge. This implicitly assumes that there is a single agent, not multiple agents so that it is impossible for their logic to treat communication between multiple agents.

Many logics have both temporal and epistemic modalities. Ewald [7] proposes an intuitionistic logic with temporal modality. In Kobayashi and Yonezawa's logic [15], processes appear in formulas but time does not appear in formulas because time is implicit in the system of logic programming. This logic is different from $\mathbf{K}\vee$ in that this logic is based on linear logic and that their usage is logic programming.

6 Discussions

Waitfree Computation. The Gödel Prize in 2004 was given to Herlihy and Shavit [11] and Saks and Zaharoglou [24]. This work was motivated by these papers. Herlihy and Shavit [11] used subdivision of colored simplicial complex to model waitfree computation. Each vertex $v \in V$ is colored by an agent. Each simplex $s \in S \subseteq \mathcal{P}(V)$ contains vertices with distinct colors. A vertex may have an ancestor simplex called carrier. The union of the ancestor relation and the relation $\in$ can be turned into an order $\sqsubset$ on $V \sqcup S$ by taking the reflexive and transitive closure. We can define a partial $f_a : S \to S$ where S is the set of simplex in a simplicial complex by letting $f_a(s) = \{x\}$ where x is the maximum vertex below s (w.r.t. $\sqsubset$) whose color is a. When we add a bottom simplex $\bot$ and make f_a total, we can regard a simplicial complex as a model of $\mathbf{K}\vee$ as in an example (Figure 4).

Saks and Zaharoglou [24] use full-information protocols [29]. Even the shared variables remember the whole history. In every component, knowledge increases monotonically through time. This monotonicity suggests that their model can be analyzed effectively in Kripke models for intuitionistic logic. Saks and Zaharoglou [24] also suggest that "it will be worthwhile to explore the connection with the formal theory of distributed knowledge." This work is following their suggestion by treating waitfree communication in a formal way, especially using a logic with epistemic modalities.

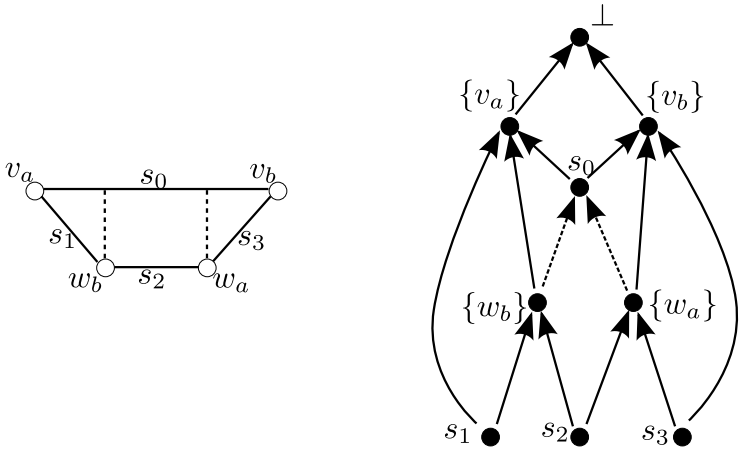


Fig. 4. How subdivision of simplicial complexes is transformed into a $\mathbf{K}\vee$ frame. Left: A simplex $s_0 = \{v_a, v_b\}$ is subdivided into $s_1 = \{v_a, w_b\}$, $s_2 = \{w_a, w_b\}$ and $s_3 = \{w_a, v_b\}$. Right: $\mathbf{K}\vee$ frame obtained from the left subdivision.

Other Consistency Models. Steinke and Nutt [25] gave a lattice of consistency properties including: sequential, causal, processor, PRAM, cache, slow, and local consistency. It remains to model these other consistency properties with axiom schemata in $\mathbf{K}\vee$.

Sequential Consistency or Linearizability. Attiya and Welch [2] pointed out that sequential consistency [17] and linearizability [12] are often confused. We make sure that the deduction system $\vdash_{SC}$ does not characterize linearizability. Herlihy [12] stated that linearizability is a local property; in other words, when each memory object satisfies linearizability, the combined system also has linearizability. However, the axiom type *SC* is not local. To see that, let us consider two variants of the axiom type *SC* for different two memory objects m and m', namely, $(K_{\mathsf{m}}\varphi \supset K_{\mathsf{m}}\psi)\vee(K_{\mathsf{m}}\psi \supset K_{\mathsf{m}}\varphi)$ and $(K_{\mathsf{m}'}\varphi \supset K_{\mathsf{m}'}\psi)\vee(K_{\mathsf{m}'}\psi \supset K_{\mathsf{m}'}\varphi)$. Even with both of these, the *mixed* axiom type $(K_{\mathsf{m}'}\varphi \supset K_{\mathsf{m}}\psi)\vee(K_{\mathsf{m}}\psi \supset K_{\mathsf{m}'}\varphi)$ is not derivable.

Latency versus Throughput. Our logic is more suitable for a situation where latency is more important than throughput. Since we consider time as the partial order of intuitionistic Kripke models, all knowledge must be preserved during time progress. Communication must be done in full-information manner (as in full-information protocols in [29]) because messages define the partial order. Our logic is advantageous when latency is important so that it is important to know how many message interactions are needed to accomplish a certain task. We plan to investigate network protocols with $\mathbf{K}\vee$.

Disjunction Distribution Over K Modality. Since the semantics for modalities is defined by functions on Kripke frames, the disjunction distributes modalities

in $\mathbf{K}\vee$. Kojima and Igarashi [16] avoids the distribution of modalities over disjunction by giving up functional modality. On the other hand, $\mathbf{K}\vee$ has distribution. We speculate that the difference comes from the different interpretations of modalities according to time: in [16] inner subformulas within the scope of the modality are interpreted in the future; while in $\mathbf{K}\vee$, inner subformulas within the scope of the modalities are interpreted in the past.

By translation of Suzuki [26], when A is a singleton set, $\mathbf{K}\vee$ corresponds to the intuitionistic predicate logic with singleton domain in the same manner the models of the logic L_3 of Ono [20] correspond to the models of intuitionistic predicate logic with constant domain. This fact suggests that the semantics of $\mathbf{K}\vee$ is very simple when there is only one agent. Simplicity was our aim at the beginning.

Acknowledgments. The author thanks Masami Hagiya and Yoshihiko Kakutani for encouragements and valuable advice. The anonymous refrees' careful comments considerably improved this paper.

References

[1] Alechina, N., Mendler, M., de Paiva, V., Hitter, E.: Categorical and Kripke Semantics for Constructive S4 Modal Logic. In: Fribourg, L. (ed.) CSL 2001 and EACSL 2001. LNCS, vol. 2142, pp. 292–307. Springer, Heidelberg (2001)
[2] Attiya, H., Welch, J.L.: Sequential consistency versus linearizability. ACM Transactions on Computer Systems 12(2), 122 (1994)
[3] Balbiani, P., et al.: 'Knowable' as 'known after an announcement'. The Review of Symbolic Logic 1(03), 305–334 (2008)
[4] Baltag, A., Coecke, B., Sadrzadeh, M.: Epistemic actions as resources. Journal of Logic and Computation 17(3), 555 (2007)
[5] Bieber, P., Onera-Cert, T.: A logic of communication in hostile environment. In: Proceedings of Computer Security Foundations Workshop III, 1990, pp. 14–22 (1990)
[6] Costa, V., Benevides, M.: Formalizing concurrent common knowledge as product of modal logics. Logic Journal of IGPL 13(6), 665 (2005)
[7] Ewald, W.B.: Intuitionistic tense and modal logic. The Journal of Symbolic Logic 51(1), 166–179 (1986)
[8] Gafni, E., Koutsoupias, E.: Three-processor tasks are undecidable. SIAM Journal on Computing 28(3), 970–983 (1999)
[9] Halpern, J.Y., Moses, Y.: Knowledge and common knowledge in a distributed environment. Journal of the ACM (JACM) 37(3), 549–587 (1990)
[10] Herlihy, M.: Wait-free synchronization. ACM Transactions on Programming Languages and Systems 13(1), 124–149 (1991)
[11] Herlihy, M., Shavit, N.: The topological structure of asynchronous computability. Journal of the ACM 46(6), 858–923 (1999)
[12] Herlihy, M., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems (TOPLAS) 12(3), 463–492 (1990)
[13] Hintikka, J.: Knowledge and belief: an introduction to the logic of the two notions. Cornell University Press, Ithica (1962)

[14] Jia, L., Walker, D.: Modal Proofs as Distributed Programs (Extended Abstract). In: Schmidt, D. (ed.) ESOP 2004. LNCS, vol. 2986, p. 219. Springer, Heidelberg (2004)
[15] Kobayashi, N., Yonezawa, A.: Asynchronous communication model based on linear logic. Formal Aspects of Computing 7(2), 113–149 (1995)
[16] Kojima, K., Igarashi, A.: On constructive linear-time temporal logic. In: Proc. of IMLA, p. 8 (2008)
[17] Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess program. IEEE Transactions on Computers 100(28), 690–691 (1979)
[18] Liau, C.J.: Belief, information acquisition, and trust in multi-agent systemsA modal logic formulation. Artificial Intelligence 149(1), 31–60 (2003)
[19] Majer, O., Peliš, M.: Epistemic logic with relevant agents. In: The Logica Yearbook 2008, pp. 123–135. Kings College Publications (2009)
[20] Ono, H.: On some intuitionistic modal logics. Publ. Res. Inst. Math. Sci. 13(3), 687–722 (1977)
[21] Peleg, D.: Communication in concurrent dynamic logic. J. Comp. Syst. Sci. 35(1), 23–58 (1987)
[22] Plaza, J.: Logics of public communications. Synthese 158(2), 165–179 (2007)
[23] Plotkin, G., Stirling, C.: A framework for intuitionistic modal logics: extended abstract. In: TARK 1986: Proceedings of the 1986 Conference on Theoretical Aspects of Reasoning About Knowledge, pp. 399–406. Morgan Kaufmann Publishers Inc., San Francisco (1986)
[24] Saks, M., Zaharoglou, F.: Wait-free k-set agreement is impossible: The topology of public knowledge. SIAM Journal on Computing 29(5), 1449–1483 (2000)
[25] Steinke, R.C., Nutt, G.J.: A unified theory of shared memory consistency. Journal of the ACM 51(5), 800–849 (2004)
[26] Suzuki, N.Y.: Kripke bundles for intermediate predicate logics and Kripke frames for intuitionistic modal logics. Studia Logica 49(3), 289–306 (1990)
[27] Troelstra, A.S., van Dalen, D.: Constructivism in Mathematics: An Introduction, vol. 1. North-Holland, Amsterdam (1988)
[28] van Benthem, J.: The information in intuitionistic logic. Synthese 167(2), 251–270 (2009)
[29] Woo, T.Y.C., Lam, S.S.: A lesson on authentication protocol design. SIGOPS Oper. Syst. Rev. 28(3), 24–37 (1994)

Disunification for Ultimately Periodic Interpretations

Matthias Horbach

Max-Planck-Institut für Informatik and Saarland University
Saarbrücken, Germany
horbach@mpi-inf.mpg.de

Abstract. Disunification is an extension of unification to first-order formulae over syntactic equality atoms. Instead of considering only syntactic equality, I extend a disunification algorithm by Comon and Delor to ultimately periodic interpretations, i.e. minimal many-sorted Herbrand models of predicative Horn clauses and, for some sorts, equations of the form $s^l(x) \simeq s^k(x)$. The extended algorithm is terminating and correct for ultimately periodic interpretations over a finite signature and gives rise to a decision procedure for the satisfiability of equational formulae in ultimately periodic interpretations.

As an application, I show how to apply disunification to compute the completion of predicates with respect to an ultimately periodic interpretation. Such completions are a key ingredient to several inductionless induction methods.

1 Introduction

Originally, *unification* [23] was the task of finding solutions to an equation $t \simeq t'$ of terms with respect to the free term algebra $\mathcal{T}(\mathcal{F})$, i.e. substitutions σ that instantiate the free variables of t and t' in such a way that $t\sigma$ and $t'\sigma$ are syntactically equal. The notion was then generalized to solving systems (i.e. conjunctions) of equations, and unification was recognized as a procedure that can be expressed using transformations of such systems [16,1].

From there on, the idea of unification was extended in at least two directions that are relevant for this work: On the one hand, Lassez et al. [18] examined systems of disequations, and later on a unified framework for the analysis of both equations and disequations was finally found in *disunification* [21,20,9]. Algorithmically, disunification procedures are algorithms rewriting first-order formulae over syntactic equality atoms into an equivalent normal form. On the theoretical side, they provide a decision procedure for the satisfiability in $\mathcal{T}(\mathcal{F})$ of (possibly quantified) formulae containing equality $\simeq$ as the only predicate symbol. Disunification has various applications, in particular in areas as logic programming [5], automated model building [3,12] and inductive theorem proving [7,10,14,15].

On the other hand, Plotkin [22] integrated sets E of equational axioms into the transformation rules used for unification, effectively unifying with respect

E.M. Clarke and A. Voronkov (Eds.): LPAR-16, LNAI 6355, pp. 290–311, 2010.

not to $\mathcal{T}(\mathcal{F})$ but to quotients $\mathcal{T}(\mathcal{F})/_E$ (see also [17]). Similar extensions were also made to disunification: Comon [6] developed disunification algorithms with respect to quotients $\mathcal{T}(\mathcal{F})/_E$ where E is a so-called quasi-free or compact axiomatization. Examples of such axiomatizations include sets of associativity and commutativity axioms. Fernández [13] used a narrowing-based approach to show that if E is a ground convergent rewrite system, the existential fragment of disunification is semi-decidable but in general undecidable even if E-unification is decidable and finitary.

In this article, I extend disunification to more general interpretations: Instead of considering only quotients of $\mathcal{T}(\mathcal{F})$, I allow minimal many-sorted Herbrand models of predicative Horn clauses and equations of the form $s^l(x) \simeq s^k(x)$ for some sorts. I will call such interpretations *ultimately periodic*. They occur naturally as quotients of the natural numbers or when models of formulae from propositional linear time temporal logics are described by clause sets [19]. The extended algorithm gives rise to a decision procedure for the satisfiability of equational formulae in ultimately periodic interpretations.

My algorithm is based on the disunification algorithm by Comon and Delor [8]. While there are other disunification algorithms available, this one has the advantage of being flexible in the sense that the control on its rules is kept as weak as possible.[1] Earlier algorithms like [9,7] required an often inefficient normal form (e.g. conjunctive normal form) computation after every step. The weak control used by Comon and Delor leaves wide space for the development of efficient instances in concrete implementations, which is important because disunification is NP-hard (SAT can easily be encoded by $x \mapsto x \simeq \text{true}$ and $\neg x \mapsto x \simeq \text{false}$). On the downside, the weak control makes the termination argument considerably more complicated than when formulae are kept normalized.

Predicative atoms are often integrated into a multi-sorted equational framework not explicitly but by adding a new sort `bool`, replacing each predicative atom $P(t_1, \ldots, t_n)$ by an equation $f_p(t_1, \ldots, t_n) \simeq \text{true}$ between terms of this sort, and then using algorithms designed for the purely equational setting. This is not so trivial for disunification because it does not prevent the need to extend disunification to a quotient $\mathcal{T}(\mathcal{F})/_E$, where E encodes the set of valid predicative atoms.

The addition of predicative atoms often makes disunification applicable for the completion of predicates, i.e. for the computation of those instances of a predicate that do not hold in a given interpretation. Comon and Nieuwenhuis [10] gave an algorithm how to complete predicates in Herbrand models of universally reductive Horn clause sets but did not formally prove its correctness. I will generalize their approach to ultimately periodic models and prove its correctness, which also implies the correctness of the original algorithm.

[1] In addition to flexibility, the emphasis in Comon and Delor's algorithm lies in a very rich constraint-based sort structure. This sort structure and the consideration of quotient algebras are orthogonal problems. To restrict the presentation of the current results to its essential kernel, I will mostly ignore the sort constraints in this paper.

This paper is structured as follows: After recalling the relevant notation in Section 2, I will present a disunification algorithm for ultimately periodic interpretations and prove it correct and terminating in Section 3. As a first application, I will show in Section 4 how to use disunification to compute the completion of predicates with respect to ultimately periodic interpretations. In Section 5, I combine results from the previous sections to prove that the satisfiability of equational formulae in ultimately periodic interpretations is decidable.

2 Preliminaries

I build on the notions of [1,8,14] and shortly recall here the most important concepts.

Terms and Formulas. Let X be an infinite set of variables. A *signature* $\Sigma = (\mathcal{S}, \mathcal{P}, \mathcal{F}, \tau)$ consists of (i) three finite sets $\mathcal{S}$, $\mathcal{P}$, $\mathcal{F}$ of *sorts*, *predicate symbols* and *function symbols* such that $\mathcal{S}$ and $\mathcal{F}$ are non-empty and X, $\mathcal{P}$ and $\mathcal{F}$ are disjoint, and (ii) a mapping τ that assigns to every variable in X a sort, to every symbol in $\mathcal{P}$ a tuple of sorts and to every symbol in $\mathcal{F}$ a non-empty tuple of sorts. Sort assignments $\tau(P) = (S_1, \ldots, S_n)$ and $\tau(f) = (S_1, \ldots, S_n, S)$ for $P \in \mathcal{P}, f \in \mathcal{F}$ and $n \geq 0$ are written as $P : S_1, \ldots, S_n$ and $f : S_1, \ldots, S_n \to S$. We assume that there are infinitely many variables and at least one term of each sort.

Let $\mathcal{T}(\mathcal{F}, X)$ be the set of all well-sorted *terms* over $\mathcal{F}$ and X defined as usual. Let $\mathcal{T}(\mathcal{F})$ be the set of all *ground terms* over $\mathcal{F}$. To improve readability, a list $t_1, \ldots, t_n$ of terms is often written as $\vec{t}$, and the n-fold application $f(\ldots(f(t))\ldots)$ of a unary function symbol f to a term t is written as $f^n(t)$.

A *predicative atom* over Σ is a well-sorted expression $P(t_1, \ldots, t_n)$, where $P : S_1, \ldots, S_n$ is a predicate symbol $t_1 : S_1, \ldots, t_n : S_n$ are terms of the corresponding sorts. An *equation* (or *disequation*, respectively) is a multiset of two terms of the same sort, usually written as $t \simeq t'$ (or $t \not\simeq t'$). The expression $t \mathrel{\dot{\simeq}} t'$ stands for either $t \simeq t'$ or $t \not\simeq t'$. An *atom* is either a predicative atom or an equation or one of the symbols $\top, \bot$ (true and false). A *literal* is an atom or a disequation or a negated predicative atom. *Formulae* are constructed from atoms by the constructors $\exists x., \forall x., \wedge, \vee$ and $\neg$. The notation $\exists \vec{x}.\phi$ is a shorthand notation for $\exists x_1. \cdots \exists x_n.\phi$, and analogously for $\forall \vec{x}.\phi$. In both cases, $\vec{x}$ may be empty. *Equational formulae* are formulae that do not contain predicative atoms. A formula is in *negation normal form* if the symbol $\neg$ appears only in literals. By pushing down all negations and eliminating double negations, each formula can be transformed into an equivalent negation normal form.

The set of variables occurring freely in a formula ϕ is denoted by $\mathrm{vars}(\phi)$. The expression $\phi|_p$ denotes the subformula at position p, and $\phi[\psi]_p$ denotes the result of replacing $\phi|_p$ by ψ. For terms t, $\mathrm{vars}(t)$, $t|_p$ and $t[t']_p$ are defined analogously.

Rewrite Systems. A *rewrite rule* is a pair (l, r) of two terms of the same sort or of two formulae, written $l \to r$, such that all variables in r also occur in l. A set of rewrite rules is called a *rewrite system*. For a given rewrite system R, a term (or formula) t *rewrites* to a term (or formula) t', written $t \to_R t'$, if $t|_p = l\sigma$

and $t' = t[r\sigma]_p$, for some rule $l \to r$ in R, position p in t, and substitution σ. A term (or formula) t is called *irreducible* or a *normal form* if there is no term (or formula) t' such that $t \to_R t'$.

A rewrite system R is *terminating* if there is no infinite chain $t_1 \to_R t_2 \to_R \ldots$; it is *convergent* if it is terminating and every term rewrites to a unique normal form.

Substitutions. A *substitution* σ is a map from X to $\mathcal{T}(\mathcal{F}, X)$ that maps each variable to a term of the same sort and acts as the identity map on all but a finite number of variables. A substitution is identified with its homomorphic extensions to terms and atoms, and with its capure-free extension to formulae. The application of a substitution σ mapping variables $x_1, \ldots, x_n$ to terms $t_1, \ldots, t_n$ to a term t (or a formula ϕ) is written as $t\sigma$ or $t\{x_1 \mapsto t_1, \ldots, x_n\}$ (or $\phi\sigma$ or $\phi\{x_1 \mapsto t_1, \ldots, x_n\}$).

Orderings. A (strict) partial ordering $>$ on a set T is a binary relation that is antisymmetric ($t_1 > t_2$ implies $t_2 \not> t_1$) and transitive ($t_1 > t_2$ and $t_2 > t_3$ implies $t_1 > t_3$). A partial ordering $>$ on a set T can be extended to a partial ordering $>^{mul}$ on multisets over T, i.e. maps from T into the non-negative integers, as follows: $M >^{mul} N$ if $M \neq N$ and whenever there is a $t \in T$ such that $N(t) > M(t)$ then $M(t') > N(t')$ for some $t' > t$. It can be extended to a partial ordering $>^{lex}$ on n-tuples over T as follows: $(t_1, \ldots, t_n) >^{lex} (t'_1, \ldots, t'_n)$ if there is an index $1 \leq i \leq n$ such that $t_j = t'_j$ for all $1 \leq j < i$ and $t_i > t'_i$.

There are several ways to extend orderings on the set $\mathcal{F}$ of function symbols to terms over $\mathcal{F}$. The ones used in this article are the recursive path ordering [11] and the associative path ordering [2]. Let $\mathcal{F}$ be a set of function symbols equipped with a partial ordering $<$ and let $stat : \mathcal{F} \to \{lex, mul\}$ be a function assigning to every function symbol either lexicographic or multiset status.

The *recursive path ordering* $>_{\text{rpo}}$ on $\mathcal{T}(\mathcal{F})$ is given as follows: For terms $t = f(t_1, \ldots, t_m), t' = g(t'_1, \ldots, t'_n) \in \mathcal{T}(\mathcal{F}, X)$, $t >_{\text{rpo}} t'$ if either (i) $t_i = t'$ or $t_i >_{\text{rpo}} t'$ for some i or (ii) $t >_{\text{rpo}} t'_i$ for all $1 \leq i \leq n$ and either $f > g$, or $f = g$ and $(t_1, \ldots, t_m) >_{\text{rpo}}^{stat(f)} (t'_1, \ldots, t'_n)$.

Let $\mathcal{F}$ contain the symbols $\wedge$ and $\vee$. For a term $t \in \mathcal{T}(\mathcal{F})$, let $t\!\downarrow$ be the normal form of t with respect to the distributivity rule $t_0 \wedge (t_1 \vee t_2) \to (t_0 \wedge t_1) \vee (t_0 \wedge t_2)$. Define the *associative path ordering* $>_{apo}$ on $\mathcal{T}(\mathcal{F})$ as follows: $t >_{\text{apo}} t'$ iff (i) $t\!\downarrow >_{\text{rpo}} t'\!\downarrow$ or (ii) $t\!\downarrow = t'\!\downarrow$ and $|t'| > |t|$.

Note that $>_{\text{apo}}$ is compatible with AC identities, monotonic (i.e. $t >_{\text{apo}} t'$ implies $u[s] >_{\text{apo}} u[t]$), has the subterm property (i.e. $t[t']_p \not\approx t'$ implies $t[t']_p >_{\text{apo}} t'$), and is well-founded.

Clauses and Herbrand Interpretations. A *clause* is a pair of multisets of atoms, written $\Gamma \to \Delta$. A clause is *Horn* if Δ contains at most one atom. It is *predicative* if all atoms in Γ and Δ are predicative. A Horn clause $C = \Gamma \to A$ is *universally reductive* if all variables that occur in C also occur in A.

A *Herbrand interpretation* $\mathcal{I}$ over the signature Σ is a set of ground atoms over Σ that is closed under rewriting with the rewrite system consisting of all

rules $t \to t'$ and $t' \to t$ such that $t \simeq t' \in \mathcal{I}$ (i.e. $\simeq$ is interpreted as a congruence in $\mathcal{I}$). A Herbrand interpretation $\mathcal{I}$ is a *model* of a set N of clauses if for every ground instance $\Gamma \to \Delta$ of a clause in N it holds that $\Gamma \subseteq \mathcal{I}$ implies $\Delta \cap \mathcal{I} \neq \emptyset$. The unique *minimal model* with respect to set inclusion of a satisfiable set N of Horn clauses is denoted by $\mathcal{I}_N$.

3 Disunification

3.1 The Disunification Algorithm PDU

Disunification provides a means to transform an equational formula ϕ into a simpler equational formula ϕ' for which satisfiability with respect to the considered interpretation is easily decidable.

Example 1. Consider the elementary case of reasoning modulo the free term algebra. In the formula $\phi = \exists y. f(x,y) \not\simeq f(s(s(x)),0) \wedge f(x,y) \not\simeq f(s(s(x)),s(0))$ over a signature containing two sorts S, T and the function symbols $0 : S$, $s : S \to S$ and $f : S, S \to T$, the disequations can be decomposed as $x \not\simeq (s(s(x))) \vee y \not\simeq 0$ and $x \not\simeq (s(s(x))) \vee y \not\simeq s(0)$, respectively. The subformula $x \not\simeq (s(s(x)))$ is equivalent to the constant $\top$, which means that the whole formula can be transformed to $\top$. Since $\top$ is trivially satisfiable with respect to $\mathcal{T}(\{0,s,f\})$, so is the initial formula ϕ.

Disunification algorithms usually have the aim to simplify a formula while preserving its solution set with respect to an interpretation.

Definition 2 (Solutions). *Let $\mathcal{I}$ be an interpretation, X a set of variables, and ϕ a formula over $\Sigma = (\mathcal{S}, \mathcal{P}, \mathcal{F}, \tau)$. The substitution set* $\mathrm{Sol}(\phi, X, \mathcal{I})$ *of* solutions of ϕ with respect to X and $\mathcal{I}$ *is defined as*

$$\mathrm{Sol}(\phi, X, \mathcal{I}) = \{\sigma \in X \to \mathcal{T}(\mathcal{F}) \mid \mathcal{I} \models \phi\sigma\} \ .$$

Two formulae ϕ, ϕ' are called equivalent with respect to $\mathcal{I}$ *if* $\mathrm{Sol}(\phi, X, \mathcal{I}) = \mathrm{Sol}(\phi', X, \mathcal{I})$*, where X consists of the free variables of ϕ and ϕ'.*

Disunification can in general not compute the solutions with respect to a general equational theory $\mathcal{T}(\mathcal{F})/_E$, where E is a set of equations: For $E = \{s(s(x)) \simeq x\}$, the formula ϕ from Example 1 is unsatisfiable in $\mathcal{T}(\{0,s,f\})/_E$. One of the problems is that $x \not\simeq (s(s(x)))$ is *not* equivalent to $\top$ in this interpretation.

Because a terminating disunification procedure with respect to an equational theory $\mathcal{T}(\mathcal{F})/_E$ results in a decision procedure for satisfiability in $\mathcal{T}(\mathcal{F})/_E$, disunification with respect to equational theories is in general not possible. I will now show that disunification can nevertheless be extended to interpretations as in the previous example, where the equalities in E are restricted to the form $s^l(x) \simeq s^k(x)$.

Definition 3 (Ultimately Periodic Interpretation). *Let $\Sigma = (\mathcal{S}, \mathcal{P}, \mathcal{F}, \tau)$ be a signature. Let $S_1, \ldots, S_n$ be n different sorts such that all ground terms*

Formulae are always kept normalized with respect to these rules.

Propagation of Negation:

$\neg\top \to \bot \qquad \neg(\phi \vee \phi') \to \neg\phi \wedge \neg\phi' \qquad \neg(\exists x.\phi) \to \forall x.\neg\phi \qquad \neg\neg\phi \to \phi$

$\neg\bot \to \top \qquad \neg(\phi \wedge \phi') \to \neg\phi \vee \neg\phi' \qquad \neg(\forall x.\phi) \to \exists x.\neg\phi$

Propagation of Truth and Falsity:

$\top \wedge \phi \to \phi \qquad \top \vee \phi \to \top \qquad \exists x.\top \to \top$

$\bot \wedge \phi \to \bot \qquad \bot \vee \phi \to \phi \qquad \exists x.\bot \to \bot$

$\phi \wedge \top \to \phi \qquad \phi \vee \top \to \top \qquad \forall x.\top \to \top$

$\phi \wedge \bot \to \bot \qquad \phi \vee \bot \to \bot \qquad \forall x.\bot \to \bot$

Quantifier Accumulation:

P1: $\forall\vec{x}.\phi[\forall\vec{y}.\phi']_p \to \forall\vec{x},\vec{y}.\phi[\phi']_p$

P2: $\exists\vec{x}.\phi[\exists\vec{y}.\phi']_p \to \exists\vec{x},\vec{y}.\phi[\phi']_p$

if $\vec{x}$ and $\vec{y}$ are not empty in P1,P2 and there is none of the symbols $\neg, \forall, \exists$ between the two melted quantifiers; if a variable of $\vec{y}$ occurs in $\phi[\top]_p$, it is renamed to avoid capturing.

Fig. 1. Normalization Rules

of sort S_i are of the form $s_i^m(0_i)$ for two function symbols $s_i, 0_i$. Let $E = \{s_1^{l_1}(x) \simeq s_1^{k_1}(x), \ldots, s_n^{l_n}(x) \simeq s_n^{k_n}(x)\}$ be a finite set of equations between terms in $S_1, \ldots, S_n$, with $l_i > k_i$ for all i. Each sort S_i, $1 \le i \le n$, is called ultimately periodic of type (k_i, l_i). *All other sorts are called* free.

Let N be a finite set of predicative Horn clauses such that $N \cup E$ is satisfiable. The minimal Herbrand model $\mathcal{I}_{N\cup E}$ of $N \cup E$ is called an ultimately periodic interpretation.

The disunification procedure for ultimately periodic interpretations is based on a disunification algorithm by Comon and Delor [8], which I will call DU. They treat the sorting discipline explicitly by enriching formulae (over an unsorted signature) with sorting constraints of the form $t \in S$, where t is a term and S is a so-called sort expression, e.g. $\mathrm{Nat} \vee f(\mathrm{Nat}, \mathrm{Nat})$. On the one hand, this allows very rich sort structures. On the other hand, it constitutes an additional considerable technical complication of the algorithm. Since multi-sorting can nicely be expressed by formulae over a sorted signature and the addition of sort constraints is a rather orthogonal problem, the variation of the algorithm used below does not rely on explicit sort constraints but on implicit well-sortedness.

Definition 4 (PDU). *Let $\mathcal{I}$ be an ultimately periodic interpretation. The* periodic disunification calculus PDU for $\mathcal{I}$ *consists of the rules in Figures 1–3.*[2]

[2] The rules of Figures 1 and 2 are essentially identical to the rules of DU. The only exceptions are Q7/8, Finite Sort Reduction, and Explosion, which differ from the formulation in [8] in that they are a straightforward combination of originally unsorted rules with rules that manipulate explicit sorting constraints. The original rule Ex1 also always required $\vec{x}$ to be non-empty. This is too weak for the completion algorithm of Section 4, which is why PDU uses a version of Ex1 by Comon and Lescanne [9].

Decomposition, Clash, and Occurrence Check:

D1: $f(u_1,\ldots,u_n)\simeq f(u'_1,\ldots,u'_n) \to u_1\simeq u'_1 \wedge \ldots \wedge u_n\simeq u'_n$

D2: $f(u_1,\ldots,u_n)\not\simeq f(u'_1,\ldots,u'_n) \to u_1\not\simeq u'_1 \vee \ldots \vee u_n\not\simeq u'_n$

C1: $f(u_1,\ldots,u_m)\simeq g(u'_1,\ldots,u'_n) \to \bot$ if $f \neq g$

C2: $f(u_1,\ldots,u_m)\not\simeq g(u'_1,\ldots,u'_n) \to \top$ if $f \neq g$

O1: $t\simeq u[t] \to \bot$ if $u[t] \neq t$

O2: $t\not\simeq u[t] \to \top$ if $u[t] \neq t$

if $f(u_1,\ldots,u_n)$, t and $u[t]$ belong to a free sort

Quantifier Elimination:

Q1: $\exists\vec{x}.\phi_1 \vee \phi_2 \to (\exists\vec{x}.\phi_1) \vee (\exists\vec{x}.\phi_2)$ if $\vec{x} \cap \mathrm{vars}(\phi_1,\phi_2) \neq \emptyset$

Q2: $\forall\vec{x}.\phi_1 \wedge \phi_2 \to (\forall\vec{x}.\phi_1) \wedge (\forall\vec{x}.\phi_2)$ if $\vec{x} \cap \mathrm{vars}(\phi_1,\phi_2) \neq \emptyset$

Q3: $\exists\vec{x},x.\phi \to \exists\vec{x}.\phi$ if $x \notin \mathrm{vars}(\phi)$

Q4: $\forall\vec{x},x.\phi \to \forall\vec{x}.\phi$ if $x \notin \mathrm{vars}(\phi)$

Q5: $\forall\vec{x},x.x\not\simeq t \vee \phi \to \forall\vec{x}.\phi\{x \mapsto t\}$ if $x \notin \mathrm{vars}(t)$

Q6: $\exists\vec{x},x.x\simeq t \wedge \phi \to \exists\vec{x}.\phi\{x \mapsto t\}$ if $x \notin \mathrm{vars}(t)$

Q7: $\forall\vec{z},\vec{x}.y_1\simeq t_1 \vee \ldots \vee y_n\simeq t_n \vee \phi \to \forall\vec{z}.\phi$

Q8: $\exists\vec{z},\vec{x}.y_1\not\simeq t_1 \wedge \ldots \wedge y_n\not\simeq t_n \wedge \phi \to \exists\vec{z}.\phi$

if in Q7 and Q8 $y_i \neq t_i$ and $\mathrm{vars}(y_i\dot{\simeq}t_i) \cap \vec{x} \neq \emptyset$ for all i and $\mathrm{vars}(\phi) \cap \vec{x} = \emptyset$ and the sorts of all variables in $\vec{x}$ contain infinitely many ground terms (in particular, all t_i are of a free sort).

Q1 and Q2 also require that no redex for P1 or P2 is created.

Finite Sort Reduction:

S1: $\forall\vec{x},x.\phi \to \forall\vec{x}.\phi\{x \mapsto t_1\} \wedge \ldots \wedge \phi\{x \mapsto t_n\}$

S2: $\exists\vec{x},x.\phi \to \exists\vec{x}.\phi\{x \mapsto t_1\} \vee \ldots \vee \phi\{x \mapsto t_n\}$

if the sort S of x is free and finite and $t_1,\ldots,t_n$ are the finitely many ground terms in S.

Distribution:

N1: $\forall\vec{x}.\phi[\phi_0 \vee (\phi_1 \wedge \phi_2)]_p \to \forall\vec{x}.\phi[(\phi_0 \vee \phi_1) \wedge (\phi_0 \vee \phi_2)]_p$

N2: $\exists\vec{x}.\phi[\phi_0 \wedge (\phi_1 \vee \phi_2)]_p \to \exists\vec{x}.\phi[(\phi_0 \wedge \phi_1) \vee (\phi_0 \wedge \phi_2)]_p$

if ϕ_0,ϕ_1,ϕ_2 are quantifier-free, $\mathrm{vars}(\phi_1) \cap \vec{x} \neq \emptyset$, ϕ_1 is not a conjunction in N1 and not a disjunction in N2 and does not contain a redex for N1 or N2, and there is no negation and no quantifier in ϕ along the path p.

Explosion:

Ex1: $\exists\vec{x}.\phi \to \bigvee_{f\in\mathcal{F}'} \exists\vec{x},\vec{x}_f.y\simeq f(\vec{x}_f) \wedge \phi\{y \mapsto f(\vec{x}_f)\}$

if y is free in ϕ and $\forall\vec{x}'.\phi'$, respectively, no other rule except Ex2 can be applied, there is in ϕ a literal $y\dot{\simeq}t$ where t contains a universally quantified variable, and $\vec{x}$ is non-empty or $\phi = \forall\vec{x}'.\phi'$. If y is of sort S, then $\mathcal{F}' \subseteq \mathcal{F}$ is the set of function symbols of sort $S_1,\ldots,S_n \to S$.

Ex2: $\forall\vec{x}.\phi \to \bigwedge_{f\in\mathcal{F}'} \forall\vec{x},\vec{x}_f.y\not\simeq f(\vec{x}_f) \vee \phi\{y \mapsto f(\vec{x}_f)\}$

if y is free in ϕ, no other rule can be applied, there is in ϕ a literal $y\simeq t$ or $y\not\simeq t$ where t contains an existentially quantified variable, and $\vec{x}$ is non-empty. If y is of sort S, then $\mathcal{F}' \subseteq \mathcal{F}$ is the set of function symbols of sort $S_1,\ldots,S_n \to S$.

Fig. 2. Rules for both Free and Ultimately Periodic Sorts

Periodic Reduction:

PR: $A[s^l(t)]_p \to A[s^k(t)]_p$

if A is an atom and $s^l(t)$ belongs to an ultimately periodic sort of type (k,l).

Periodic Decomposition:

$$\text{PD1: } s(t)\simeq s(t') \to \begin{cases} t\simeq t' & \text{if } t \text{ and } t' \text{ are ground} \\ t\simeq s^{k-1}(0) \vee t\simeq s^{l-1}(0) & \text{if } t \text{ is not ground and } s(t') = s^k(0) \\ t\simeq t' & \text{if } t \text{ is not ground and } t' \text{ is ground and } s(t') \neq s^k(0) \\ t\simeq t' \vee (t\simeq s^{k-1}(0) \wedge t'\simeq s^{l-1}(0)) & \\ \quad \vee (t\simeq s^{l-1}(0) \wedge t'\simeq s^{k-1}(0)) & \text{if } t, t' \text{ are not ground} \end{cases}$$

$$\text{PD2: } s(t)\not\simeq s(t') \to \begin{cases} t\not\simeq t' & \text{if } t \text{ and } t' \text{ are ground} \\ t\not\simeq s^{k-1}(0) \wedge t\not\simeq s^{l-1}(0) & \text{if } t \text{ is not ground and } s(t') = s^k(0) \\ t\not\simeq t' & \text{if } t \text{ is not ground and } t' \text{ is ground and } s(t') \neq s^k(0) \\ t\not\simeq t' \wedge (t\not\simeq s^{k-1}(0) \vee t'\not\simeq s^{l-1}(0)) & \\ \quad \wedge (t\not\simeq s^{l-1}(0) \vee t'\not\simeq s^{k-1}(0)) & \text{if } t, t' \text{ are not ground} \end{cases}$$

if $s(t)$ belongs to an ultimately periodic sort of type (k,l) and $s(t)\dot{\simeq}s(t')$ is irreducible by PR. For $k = 0$, the atom $\bot$ replaces $t\simeq s^{k-1}(0)$ and $\top$ replaces $t\not\simeq s^{k-1}(0)$.

Periodic Clash Test:

$$\text{PC1: } s(t)\simeq 0 \to \begin{cases} t\simeq s^{l-1}(0) & \text{if } k = 0 \text{ and } t \text{ is not ground} \\ \bot & \text{if } k > 0 \text{ or } t \text{ is ground} \end{cases}$$

$$\text{PC2: } s(t)\not\simeq 0 \to \begin{cases} t\not\simeq s^{l-1}(0) & \text{if } k = 0 \text{ and } t \text{ is not ground} \\ \top & \text{if } k > 0 \text{ or } t \text{ is ground} \end{cases}$$

if $s(t)$ belongs to an ultimately periodic sort of type (k,l) and $s(t)\dot{\simeq}0$ is irreducible by PR.

Periodic Occurrence:

$$\text{PO1: } x\simeq s^n(x) \to \begin{cases} x\simeq s^k(0) \vee \ldots \vee x\simeq s^{l-1}(0) & \text{if } l-k \text{ divides } n \\ \bot & \text{if } l-k \text{ does not divide } n \end{cases}$$

$$\text{PO2: } x\not\simeq s^n(x) \to \begin{cases} x\not\simeq s^k(0) \wedge \ldots \wedge x\not\simeq s^{l-1}(0) & \text{if } l-k \text{ divides } n \\ \top & \text{if } l-k \text{ does not divide } n \end{cases}$$

if x and $s^n(x)$ belong to an ultimately periodic sort of type (k,l) and $n > 0$.

Periodic Sort Reduction:

PS1: $\forall \vec{x}, x.\phi \to \forall \vec{x}.\phi\{x \mapsto 0\} \wedge \ldots \wedge \phi\{x \mapsto s^{l-1}(0)\}$

PS2: $\exists \vec{x}.x.\phi \to \exists \vec{x}.\phi\{x \mapsto 0\} \vee \ldots \vee \phi\{x \mapsto s^{l-1}(0)\}$

if x belongs to an ultimately periodic sort of type (k,l) and x occurs in ϕ.

Fig. 3. Rules for Ultimately Periodic Sorts

All rules can be applied at any position in a formula and they are applied modulo associativity and commutativity of $\vee$ *and* $\wedge$ *and modulo the identities* $\exists\vec{x},\vec{y}.\phi = \exists\vec{y},\vec{x}.\phi$ *and* $\forall\vec{x},\vec{y}.\phi = \forall\vec{y},\vec{x}.\phi$.

Example 5. For Example 1 and $E = \{s(s(x))\simeq x\}$, the normalization with respect to PDU runs as follows:

$$\begin{aligned}\phi &\to^*_{D2} \exists y.(x\not\simeq s(s(x)) \vee y\not\simeq 0) \wedge (x\not\simeq s(s(x)) \vee y\not\simeq s(0))\\ &\to^*_{PR} \exists y.(x\not\simeq x \vee y\not\simeq 0) \wedge (x\not\simeq x \vee y\not\simeq s(0)) \to^*_{\text{normalize}} \exists y.y\not\simeq 0 \wedge y\not\simeq s(0)\\ &\to_{PS2} (0\not\simeq 0 \wedge 0\not\simeq s(0)) \vee (s(0)\not\simeq 0 \wedge s(0)\not\simeq s(0)) \to^*_{\text{normalize}} \bot\end{aligned}$$

3.2 Correctness and Termination

I will first show that $\to_{PDU}$ rewrite steps do not change the solutions of a formula.

Theorem 6 (Correctness). *Let* $\mathcal{I}$ *be an ultimately periodic interpretation and let* ϕ,ϕ' *be two formulae such that* $\phi \to_{PDU} \phi'$. *Let* X *be a set of variables containing the free variables of* ϕ. *Then* $\mathrm{Sol}(\phi, X, \mathcal{I}) = \mathrm{Sol}(\phi', X, \mathcal{I})$.

Proof. For free sorts and all rules but Ex1, this has been proved in [8, Proposition 1]. For any sort, correctness of Ex1 has been shown in [9, Proposition 3]. For ultimately periodic sorts, correctness of all the rules in Figures 1 and 2 and Periodic Sort Reduction is follows easily.

Periodic Reduction is correct because $\mathcal{I} \models s^l(x)\simeq s^k(x)$ implies that $\mathcal{I} \models A[s^l(t)]_p\sigma \Leftrightarrow \mathcal{I} \models A[s^k(t)]_p\sigma$.

For *Periodic Decomposition*, let $\mathcal{I} \models (s(t)\simeq s(t'))\sigma$. For free sorts, this is of course equivalent to $\mathcal{I} \models (t\simeq t')\sigma$, but for periodic sorts, it is also possible that $s(t)\sigma\simeq s(t')\sigma\simeq s^k(0)$ and $t\sigma\not\simeq t'\sigma$, namely if $t\sigma\simeq s^{l-1}(0)$ and $t'\sigma\not\simeq s^{l-1}(0)$ or vice versa. In this case, $t'\sigma$ (or $t\sigma$, respectively) must be equal to $s^{k-1}(0)$ in $\mathcal{I}$. On the other hand, it is easy to verify that every solution of the reduct is also a solution of $s(t)\simeq s(t')$: If, e.g., $\mathcal{I} \models (t\simeq s^{k-1}(0) \wedge t'\simeq s^{l-1}(0))\sigma$, then $\mathcal{I} \models s(t)\simeq s^k(0)\simeq s^l(0)\simeq s(t')$.

For *Periodic Clash Test*, assume that $\mathcal{I} \models (s(t)\simeq 0)\sigma$. This is equivalent to $s(t)\sigma \to^*_E 0$. For $k \neq 0$, such a reduction is not possible. For $k = 0$, $s^l(0) \to_E 0$ implies that $\mathcal{I} \models (s(t)\simeq 0)\sigma$ is equivalent to $\mathcal{I} \models (s(t)\simeq s^l(0))\sigma$. The equivalence with $\mathcal{I} \models (t\simeq s^{l-1}(0))\sigma$ follows as for PD, using $k = 0$.

If *Periodic Occurrence* is applicable to a literal $x\dot{\simeq}s^n(x)$, $n \geq 0$, then any ground instance of the literal must be of the form $s^m(0)\dot{\simeq}s^{n+m}(0)$. Then it holds that $s^m(0) \leftrightarrow^*_E s^{n+m}(0)$ iff $l-k$ divides n and $m \geq k$.

To prove the termination of the system, I will use many ideas of the termination proof for DU from [8]. However, the original course of action makes extensive use of symmetries in the rules and cannot be taken over because the generalized Explosion creates an asymmetry.

The proof consists of two steps: First I prove that the number of rule applications between two successive applications of the Explosion rules is finite, then I show that the number of applications of the Explosion rules is also finite. Both parts of the proof rely on a transformation of formulae, during which variable occurrences are annotated by two values: The number of $\forall\exists$ quantifier alternations above the binding position and the number of variables bound by the same quantifier.

Definition 7 ($\mathrm{N}_\phi(x), \mathrm{sl}_\phi(x)$)**.** *Let ϕ be a formula in which each variable is bound at most once and let x be a variable occurring in ϕ. Associate to x and ϕ two integers $\mathrm{N}_\phi(x)$ and $\mathrm{sl}_\phi(x)$: If x is free in ϕ, define $\mathrm{sl}_\phi(x)$ to be the number of free variables in ϕ and $\mathrm{N}_\phi(x) = 0$. If x is bound in ϕ at position p, define $\mathrm{sl}_\phi(x)$ to be the number of variables bound at the same position of ϕ, and $\mathrm{N}_\phi(x)$ is one plus the number of $\forall\exists$ quantifier alternations in ϕ above p.*

This definition of $\mathrm{N}_\phi(x)$ is slightly different from the one in [8], where all quantifier alternations are counted. This difference is negligible for rules that do not introduce any quantifiers because $\mathrm{N}_\phi(x)$ (non-strictly) decreases for all variables with respect to both definitions. However, the difference is crucial when the generalized Explosion rule Ex1 is considered.

Definition 8 ($>_I$)**.** *For a signature $\Sigma = (\mathcal{S}, \mathcal{P}, \mathcal{F}, \tau)$, let the set $\mathcal{F}' = \mathcal{F} \cup \mathcal{P} \cup \{\simeq, \not\simeq, \exists, \top, \bot, \wedge, \vee, \neg, g, h, a\}$ be an extension of $\mathcal{F}$ by fresh symbols. All elements of $\mathcal{F}'$ are considered as function symbols over a single sort S. The symbols $\top, \bot, a$ are constants, $\exists, \neg, h$ are unary, and $\simeq, \not\simeq, \wedge, \vee, g$ are binary. For a formula ϕ over Σ, that is assumed to be renamed such that each variable is bound at most once in ϕ, inductively define a function $\mathrm{I}_\phi(.)$ from formulae over Σ and terms over $\mathcal{F}$ to terms over $\mathcal{F}'$ as follows. First every universal quantifier occurrence $\forall\vec{x}.\psi'$ in the argument is replaced by $\neg\exists\vec{x}.\neg\psi'$ and the result is normalized by the Normalization rules from Figure 1 except $\neg(\exists\vec{x}.\phi) \not\to \forall\vec{x}.\neg\phi$. Finally*

$$
\begin{aligned}
\mathrm{I}_\phi(\psi_1 \circ \psi_2) &= \mathrm{I}_\phi(\psi_1) \circ \mathrm{I}_\phi(\psi_2) && \text{for } \circ \in \{\wedge, \vee\}\\
\mathrm{I}_\phi(\exists x_1, \ldots, x_n.\psi) &= \exists^n(\mathrm{I}_\phi(\psi))\\
\mathrm{I}_\phi(\neg P(\vec{t})) &= \mathrm{I}_\phi(P(\vec{t}))\\
\mathrm{I}_\phi(\neg\psi) &= \neg\mathrm{I}_\phi(\psi) && \text{if } \psi \text{ is not an atom}\\
\mathrm{I}_\phi(\circ(t_1, \ldots, t_n)) &= \circ(\mathrm{I}_\phi(t_1), \ldots, \mathrm{I}_\phi(t_n)) && \text{for } \circ \in \mathcal{F} \cup \mathcal{P} \cup \{\top, \bot, \simeq, \not\simeq\}\\
\mathrm{I}_\phi(x) &= g(h^{\mathrm{N}_\phi(x)}(a), h^{\mathrm{sl}_\phi(x)}(a))
\end{aligned}
$$

Assume the partial ordering $\neg > g > h > f > a > \simeq > \not\simeq > P > \top > \bot > \exists > \wedge > \vee$ on $\mathcal{F}'$, where f is any symbol in $\mathcal{F}$ and P is any symbol in $\mathcal{P}$, and symbols within $\mathcal{F}$ and within $\mathcal{P}$, respectively, are incomparable. The symbols $\wedge, \vee, \simeq, \not\simeq$ have multiset status and g and all symbols in $\mathcal{P}$ and $\mathcal{F}$ have lexicographic status. Define an ordering $>_I$ by $\phi >_I \phi' \iff \mathrm{I}_\phi(\phi) >_{apo} \mathrm{I}_{\phi'}(\phi')$.

Example 9. If t is a term that contains at least one variable and t' is a ground term, then $\mathrm{I}_\phi(t) >_{\mathrm{apo}} \mathrm{I}_{\phi'}(t')$ for any formulae ϕ, ϕ'. In fact, $\mathrm{I}_\phi(t)$ contains a

subterm of the form $g(h^n(a), h^m(a))$ which is already greater than $\mathrm{I}_{\phi'}(t')$ because the latter term contains only symbols from $\mathcal{F} \cup \{\simeq, \not\simeq\}$ and g is greater than all these symbols.

I will show that the combination of a rule application and the following normalization decreases $>_I$ for all rules except Explosion. Since normalization is obviously terminating, this implies that only a finite number of transformation steps can occur between two successive explosions.

Proposition 10. *There is no infinite chain $\phi_1, \phi_2, \phi_3, \ldots$ of formulae such that $\phi_1 \to_{PDU} \phi_2 \to_{PDU} \phi_3 \to_{PDU} \cdots$ and none of the steps is an Explosion.*

Proof. If $\phi \to_{PDU} \phi'$ by application of a non-Explosion rule implies $\phi >_I \phi'$, the proposition follows because the associative path ordering $>_I$ is well-founded.

For the rules in Figures 1 and 2, $\phi >_I \phi'$ was proved by Comon and Delor [8, Lemmas 3–9]. *Periodic Sort Reduction* is a syntactic variation of Finite Sort Reduction, so the proofs for both rules are identical.

For *Periodic Reduction* $A[t] \to A[t']$, note that $\mathrm{N}_\phi(x) = \mathrm{N}_{\phi'}(x)$ and $\mathrm{sl}_\phi(x) = \mathrm{sl}_{\phi'}(x)$. This implies that $\mathrm{I}_{\phi'}(t')$ is a strict subterm of $\mathrm{I}_\phi(t)$. So $\mathrm{I}_\phi(t) >_{\mathrm{apo}} \mathrm{I}_{\phi'}(t')$, i.e. $\mathrm{I}_\phi(\phi) >_{\mathrm{apo}} \mathrm{I}_{\phi'}(\phi')$ by monotonicity of $>_{\mathrm{apo}}$.

For *Periodic Clash Test* and $k > 0$ or ground t, the proposition follows from the ordering $\doteq > \top, \bot$ on the top symbols and monotonicity of $>_{\mathrm{apo}}$. For $k = 0$ and non-ground t, consider the rewriting $s(t) \doteq 0 \to t \doteq s^{l-1}(0)$. By monotonicity of $>_{\mathrm{apo}}$, I have to show that $\mathrm{I}_\phi(s(t) \doteq 0) >_{\mathrm{apo}} \mathrm{I}_{\phi'}(t \doteq s^{l-1}(0))$, which reduces after application of the definition of I to $s(\mathrm{I}_\phi(t)) \doteq 0 >_{\mathrm{apo}} \mathrm{I}_{\phi'}(t) \doteq s^{l-1}(0)$. Again, N and sl do not change in this step, and so $\mathrm{I}_\phi(t) = \mathrm{I}_{\phi'}(t)$.

By definition of $>_{\mathrm{apo}}$, it suffices to show that the three relations $s(\mathrm{I}_\phi(t)) \simeq 0 >_{\mathrm{apo}} \mathrm{I}_\phi(t)$ and $s(\mathrm{I}_\phi(t)) \simeq 0 >_{\mathrm{apo}} s^{l-1}(0)$ and $\{s(\mathrm{I}_\phi(t)), 0\} >_{\mathrm{apo}}^{mul} \{\mathrm{I}_\phi(t), s^{l-1}(0)\}$ hold. All three relations follow from $s(\mathrm{I}_\phi(t)) >_{\mathrm{apo}} s^{l-1}(0)$ (c.f. Example 9), and the subterm property of $>_{\mathrm{apo}}$.

For *Periodic Decomposition*, it suffices to show $\mathrm{I}_\phi(s(t) \doteq s(t')) >_{\mathrm{apo}} \mathrm{I}_{\phi'}(A)$ for all newly introduced atoms A (remember that $\doteq > \wedge$ and $\doteq > \vee$). Clearly $\mathrm{I}_\phi(s(t) \doteq s(t')) >_{\mathrm{apo}} \mathrm{I}_{\phi'}(t \doteq t')$ by the subterm property of $>_{\mathrm{apo}}$. For all other atoms, the relation $\mathrm{I}_\phi(s(t) \doteq s(t')) >_{\mathrm{apo}} \mathrm{I}_{\phi'}(A)$ follows as in the case of Periodic Clash Test and $k = 0$.

For *Periodic Occurrence*, the proposition follows from monotonicity and the ordering $\doteq > \top, \bot$ on the top symbols if the literal is replaced by $\top$ or $\bot$; the argument is analogous to the one for Periodic Decomposition if the literal is replaced by a conjunction or disjunction.

An application of Explosion to a formula does not reduce the formula with respect to $>_I$. Because of this, a different ordering is needed to handle explosions. This ordering will be a lexicographic combination of orderings based on $\mathrm{I}_\phi(\psi)$.

Lemma 11. *For a formula ϕ, let $H(\phi)$ be one plus the maximal number of $\forall\exists$ quantifier alternations along a path in ϕ. Then every application of a rule in PDU non-strictly decreases H.*

Proof. The only rules that can add new quantifier symbols to a formula are Q1/2 and the Sort Reduction, Distribution and Explosion rules. Q1/2, Sort Reduction and Distribution only duplicate existing quantifiers and cannot introduce new quantifier alternations. Both Ex1 applied at an existential quantifier position and Ex2 obviously do not introduce a new quantifier alternation. Because Explosion only applies to a formula to which the rule P1 is not applicable, Ex1 also does not introduce a new $\forall\exists$ quantifier alternation if it is applied to at a universal quantifier position.

Definition 12 ($>_\omega$). *Let ϕ, ψ be formulae over the signature $\Sigma = (\mathcal{S}, \mathcal{P}, \mathcal{F}, \tau)$, let ω be a new function symbol, and let $i \geq 1$. Define the formula $\Omega_{\phi,i}(\psi)$ as the normal form of ψ under the following rewrite system that is confluent modulo associativity and commutativity [8, Lemma 12]:*

$$
\begin{aligned}
\circ(\omega, \ldots, \omega) &\to \omega && \text{if } \circ \in \mathcal{F} \cup \mathcal{P} \cup \{\simeq, \not\simeq, \vee, \neg\} \\
x &\to \omega && \text{if } \mathrm{N}_\phi(x) < i \\
\psi_1 \wedge \omega &\to \omega \\
\neg\psi_1 \vee \omega &\to \neg\psi_1 \\
\forall\vec{x}.\psi_1 &\to \neg\exists\vec{x}.\neg\psi_1 \\
\exists x.\psi_1 &\to \psi_1 && \text{if } x \notin \mathrm{vars}(\psi_1) \\
(\exists x.\psi_1) \vee \omega &\to \exists x.\psi_1 && \text{if } \exists x.\psi_1 \text{ is irreducible}
\end{aligned}
$$

Let $\Omega_i(\phi) = \Omega_{\phi,i}(\phi)$. Extend the previous ordering from Definition 8 by $\top > \omega > \exists$. Moreover, define partial orderings $>_i$ and $>_\omega$ by $\phi >_i \phi' \iff \mathrm{I}_\phi(\Omega_i(\phi)) >_{apo} \mathrm{I}_{\phi'}(\Omega_i(\phi'))$ and $\phi >_\omega \phi' \iff$ there is an index i such that $\mathrm{I}_\phi(\Omega_j(\phi)) = \mathrm{I}_{\phi'}(\Omega_j(\phi'))$ for all $j > i$ and $\phi >_i \phi'$.

Lemma 11 implies $\Omega_j(\phi_n) = \omega$ for all formulae ϕ_n in a derivation $\phi_1 \to_{PDU} \phi_2 \to_{PDU} \cdots$ and all j above the fixed boundary $H(\phi_1)$. Hence $>_\omega$ is well-founded on $\{\phi_1, \phi_2, \ldots\}$ as a lexicographic combination of the well-founded partial orderings $>_{H(\phi_1)}, \ldots, >_2, >_1$.

Lemma 13. *Let $\phi \to \phi'$ not be an Explosion. Then $\phi \geq_i \phi'$ for all $i \geq 1$.*

Proof. For the rules in Figures 1 and 2, this was proved by Comon and Delor [8, Lemma 15].

For *Periodic Reduction* $A[t]_p \to A[t']_p$, $\mathrm{N}_x(\phi) = \mathrm{N}_x(\phi')$ implies that $\Omega_{\phi',i}(t')$ is a subterm of $\Omega_{\phi,i}(t)$.

For *Periodic Clash Test* and $k > 0$ or ground t, note that the top symbol of $\Omega_{\phi,i}(s(t) \doteq 0)$ is either $\doteq$ or ω, and both are larger than $\bot$. For $k = 0$ and non-ground t, consider the rewriting $s(t) \doteq 0 \to t \doteq s^{l-1}(0)$. Again, N does not change in this step, and so applying $\Omega_{\phi,i}$ and $\Omega_{\phi',i}$, respectively, yields the normal form ω for both sides if t reduces to ω, and the literals $s(t') \doteq \omega$ and $t' \doteq \omega$ if t reduces to some other term t'. In both cases, $\mathrm{I}_\phi(\Omega_{\phi,i}(s(t) \doteq 0)) \geq_{\mathrm{apo}} \mathrm{I}_{\phi'}(\Omega_{\phi',i}(t \doteq s^{l-1}(0)))$.

For *Periodic Decomposition*, let $\psi = s(t) \doteq s(t')$ reduce to ψ'. If t and t' are ground, then $\Omega_{\phi,i}(\psi) = \Omega_{\phi',i}(\psi') = \omega$. So let t not be ground. It suffices

to show that $\mathrm{I}_\phi(\Omega_{\phi,i}(\psi))$ is at least as large as $\mathrm{I}_\phi(\Omega_{\phi,i}(A)) = \mathrm{I}_{\phi'}(\Omega_{\phi',i}(A))$ for all newly introduced literals A. $\mathrm{I}_\phi(\Omega_{\phi,i}(\psi)) \geq_{\mathrm{apo}} \mathrm{I}_\phi(\Omega_{\phi,i}(t \doteq t'))$ holds because $\mathrm{I}_\phi(\Omega_{\phi,i}(\psi)) \geq_{\mathrm{apo}} \mathrm{I}_\phi(\Omega_{\phi,i}(t))$ and $\mathrm{I}_\phi(\Omega_{\phi,i}(\psi)) \geq_{\mathrm{apo}} \mathrm{I}_\phi(\Omega_{\phi,i}(t'))$ and also $\{\mathrm{I}_\phi(\Omega_{\phi,i}(s(t))), \mathrm{I}_\phi(\Omega_{\phi,i}(s(t')))\} \geq^{mul}_{\mathrm{apo}} \{\mathrm{I}_\phi(\Omega_{\phi,i}(t)), \mathrm{I}_\phi(\Omega_{\phi,i}(t'))\}$, all by the subterm property of $>_{\mathrm{apo}}$. The ground term $s^{l-1}(0)$ reduces to the minimal term ω, and so $\mathrm{I}_\phi(\Omega_{\phi,i}(\psi)) \geq_{\mathrm{apo}} \mathrm{I}_\phi(\Omega_{\phi,i}(t \doteq t')) \geq_{\mathrm{apo}} \mathrm{I}_\phi(\Omega_{\phi,i}(t \doteq s^{l-1}(0)))$. The same holds for the literals $t' \doteq s^{l-1}(0)$, $t \doteq s^{k-1}(0)$, and $t' \doteq s^{k-1}(0)$.

For *Periodic Occurrence*, the proposition follows from $\Omega_{\phi',i}(\top) = \Omega_{\phi',i}(\bot) = \omega$ and the ordering $\doteq, \omega \geq \omega$ if the literal is replaced by $\top$ or $\bot$; otherwise it follows as for Periodic Decomposition.

The ordering $>_\omega$ is still not strong enough to show directly that the Explosion rules are decreasing in some sense. In fact, if $\phi \to_{Ex1/2} \phi'$, then $\phi' >_\omega \phi$. However, the non-Explosion rule applications following such a step revert this increase. The proofs of the following Lemmas 15–17 are almost identical to the respective proofs for DU [8, Lemmas 16–18]; they are presented anyway because it is there that the difference in definitions of $\mathrm{N}_\phi(x)$ is of importance.

Definition 14 (Explosion Level L_ϕ). *Let $\phi \to_{Ex1/2} \phi'$ be an explosion using the literal $y \doteq t[z]$. The* explosion level *L_ϕ is defined as $L_\phi = \mathrm{N}_\phi(z)$.*

By the control on Ex1/2, it holds that $\mathrm{N}_\phi(z) \geq \mathrm{N}_\phi(y)$.

Lemma 15. *Let $\phi \to_{Ex1} \bigvee_f \phi_f$ or $\phi \to_{Ex2} \bigwedge_f \phi_f$ be an explosion at the root position of ϕ and let $\phi_f \to^*_{PDU \setminus \{Ex1,Ex2\}} \psi_f$ such that every ψ_f is irreducible with respect to $PDU \setminus \{Ex1, Ex2\}$. Then for every f there is an index $i \geq L_\phi$ such that (i) $\mathrm{I}_{\phi_f}(\Omega_j(\phi_f)) = \mathrm{I}_{\psi_f}(\Omega_j(\psi_f))$ for every $j > i$ and (ii) $\phi_f >_i \psi_f$.*

Proof. Let i be the largest index such that $\mathrm{I}_{\phi_f}(\Omega_i(\phi_f)) \neq \mathrm{I}_{\psi_f}(\Omega_i(\psi_f))$. By the side conditions of Ex1/2, i exists and $i \geq L_\phi$. By Lemma 13, $\mathrm{I}_{\phi_f}(\Omega_i(\phi_f)) \geq \mathrm{I}_{\psi_f}(\Omega_i(\psi_f))$ and since, by definition of i, they are distinct, $\phi_f >_i \psi_f$ follows.

Lemma 16. *Let $\phi \to_{Ex1} \bigvee_f \phi_f$ or $\phi \to_{Ex2} \bigwedge_f \phi_f$ be an explosion at the root position of ϕ and let $\bigvee_f \phi_f \to^*_{PDU \setminus \{Ex1,Ex2\}} \psi$, or $\bigwedge_f \phi_f \to^*_{PDU \setminus \{Ex1,Ex2\}} \psi$, respectively, such that ψ is irreducible with respect to $PDU \setminus \{Ex1, Ex2\}$. Then $\phi >_\omega \psi$.*

Proof. I show the proposition for Ex1; the case of Ex2 is analogous. By definition of Ω and L_ϕ, $\mathrm{I}_\phi(\Omega_i(\phi)) = \mathrm{I}_{\phi_f}(\Omega_i(\phi_f))$ for every f and every $i \geq L_\phi$.[3] No rule can affect two disjuncts at the same time unless one becomes equal to $\top$ and the whole problem reduces to $\top$ and hence obviously decreases wrt. $>_\omega$. So let $\psi = \bigvee_f \psi_f$ and $\phi_f \to^*_{PDU \setminus \{Ex1,Ex2\}} \psi_f$. Then $\phi >_\omega \psi_f$ for every f because of Lemma 15. Let i_f be the index from this lemma for ϕ_f and let i be the maximum of the i_f. The top symbol of ϕ can only be a quantifier, i.e. the top of $\mathrm{I}_\phi(\Omega_i(\phi))$ is either $\exists\neg$ or $\neg$ and hence greater than the top symbol of $\mathrm{I}_\psi(\Omega_i(\psi))$. Hence $\phi >_\omega \psi$.

[3] For Ex1, this does not hold for the original definition of $\mathrm{N}_\phi(x)$, because an additional quantifier level may have been introduced!

Lemma 17. *Let* $\phi \to_{Ex1/2} \phi' \to^*_{PDU\setminus\{Ex1,Ex2\}} \psi \to_{Ex1/2} \cdots$. *Then* $\phi >_\omega \psi$.

Proof. Lemma 16 shows this when Ex1 or Ex2 is applied at the top position of ϕ. At other positions, no rule can modify the context of the exploded subformula, unless this subformula itself reduces to $\top$ or $\bot$. But if this happens, the part which disappears is deeper than the part which is modified and $\phi >_\omega \psi$.

Theorem 18 (Termination). *There is no infinite chain* $\phi_1, \phi_2, \phi_3, \ldots$ *of formulae such that* $\phi_1 \to_{PDU} \phi_2 \to_{PDU} \phi_3 \to_{PDU} \cdots$.

Proof. Because of Lemma 17, each chain can only contain a finite number of Explosion steps. Because of Proposition 10, there can also only be finitely many successive non-Explosion steps.

4 Predicate Completion

When an interpretation is given as the minimal model $\mathcal{I}_N$ of a set N of Horn clauses, it is often of interest to enrich N to a set N' in such a way that N' does not have any Herbrand models other than $\mathcal{I}_N$. The key to this enrichment is the so-called *completion* of predicates [4]: For each predicate P, the set N' has to describe for which arguments P does *not* hold in $\mathcal{I}_N$.

Example 19. If $N_{\mathrm{Even}} = \{\mathrm{Even}(0),\ \mathrm{Even}(x) \to \mathrm{Even}(s(s(x)))\}$ describes the even numbers over the signature $\Sigma_{\mathrm{Even}} = (\{\mathrm{Nat}\}, \{\mathrm{Even}\}, \{s, 0\}, \{s : \mathrm{Nat} \to \mathrm{Nat},\ 0 : \mathrm{Nat},\ \mathrm{Even} : \mathrm{Nat}\})$, then $\mathrm{Even}(s^n(0))$ holds in the minimal model $\mathcal{I}_{N_{\mathrm{Even}}}$ if and only if n is an even number. Let N'_{Even} contain N_{Even} and the additional clauses $\mathrm{Even}(s(0)) \to$ and $\mathrm{Even}(s(s(x))) \to \mathrm{Even}(x)$. Then $\mathcal{I}_{N_{\mathrm{Even}}}$ is the only Herbrand model of N'_{Even} over Σ_{Even}.

For predicative Horn clause sets N, $\simeq$ is interpreted as syntactic equality in $\mathcal{I}_N$. Comon and Nieuwenhuis [10, Section 7.3] used this fact to develop a predicate completion procedure for predicative and universally reductive Horn clause sets based on a disunification algorithm. They did not, however, give a formal proof of the correctness of the procedure. In this section, I will extend the predicate completion algorithm to clause sets describing ultimately periodic interpretations and prove its correctness and termination.

Definition 20 (PC). *Let* $\Sigma = (\mathcal{S}, \mathcal{P}, \mathcal{F}, \tau)$ *be a signature and let* $\mathcal{I}_{N\cup E}$ *be an ultimately periodic interpretation over* Σ *as in Definition 3 and let all clauses in* N *be universally reductive.*

The predicate completion algorithm PC *works as follows:*

(1) For $P \in \mathcal{P}$, *let* $N_P \subseteq N$ *be the set of clauses in* N *of the form* $\Gamma \to P(\vec{t})$. *Combine all these clauses into the single formula* $\forall\vec{x}.(\phi_P \to P(\vec{x}))$ *where*

$$\phi_P = \exists\vec{y}. \bigvee_{\Gamma\to P(\vec{t})\in N_P} (x_1 \simeq t_1 \wedge \ldots \wedge x_n \simeq t_n \wedge \bigwedge_{A\in\Gamma} A) \ ,$$

the y_i *are the variables appearing in* N_P, *and the* x_j *are fresh variables.*

(2) In the interpretation $\mathcal{I}_N$, the formula $\forall\vec{x}.(\phi_P \rightarrow P(\vec{x}))$ is equivalent to $\forall\vec{x}.(\neg\phi_P \rightarrow \neg P(\vec{x}))$. Transform $\neg\phi_P$ using the algorithm PDU with respect to $\mathcal{I}_{N\cup E}$ into an equivalent formula ϕ'_P that does not contain any universal quantifiers.
(3) Write the formula $\forall\vec{x}.(\phi'_P \rightarrow \neg P(\vec{x}))$ as a set finite N'_P of clauses.
(4) Let N' be the union of N and all sets N'_P, $P \in \mathcal{P}$.

Clark [4] showed that, given that the transformation of $\neg\phi_P$ to ϕ'_P is correct and $\forall\vec{x}.(\phi'_P \rightarrow \neg P(\vec{x}))$ does correspond to a set of clauses, N' is a completion of N. So the critical steps are (2) and (3): It is neither obvious that the universal quantifiers can be eliminated from $\neg\phi_P$, nor is it obvious that, once the universal quantifiers are gone, the result can be written as a finite set of clauses. To address the second issue, I can make use of the fact that certain normal forms with respect to PDU can be transformed into a particularly simple form:

Definition 21 (Solved Form). *A formula ϕ is a* solved form *if $\phi = \top$, $\phi = \bot$, or ϕ is a disjunction $\phi = \phi_1 \vee \ldots \vee \phi_m$ and each ϕ_j is of the shape*

$$\phi_j = \exists\vec{y}.x_{i_1}\simeq t_1 \wedge \ldots \wedge x_{i_n}\simeq t_n \wedge A_1 \wedge \ldots \wedge A_k \wedge \neg B_1 \wedge \ldots \wedge \neg B_{k'} \wedge z_1 \not\simeq t'_1 \wedge \ldots \wedge z_l \not\simeq t'_l\ ,$$

where $x_{i_1}, \ldots, x_{i_n}$ occur only once in ϕ_j, the A_i and B_i are predicative atoms, the z_i are variables and $z_i \neq t'_i$.

Lemma 22. *Let $\mathcal{I}$ be an ultimately periodic interpretation and let ϕ be a formula in negation normal form that does not contain any universal quantifiers. Then ϕ can be transformed into an equivalent solved form. If furthermore ϕ is irreducible by PDU, then all bound variables of the solved from are of infinite sorts.*[4]

Proof. A terminating algorithm to perform the conversion for equational formulae was given by Comon and Lescanne [9, Section 5.3] and Comon and Delor [8, Sections 6.2 and 6.3]. This algorithm preserves the solutions of a formula with respect to every interpretation $\mathcal{I}$. It mainly uses variations of the rules Q1, Q6 and N2 and the rule $x\simeq t \wedge x\simeq u \rightarrow x\simeq t \wedge t\simeq u$ (where t is not a variable). Predicative atoms $P(\vec{t})$ can be encoded as equations $f_P(\vec{t})\simeq$true, where f_P and true are new function symbols.

If ϕ is irreducible by PDU, then it is in particular irreducible by Finite and Periodic Sort Reduction and does not contain any bound variables of a finite free or ultimately periodic sort. Since the transformation algorithm does not introduce any new quantifiers, this invariant is preserved throughout the transformation.

So the formula $\forall\vec{x}.(\phi'_P \rightarrow \neg P(\vec{x}))$ is equivalent to either $\top$ or $\forall x.\neg P(\vec{x})$ (i.e. to the empty clause set or to the single clause $\{P(\vec{x}) \rightarrow\}$), or to $\forall\vec{x}.\bigwedge_j(\phi'_j \rightarrow \neg P(\vec{x}))$, and each conjunct can equivalently be written as a clause of the form $A_1, \ldots, A_k, P(\vec{x})\{x_{i_1} \mapsto t_1, \ldots, x_{i_n} \mapsto t_n\} \rightarrow B_1, \ldots, B_k, z_1\simeq t'_1, \ldots, z_l\simeq t'_l$.

[4] The second part of this lemma is not relevant to the current considerations, but it will be used in Section 5.

To prove that the universal quantifiers can in fact be eliminated from $\neg\phi_P$, I will examine an invariant that holds for $\neg\phi_P$ (Lemma 23), is preserved during the application of PDU (Lemma 24), and holds only for normal forms that do not contain universal quantifiers (Lemma 27).

Invariant 1. *Let $\phi\downarrow$ be the normal form of a formula ϕ under the Normalization rules, Decomposition, Periodic Decomposition and Distribution. Consider the following properties of ϕ:*

(1) No subformula of $\phi\downarrow$ of the form $\forall\vec{x}.\phi'$ contains a quantifier.
(2) Universally quantified variables occur in $\phi\downarrow$ only in predicative literals or in disequations $t[x]\not\approx t'$ where all variables in t' are free or existentially quantified.

For every predicative literal occurrence A_x in $\phi\downarrow$ containing a universally quantified variable x, there is a subformula of $\phi\downarrow$ of the form $A_x \vee B_x \vee \phi_x$ where B_x is a disequation containing x.

Lemma 23. *Let N be a universally reductive Horn clause set and let ϕ_P be defined as in Definition 20. Then the invariant holds for $\neg\phi_P$.*

Proof. The normal form $(\neg\phi_P)\downarrow$ of $\neg\phi_P$ is

$$(\neg\phi_P)\downarrow = \forall\vec{y}. \bigwedge_{\Gamma\to P(\vec{t})\in N_P} (x_1\not\approx t_1 \vee \ldots \vee x_n\not\approx t_n \vee \bigvee_{A\in\Gamma} \neg A) \ .$$

Invariant 1 holds because there are no nested quantifiers in $(\neg\phi_P)\downarrow$. Invariant 2 holds because all clauses in N are universally reductive, and so every variable that occurs in a predicative literal A also occurs in one of the disequations $x_i\not\approx t_i$ in the same conjunct of $(\neg\phi_P)\downarrow$.

Lemma 24. *Let $\phi \to_{PDU} \phi'$. If ϕ satisfies the invariant then so does ϕ'.*

Proof. Invariant 1: The only rule to introduce a possibly critical quantifier symbol is the rule Ex1 applied to a subformula of the form $\forall\vec{x}.\psi'$, i.e. $\phi[\forall\vec{x}.\psi']_p \to \phi[\bigvee_{f\in\mathcal{F}'} \exists\vec{x}_f.y\simeq f(\vec{x}_f)\wedge\forall\vec{x}.\psi'\{y\mapsto f(\vec{x}_f)\}]_p$. If the new existential quantifier $\exists\vec{x}_f$ is in the scope of a universal quantifier in ϕ' and $\phi'\downarrow$, then so was the original universal quantifier $\forall\vec{x}$ in ϕ and $\phi\downarrow$.

Invariant 2: This invariant can only be destroyed by introducing new universally quantified variables into a literal, by disconnecting A_x and B_x, or by altering a disequation B_x.

It is easy to see that all rules that do not replace variables or reduce B_x preserve the invariant.

If C2, O2, PC2 or PO2 reduces B_x to $\top$, then the whole subformula $A_x \vee B_x \vee \phi_x$ is reduced to $\top$ by the normalization rules, i.e. the invariant is maintained. Alternatively, PC2 can alter B_x by $s(t[x])\not\approx 0 \to t\not\approx s^{l-1}(0)$. By PD2 and Normalization, this literal either reduces to $\top$ or to a formula ψ consisting of disjunctions, conjunctions and literals such that ψ contains x in disequations $t''[x]\not\approx t'$ as required for the invariant. In the former case, the whole disjunction

$A_x \vee \psi \vee \phi_x$ reduces to $\top$; in the latter, the distribution rule is applicable because the disequation is in the scope of a universal quantifier and Invariant 1 guarantees that there is no existential quantifier in between. Distribution brings $A_x \vee \psi \vee \phi_x$ into the form $\psi'_1 \wedge \ldots \wedge \psi'_n$ where each ψ'_i is of the form $A_x \vee x \not\simeq t' \vee \psi''_i$, i.e. the invariant is preserved.

PO2 is not applicable to a disequation B_x because one side contains a universal quantifier and the other one does not.

The Sort Reduction rules only replace variables by ground terms and thus are harmless.

If Q5: $\phi[\forall \vec{x}, x.x \not\simeq t \vee \psi]_p \to \phi[\forall \vec{x}.\psi\{x \mapsto t\}]_p$ works on a universally quantified variable x (and B_x may or may not be $x \not\simeq t$), then every occurrence of x is replaced by a term that, by Invariant 1, does not contain any universally quantified variables, which maintains the invariant.

When Q6: $\phi[\exists \vec{y}, y.y \simeq t \wedge \psi]_p \to \phi[\exists \vec{y}.\psi\{y \mapsto t\}]_p$ is applied, then t contains only free or existentially quantified variables because of Invariant 1. Again, the invariant is not affected.

An Explosion Ex1: $\phi[\exists \vec{x}.\psi]_p \to \phi[\bigvee_{f \in \mathcal{F}'} \exists \vec{x}, \vec{x}_f.y \simeq f(\vec{x}_f) \wedge \psi\{y \mapsto f(\vec{x}_f)\}]_p$ replaces a variable y that is free in ψ. By Invariant 1, this variable is existentially quantified or free in ϕ and the replacement $f(\vec{x}_f)$ contains only existentially quantified variables. The same holds for the second version of Ex1.

An Explosion Ex2: $\phi[\forall \vec{x}.\psi]_p \to \phi[\bigwedge_{f \in \mathcal{F}'} \forall \vec{x}, \vec{x}_f.y \not\simeq f(\vec{x}_f) \vee \psi\{y \mapsto f(\vec{x}_f)\}]_p$ cannot be executed because it relies on the existence of a variable that is existentially quantified in ψ, which is excluded by Invariant 1.

Lemma 25. *Let $\phi[\psi]_p$ be a formula containing a subformula ψ that is reducible by PDU. Then ϕ is also reducible by PDU.*

Proof. The only contextual conditions in the control part of the rules are irreducibility conditions; the only situations where the control can prevent a reduction $\phi[\psi]_p \to_{PDU} \phi[\psi']_p$ when $\psi \to_{PDU} \psi'$ are such that there is another redex in ϕ.[5]

Lemma 26. *A normal form with respect to PDU does not contain nested quantifiers.*

Proof. Comon and Delor showed that normal forms with respect to DU do not contain nested quantifiers [8, Lemma 22]. Every well-sorted formula that is reducible by DU is also reducible by PDU.

Lemma 27. *A normal form with respect to PDU is free of universal quantifiers if it (i) fulfills the invariant or (ii) does not contain any predicative literals.*

Proof. Consider a formula that fulfills the invariant or does not contain any predicative atoms but contains a subformula $\forall \vec{x}.\phi$. Assume that the whole formula and hence, by Lemma 25, $\forall \vec{x}.\phi$ is irreducible by PDU. This will result in a contradiction.

[5] This proof is identical to the proof of the corresponding lemma for DU [8, Lemma 21].

By Lemma 26, ϕ can only be a disjunction, a conjunction, or a literal. If it is a conjunction, then Q2 is applicable. The case that ϕ is a literal arises as the special case of a disjunction with only one disjunct. So let ϕ be a disjunction. If a disjunct contains a variable from $\vec{x}$, then it must be a literal: If it is a conjunction then rule N2 applies, and the top symbol cannot be a quantifier because of Lemma 26.

So ϕ can be written as $\phi = \phi_1 \vee \ldots \vee \phi_n \vee \phi'$, where the ϕ_i are literals containing universal variables and ϕ' does not contain any universal variables. Because of the Normalization rules, each ϕ_i can only be a predicative literal or an equational literal.

ϕ_i cannot be a disequation: Because of the decomposition rules, it would be either $x \not\simeq t$ or $y \not\simeq t[x]$ where $x \in \vec{x}$ and y is free in ϕ. In the former case, Q5 is applicable, in the latter Ex1.

ϕ_i can only be predicative in variant (i) of the lemma, and then Invariant 2 requires that one of the other ϕ_j is a disequation containing a variable from $\vec{x}$. This possibility has already been refuted.

If ϕ_i is an equation, the decomposition rules only leave the possibilities $x \simeq t$ and $y \simeq t[x]$ where $x \in \vec{x}$ and y is free in ϕ. In the latter case, Ex1 applies, so only $x \simeq t$ is possible.

All in all, ϕ is of the form $\phi = x_{i_1} \simeq t_1 \vee \ldots \vee x_{i_n} \simeq t_n \vee \phi'$ and Q6 applies.

Lemmas 23, 24, and 27(i) imply that PDU eliminates the universal quantifiers from $\neg\phi_P$. Together with Lemma 22 and the correctness and termination of disunification (Theorems 6 and 18), this implies the applicability and the termination of the predicate completion algorithm PC.

Theorem 28 (Predicate Completion). *Let $\Sigma = (\mathcal{S}, \mathcal{P}, \mathcal{F}, \tau)$ be a signature and let $\mathcal{I}_{N \cup E}$ be an ultimately periodic interpretation over Σ as in definition 3.*

If N is universally reductive, then PC terminates on $\mathcal{I}_{N \cup E}$ with a completion of N.

Example 29. For Example 19, $\phi_{\mathrm{Even}} = x \simeq 0 \vee \exists y.y \simeq s(s(x)) \wedge \mathrm{Even}(y)$ and a normal form of $\neg\phi_{\mathrm{Even}}$ with respect to PDU and an equation $E = \{s^l(x) \simeq x\}$ is

$$\phi'_{\mathrm{Even}} = \begin{cases} x \not\simeq s^{l-1}(0) \wedge \neg\mathrm{Even}(x) & \text{for } l \in \{1,2\} \\ (x \simeq s(0) \wedge \neg\mathrm{Even}(s^{l-1}(0))) & \\ \quad \vee (\exists z.x \simeq s(s(z)) \wedge \neg\mathrm{Even}(z) \wedge z \not\simeq s^{l-2}(0)) & \text{for } l > 2\ . \end{cases}$$

This formula corresponds to the following clauses in N'_{Even}:

$$\begin{cases} \mathrm{Even}(x) \to \mathrm{Even}(x), x \simeq s^{l-1}(0) & \text{for } l \in \{1,2\} \\ \mathrm{Even}(s(0)) \to \mathrm{Even}(s^{l-1}(0)) & \\ \quad \text{and } \mathrm{Even}(s(s(z))) \to \mathrm{Even}(z), z \simeq s^{l-2}(0) & \text{for } l > 2 \end{cases}$$

5 Decidability of the Satisfiability of Equational Formulae

It is obvious that the decidability of the satisfiability of equational formulae in models over free sorts implies the decidability of the satisfiability of equational formulae in models over both free and ultimately periodic sorts: As there

are only finitely many non-equivalent terms of each ultimately periodic sort, variables of such sorts can be eliminated by immediate grounding of quantified variables (e.g. with the rules PS1, PS2) and free variables (by the transformation $\phi \to \phi\{x \mapsto 0\} \vee \ldots \vee \phi\{x \mapsto s^{l-1}(0)\}$) in both model description and query formula. If then every term of an ultimately periodic sort is replaced by a unique representative of its equivalence class, the original disunification algorithm decides satisfiability.

The eager grounding leads, however, to a huge blow-up of both the set N of clauses describing the model and the query formula. The combination of the results of the previous sections provides a more flexible decision procedure for the satisfiability of an equational formula ϕ in a given ultimately periodic interpretation $\mathcal{I}$ that allows to postpone grounding and thus improve the practical efficiency of the decision procedure:

Lemma 30. *Let $\mathcal{I}$ be an ultimately periodic interpretation and let $\phi \neq \bot$ be a solved form in which all variables are of infinite sorts and all atoms are equational. Then ϕ has at least one solution in $\mathcal{I}$.*

Proof. For $\phi = \top$, this is trivial. Otherwise consider a disjunct as in Definition 21. Since all variables appearing in $z_1 \not\approx t'_1 \wedge \ldots \wedge z_l \not\approx t'_l$ are of infinite sorts, these disequations have a common solution σ (cf. e.g. [8, Lemma 2]). Then $\{x_{i_1} \mapsto t_1\sigma, \ldots, x_{i_n} \mapsto t_n\sigma\}$ is a solution of the considered disjunct (and hence also of ϕ) with respect to $\mathcal{I}$ and the variables $x_{i_1}, \ldots, x_{i_n}$. This solution can trivially be extended to a solution for all free variables of ϕ.

Theorem 31 (Decidability of Satisfiability). *Let $\mathcal{I}$ be an ultimately periodic interpretation over the signature Σ. Then satisfiability in $\mathcal{I}$ of an equational formula ϕ over Σ is decidable.*

Proof. Let $\vec{x}$ be the free variables in ϕ of finite (e.g. ultimately periodic) sorts. Then ϕ is satisfiable in $\mathcal{I}$ iff $\exists\vec{x}.\phi$ is. Using PDU, $\exists\vec{x}.\phi$ can be transformed into an equivalent normal form ϕ'. Because ϕ does not contain any predicative atoms, neither does ϕ'. Lemma 27(ii) implies that ϕ' is free of universal quantifiers and Lemma 22 states that it can be transformed into an equivalent solved form. By Lemma 30, it is decidable whether this solved form (and hence also ϕ) has a solution in $\mathcal{I}$.

The same does not hold for formulae containing predicative atoms, as e.g. the formula $\forall y.P(y)$ is a normal form with respect to PDU if y is of a free and infinite sort.

6 Implementation

A single-sorted version of the presented algorithms has been implemented on top of the automated theorem prover Spass [25]. Spass is a theorem prover for full first-order logic with equality and a number of non-classical logics and provides support for first-order formulas with both equational and predicative atoms.

I represent ultimate periodicity equation using a globally accessible data structure that provides easy access to the defining equation $s^l(x) \simeq s^k(x)$ itself as well as the values k and l. Formulas are represented using the term module of SPASS. To improve the efficiency of the implementation of PDU, I adapted the data structures to grant instant access to regularly needed information. In particular, I used parent links to efficiently navigate through formulas. Since some rules like Q5/6, Sort Reduction and Explosion perform non-local changes in the formula, it is not possible to normalize the formula in one run, e.g. bottom up, but multiple successive runs are needed. To keep track of which subformulas are normalized, every subformula is additionally equipped with a normality marker that is set when the subformula is normalized and only reset for subformulas in which a replacement takes place and avoids traversing subtrees where all reduction steps have already been performed.

The implementation of PC consists of several steps: First the input is parsed, the ultimate periodicity information extracted and the input clauses partitioned with respect to the predicate of their maximal succedent atom. For each predicate, the formula ϕ_P is then created and the implementation of PDU is used to compute a normalization of $\neg\phi_P$. The solved form computation is performed by the same implementation in a post-processing step. Finally, the resulting completion is extracted in a straightforward way.

Since the completed clause set can directly be used as an input for SPASS, this implementation effectively allows SPASS to perform minimal model reasoning with a first-order machinery. The implementation has been tested on and optimized with the help of the problems in the TPTP library [24]. To be able to use a broad range of problems from the different problem classes in this library, I allowed every first-order problem from this library as an input. To make them match the applicability restrictions of PC, I eliminated those clauses in each problem that were not Horn or not obviously universally reductive or that contained equations. The implementation is available from the SPASS homepage (`www.spass-prover.org/prototypes/`).

7 Conclusion

In this article, I have presented the disunification algorithm PDU for ultimately periodic interpretations, proved its correctness and termination, and used the algorithm to establish the decidability of satisfiability for equational formulae in such interpretations. I have also presented the disunification-based predicate completion algorithm PC for ultimately periodic interpretations defined by universally reductive Horn clauses. This extends work by Comon and Delor [8] and Comon and Nieuwenhuis [10]. An instance of both PDU and PC has been implemented as part of a predicate completion procedure on top of the first-order theorem prover SPASS [25]. Predicate completion can easily be generalized to non-Horn clauses (cf. [14]). However, this requires additional machinery that is beyond the scope of this paper, like a saturation concept, an adapted notion of universally reductive clauses and the consideration of non-unique minimal models.

The algorithms have been implemented. To my best knowledge, this provides the first publicly available implementation of disunification and predicate completion.

The approach of widening the scope of disunification that is most closely connected to mine is the one by Comon [6], who adapted a predecessor of the algorithm by Comon and Delor to interpretations $\mathcal{T}(\mathcal{F})/_E$ where E is a so-called quasi-free or compact axiomatization. This approach only results in a terminating procedure if (among other conditions) all equations have depth at most 1. Another way to extend disunification is by considering not only term algebras but more general structures. Comon and Lescanne [9] and Maher [20] considered rational tree algebras and Comon and Delor [8] extended disunification to classes of non-regular tree algebras.

One possible application of both disunification for ultimately periodic interpretations in general and predicate completion in particular is resolution-based inductive theorem proving: Ultimately periodic interpretations appear naturally as models of formulae of propositional linear time linear logic [19]. Moreover, it should be possible to generalize recent decidability results by Horbach and Weidenbach for inductive reasoning with respect to models represented e.g. by disjunctions of implicit generalizations [14,15].

Acknowledgements. I want to thank Michel Ludwig and Ullrich Hustadt for sharing their interest in ultimately periodic interpretations, which initiated this research. This work is supported by the German Transregional Collaborative Research Center SFB/TR 14 AVACS.

References

1. Baader, F., Snyder, W.: Unification theory. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, vol. 1, ch. 8, pp. 445–532. Elsevier and MIT Press (2001)
2. Bachmair, L., Plaisted, D.A.: Associative path orderings. In: Hsiang, J. (ed.) RTA 1995. LNCS, vol. 914, pp. 241–254. Springer, Heidelberg (1995)
3. Caferra, R., Zabel, N.: A method for simultaneous search for refutations and models by equational constraint solving. Journal of Symbolic Computation 13(6), 613–642 (1992)
4. Clark, K.L.: Negation as failure. In: Gallaire, H., Minker, J. (eds.) Logic and Data Bases, pp. 293–322. Plenum Press, New York (1977)
5. Colmerauer, A.: Equations and inequations on finite and infinite trees. In: FGCS, vol. 5, pp. 85–99 (1984)
6. Comon, H.: Unification et Disunification: Théorie et applications. PhD thesis, Institut National Polytechnique de Grenoble (July 1988)
7. Comon, H.: Disunification: A survey. In: Lassez, J.-L., Plotkin, G. (eds.) Computational Logic: Essays in Honor of Alan Robinson, pp. 322–359. MIT Press, Cambridge (1991)
8. Comon, H., Delor, C.: Equational formulae with membership constraints. Information and Computation 112(2), 167–216 (1994)
9. Comon, H., Lescanne, P.: Equational problems and disunification. Journal of Symbolic Computation 7(3-4), 371–425 (1989)

10. Comon, H., Nieuwenhuis, R.: Induction = I-axiomatization + first-order consistency. Information and Computation 159(1/2), 151–186 (2000)
11. Dershowitz, N.: Orderings for term-rewriting systems. Theoretical Computer Science 17, 279–301 (1982)
12. Fermüller, C.G., Leitsch, A.: Hyperresolution and automated model building. Journal of Logic and Computation 6(2), 173–203 (1996)
13. Fernández, M.: Narrowing based procedures for equational disunification. Applicable Algebra in Engineering, Communication and Computing 3, 1–26 (1992)
14. Horbach, M., Weidenbach, C.: Decidability results for saturation-based model building. In: Schmidt, R.A. (ed.) Automated Deduction – CADE-22. LNAI, vol. 5663, pp. 404–420. Springer, Heidelberg (2009)
15. Horbach, M., Weidenbach, C.: Deciding the inductive validity of $\forall\exists^*$ queries. In: Grädel, E., Kahle, R. (eds.) CSL 2009. LNCS, vol. 5771, pp. 332–347. Springer, Heidelberg (2009)
16. Jouannaud, J.-P., Kirchner, C.: Solving equations in abstract algebras: A rule-based survey of unification. In: Lassez, J.-L., Plotkin, G. (eds.) Computational Logic - Essays in Honor of Alan Robinson, pp. 257–321. MIT Press, Cambridge (1991)
17. Jouannaud, J.-P., Kirchner, H.: Completion of a set of rules modulo a set of equations. SIAM J. Comput. 15(4), 1155–1194 (1986)
18. Lassez, J.-L., Maher, M.J., Marriott, K.: Unification revisited. In: Boscarol, M., Levi, G., Aiello, L.C. (eds.) Foundations of Logic and Functional Programming. LNCS, vol. 306, pp. 67–113. Springer, Heidelberg (1988)
19. Ludwig, M., Hustadt, U.: Resolution-based model construction for PLTL. In: Lutz, C., Raskin, J.-F. (eds.) TIME, pp. 73–80. IEEE Computer Society, Los Alamitos (2009)
20. Maher, M.J.: Complete axiomatizations of the algebras of finite, rational and infinite trees. In: LICS, pp. 348–357. IEEE Computer Society, Los Alamitos (1988)
21. Mal'cev, A.I.: Axiomatizable classes of locally free algebra of various type. In: Wells, B.F. (ed.) The Metamathematics of Algebraic Systems: Collected Papers 1936–1967, ch. 23, pp. 262–281. North Holland, Amsterdam (1971)
22. Plotkin, G.: Building in equational theories. In: Meltzer, B.N., Michie, D. (eds.) Machine Intelligence, vol. 7, pp. 73–90. Edinburgh University Press (1972)
23. Robinson, J.A.: A machine-oriented logic based on the resolution principle. Journal of the ACM 12(1), 23–41 (1965)
24. Sutcliffe, G.: The TPTP problem library and associated infrastructure. Journal of Automomated Reasoning 43(4), 337–362 (2009)
25. Weidenbach, C., Dimova, D., Fietzke, A., Kumar, R., Suda, M., Wischnewski, P.: SPASS version 3.5. In: Schmidt, R.A. (ed.) Automated Deduction – CADE-22. LNCS, vol. 5663, pp. 140–145. Springer, Heidelberg (2009)

Synthesis of Trigger Properties*

Orna Kupferman[1] and Moshe Y. Vardi[2]

[1] Hebrew University, School of Engineering and Computer Science, Jerusalem 91904, Israel
orna@cs.huji.ac.il
http://www.cs.huji.ac.il/~orna
[2] Rice University, Department of Computer Science, Houston, TX 77251-1892, U.S.A.
vardi@cs.rice.edu
http://www.cs.rice.edu/~vardi

Abstract. In automated synthesis, we transform a specification into a system that is guaranteed to satisfy the specification. In spite of the rich theory developed for temporal synthesis, little of this theory has been reduced to practice. This is in contrast with model-checking theory, which has led to industrial development and use of formal verification tools. We address this problem here by considering a certain class of PSL properties; this class covers most of the properties used in practice by system designers. We refer to this class as the class of trigger properties.

We show that the synthesis problem for trigger properties is more amenable to implementation than that of general PSL properties. While the problem is still 2EXPTIME-complete, it can be solved using techniques that are significantly simpler than the techniques used in general temporal synthesis. Not only can we avoid the use of Safra's determinization, but we can also avoid the use of progress ranks. Rather, the techniques used are based on classical subset constructions. This makes our approach amenable also to symbolic implementation, as well as an incremental implementation, in which the specification evolves over time.

1 Introduction

One of the most significant developments in the area of program verification over the last two decades has been the development of algorithmic methods for verifying temporal specifications of *finite-state* programs; see [CGP99]. A frequent criticism against this approach, however, is that verification is done *after* significant resources have already been invested in the development of the program. Since programs invariably contain errors, verification simply becomes part of the debugging process. The critics argue that the desired goal is to use the specification in the program development process in order to guarantee the design of correct programs. This is called *program synthesis*.

The classical approach to program synthesis is to extract a program from a proof that the specification is satisfiable [BDF+04, EC82, MW80, MW84]. In the late 1980s, several researchers realized that the classical approach to program synthesis is well suited to *closed* systems, but not to *open* (also called *reactive*) systems [ALW89, Dil89, PR89a]. In reactive systems, the program interacts with the environment, and a correct program

* Work supported in part by NSF grant CCF-0728882, by BSF grant 9800096, and by gift from Intel. Part of this work was done while the second author was on sabbatical at the Hebrew University of Jerusalem.

E.M. Clarke and A. Voronkov (Eds.): LPAR-16, LNAI 6355, pp. 312–331, 2010.

should satisfy the specification with respect to all environments. Accordingly, the right way to approach synthesis of reactive systems is to consider the situation as a (possibly infinite) game between the environment and the program. A correct program can be then viewed as a winning strategy in this game. It turns out that satisfiability of the specification is not sufficient to guarantee the existence of such a strategy. Abadi et al. called specifications for which a winning strategy exists *realizable*. Thus, a strategy for a program with inputs in I and outputs in O maps finite sequences of inputs (words in $(2^I)^*$ – the actions of the environment so far) to an output in 2^O – a suggested action for the program. Thus, a strategy can be viewed as a labeling of a tree with directions in 2^I by labels in 2^O.

The traditional algorithm for finding a winning strategy transforms the specification into a parity automaton over such trees such that a program is realizable precisely when this tree automaton is nonempty, i.e., it accepts some infinite tree [PR89a]. A finite generator of an infinite tree accepted by this automaton can be viewed as a finite-state program realizing the specification. This is closely related to the approach taken in [BL69, Rab72] in order to solve Church's *solvability problem* [Chu63]. In [KV00, PR89b, WD91, Var95] it was shown how this approach to program synthesis can be carried out in a variety of settings.

In spite of the rich theory developed for program synthesis, and recent demonstrations of its applicability [BGJ+07], little of this theory has been reduced to practice. Some people argue that this is because the realizability problem for linear-temporal logic (LTL) specifications is 2EXPTIME-complete [PR89a, Ros92], but this argument is not compelling. First, experience with verification shows that even nonelementary algorithms can be practical, since the worst-case complexity does not arise often. For example, while the model-checking problem for specifications in second-order logic has nonelementary complexity, the model-checking tool MONA [EKM98, Kla98] successfully verifies many specifications given in second-order logic. Furthermore, in some sense, synthesis is not harder than verification. This may seem to contradict the known fact that while verification is "easy" (linear in the size of the model and at most exponential in the size of the specification [LP85]), synthesis is hard (2EXPTIME-complete). There is, however, something misleading in this fact: while the complexity of synthesis is given with respect to the specification only, the complexity of verification is given with respect to the specification and the program, which can be much larger than the specification. In particular, it is shown in [Ros92] that there are temporal specifications for which every realizing program must be at least doubly exponentially larger than the specifications. Clearly, the verification of such programs is doubly exponential in the specification, just as the cost of synthesis.

As argued in [KPV06], we believe that there are two reasons for the lack of practical impact of synthesis theory. The first is algorithmic and the second is methodological. Consider first the algorithmic problem. The traditional approach for constructing tree automata for realizing strategies uses determinization of Büchi automata. Safra's determinization construction has been notoriously resistant to efficient implementations [ATW05, THB95] [1], results in automata with a very complicated state space, and

[1] An alternative construction is equally hard [ATW05]. Piterman's improvement of Safra includes the tree structures that proved hard to implement [Pit07].

involves the parity acceptance condition. The best-known algorithms for parity-tree-automata emptiness [Jur00] are nontrivial already when applied to simple state spaces. Implementing them on top of the messy state space that results from determinization is highly complex, and is not amenable to optimizations and a symbolic implementation. In [KV05c, KPV06], we suggested an alternative approach, which avoids determinization and circumvents the parity condition. While the Safraless approach is much simpler and can be implemented symbolically, it is based on progress ranks. The need to manipulate ranks requires multi-valued data structures, making the symbolic implementation difficult [TV07, DR09]. This is in contrast with symbolic implementations of algorithms based on the subset construction without ranks, which perform well in practice [MS08a, MS08b].

Another major issue is methodological. The current theory of program synthesis assumes that one gets a comprehensive set of temporal assertions as a starting point. This cannot be realistic in practice. A more realistic approach would be to assume an *evolving* formal specification: temporal assertions can be added, deleted, or modified. Since it is rare to have a complete set of assertions at the very start of the design process, there is a need to develop *incremental* synthesis algorithms. Such algorithms can, for example, refine designs when provided with additional temporal properties.

One approach to tackle the algorithmic problems has been to restrict the class of allowed specification. In [AMPS98], the authors studied the case where the LTL formulas are of the form $\Box p$, $\Diamond p$, $\Box \Diamond p$, or $\Diamond \Box p$.[2] In [AT04], the authors considered the fragment of LTL consisting of boolean combinations of formulas of the form $\Box p$, as well as a richer fragment in which the $\bigcirc$ operator is allowed. Since the games corresponding to formulas of these restricted fragments are simpler, the synthesis problem is much simpler; it can be solved in PSPACE or EXPSPACE, depending on the specific fragment. Anther fragment of LTL, termed $GR(1)$, was studied in [PPS06]. In the $GR(1)$ fragment (generalized reactivity(1)) the formulas are of the form $(\Box \Diamond p_1 \wedge \Box \Diamond p_2 \wedge \cdots \Box \Diamond p_m) \rightarrow (\Box \Diamond q_1 \wedge \Box \Diamond q_2 \wedge \cdots \Box \Diamond q_n)$, where each p_i and q_i is a Boolean combination of atomic propositions. It is shown in [PPS06] that for this fragment, the synthesis problem can be solved in EXPTIME, and with only $O((mn \cdot 2^{|AP|})^3)$ symbolic operations, where AP is the set of atomic propositions.

We continue the approach on special classes of temporal properties, with the aim of focusing on properties that are used in practice. We study here the synthesis problem for TRIGGER LOGIC. Modern industrial-strength property-specification languages such as Sugar [BBE+01], ForSpec [AFF+02], and the recent standards PSL [EF06], and SVA [VR05] include regular expressions. TRIGGER LOGIC is a fragment of these logics that covers most of the properties used in practice by system designers. Technically, TRIGGER LOGIC consists of positive Boolean combinations of assertions about regular events, connected by the usual regular operators as well as temporal implication, $\mapsto$ ("triggers"). For example, the TRIGGER LOGIC formula $(\mathbf{true}[*]; req; ack) \mapsto (\mathbf{true}[*]; grant)$ holds in an infinite computation if every request that is immediately followed by an acknowledge is eventually followed by a grant. Also, the TRIGGER LOGIC formula $(\mathbf{true}[*]; err) \mapsto !(\mathbf{true}[*]; ack)$ holds in a computation if once an error is detected, no acks can be sent.

[2] The setting in [AMPS98] is of real-time games, which generalizes synthesis.

We show that TRIGGER LOGIC formulas can be translated to deterministic Büchi automata using the two classical subset constructions: the determinization construction of [RS59] and the break-point construction of [MH84]. Accordingly, while the synthesis problem for TRIGGER LOGIC is still 2EXPTIME-complete, our synthesis algorithm is significantly simpler than the one used in general temporal synthesis. We show that this also yields several practical consequences: our approach is quite amenable to symbolic implementation, it can be applied to evolving specifications in an incremental fashion, and it can also be applied in an assume-guarantee setting. We believe that the simplicity of the algorithm and its practical advantages, coupled with the practical expressiveness of TRIGGER LOGIC, make an important step in bridging the gap between temporal-synthesis theory and practice.

2 Trigger Logic

The introduction of temporal logic to computer science, in [Pnu77], was a watershed point in the specification of reactive systems, which led to the development of model checking [CGP99]. The success of model checking in industrial applications led to efforts to develop "industrial" temporal logics such as Sugar [BBE+01] and ForSpec [AFF+02], as well as two industry-standard languages, PSL [EF06] and SVA [VR05].

A common feature of these languages is the use of regular expressions to describe temporal patterns. For example, the regular expression $request; \mathbf{true}^+; grant; \mathbf{true}^+; ack$ describes an occurrence of $request$, followed by $grant$, followed by ack, where these events are separated by nonempty intervals of arbitrary length. The advantage of using regular expressions over the classical temporal operators of LTL is that it avoids the need for deep nesting of untils. For that reason, regular expressions have proved to be quite popular with verification engineers,[3] to the point that the regular layer is that main layer of SVA [VR05]. The key observation is that a very large fraction of temporal properties that arise in practice can be expressed in the form of $e_1 \mapsto e_2$ or $e_1 \mapsto !e_2$ (we generally use PSL syntax in this paper), which means that an e_1 pattern should, or should not, be followed by an e_2 pattern; see, for example, [SDC01]. As an example, consider the property: "If a snoop hits a modified line in the L1 cache, then the next transaction must be a snoop writeback." It can be expressed using the PSL formula

$$(\mathbf{true}[*]; snoop \&\& modified) \mapsto (!trans\_start[*]; trans\_start \&\& writeback).$$

The extension of LTL with regular expressions is called RELTL [BFG+05]. Here we study TRIGGER LOGIC– the fragment of RELTL consisting of positive Boolean combinations of formulas of the form $e_1 \mapsto e_2$ or $e_1 \mapsto !e_2$. We now describe this logic formally.

Let Σ be a finite *alphabet*. A *finite word* over Σ is a (possibly empty) finite sequence $w = \sigma_0 \cdot \sigma_1 \cdots \sigma_n$ of concatenated letters in Σ. The length of a word w is denoted by $|w|$. The symbol ϵ denotes the empty word. We use $w[i, j]$ to denote the subword $\sigma_i \cdots \sigma_j$ of w. If $i > j$, then $w[i, j] = \epsilon$. *Regular Expressions* (REs) define languages by inductively applying union, concatenation, and repetition operators. Formally, an RE over an alphabet Σ is one of the following.

[3] See http://www.cs.rice.edu/~vardi/accelera-properties.pdf.

- $\emptyset$, ϵ, or σ, for $\sigma \in \Sigma$.
- $r_1|r_2$, $r_1; r_2$, $r[*]$, or $r[+]$, for REs r, r_1, and r_2.

We use $L(r)$ to denote the language that r defines. For the base cases, we have $L(\emptyset) = \emptyset$, $L(\epsilon) = \{\epsilon\}$, and $L(\sigma) = \{\sigma\}$. The operators $|$, $;$, $[*]$, and $[+]$ stand for union, concatenation, possibly empty repetition, and strict repetition, respectively. Formally,

- $L(r_1|r_2) = L(r_1) \cup L(r_2)$.
- $L(r_1; r_2) = \{w_1; w_2 : w_1 \in L(r_1) \text{ and } w_2 \in L(r_2)\}$.
- Let $r^0 = \{\epsilon\}$ and let $r^i = r^{i-1}; r_1$, for $i \geq 1$. Thus, $L(r^i)$ contains words that are the concatenation of i words in $L(r_1)$. Then, $L(r[*]) = \bigcup_{i \geq 0} r^i$ and $L(r[+]) = \bigcup_{i \geq 1} r^i$.

For a set X of elements, let $\mathcal{B}(X)$ denote the set of all Boolean functions $b : 2^X \rightarrow \{\mathbf{true}, \mathbf{false}\}$. In practice, members of $\mathcal{B}(X)$ are expressed by Boolean expressions over X, using with disjunction ($||$), conjunction ($\&\&$), and negation ($!$). Let $\mathcal{B}^+(X)$ be the restriction of $\mathcal{B}(X)$ to positive Boolean functions. That is, functions induced by formulas constructed from atoms in X with disjunction and conjunction, and we also allow the constants $\mathbf{true}$ and $\mathbf{false}$. For a function $b \in \mathcal{B}(X)$ and a set $Y \subseteq X$, we say that Y satisfies b if assigning $\mathbf{true}$ to the elements in Y and $\mathbf{false}$ to the elements in $X \setminus Y$ satisfies b.

For a set AP of atomic propositions, let $\Sigma = \mathcal{B}(AP)$, and let $\mathcal{R}$ be a set of atoms of the form r or $!r$, for a regular expression r over Σ. For example, for $AP = \{p, q\}$, the set $\mathcal{R}$ contains the regular expression $(p|!q)[*]|(p; p)$ and also contains $!((p|!q)[*]|(p; p))$.

The linear temporal logic TRIGGER LOGIC is a formalism to express temporal implication between regular events. We consider TRIGGER LOGIC in a positive normal form, where formulas are constructed from atoms in $\mathcal{R}$ by means of Boolean operators, regular expressions, and temporal implication ($\mapsto$). The syntax of TRIGGER LOGIC is defined as follows (we assume a fixed set AP of atomic propositions, which induces the fixed sets Σ and $\mathcal{R}$).

1. A *regular assertion* is a positive Boolean formula over $\mathcal{R}$.
2. A *trigger formula* is of the form $r \mapsto \theta$, for a regular expression r over Σ and a regular assertion θ.
3. A TRIGGER LOGIC formula is a positive Boolean formula over trigger formulas.

Intuitively, $r \mapsto \theta$ asserts that all prefixes satisfying r are followed by a suffix satisfying θ. The linear temporal logic TRIGGER LOGIC is a formalism to express temporal implication between regular events. For example, $(\mathbf{true}[*]; p) \mapsto (\mathbf{true}[*]; q)$ is regular formula, equivalent to the LTL formula $G(p \rightarrow Fq)$. We use $\theta(e_1, \ldots, e_k, !e'_1, \ldots, !e'_{k'})$ to indicate that the regular assertion θ is over the regular expressions $e_1, \ldots, e_k$ and the negations of the regular expressions $e'_1, \ldots, e'_{k'}$. Note that we do not allow nesting of $\mapsto$ in our formulas.

The semantics of TRIGGER LOGIC formulas is defined with respect to infinite words over the alphabet 2^{AP}. Consider an infinite word $\pi = \pi_0, \pi_1, \ldots \in (2^{AP})^\omega$. For indices i and j with $0 \leq i \leq j$, and a language $L \subseteq \Sigma^*$, we say that $\pi_i, \ldots, \pi_{j-1}$ *tightly satisfies* L, denoted $\pi, i, j, \models L$, if there is a word $b_0 \cdot b_1 \cdots b_{j-1-i} \in L$ such that for all $0 \leq k \leq j - 1 - i$, we have that $b_i(\pi_{i+k}) = \mathbf{true}$. Note that when $i = j$, the interval $\pi_i, \ldots, \pi_{j-1}$ is empty, in which case $\pi, i, j \models L$ iff $\epsilon \in L$. For an index $i \geq 0$ and a language $L \subseteq \Sigma^*$, we say that $\pi_i, \pi_{i+1}, \ldots$ satisfies L, denoted $\pi, i \models L$, if $\pi, i, j \models L$

for some $j \geq i$. Dually, $\pi_i, \pi_{i+1}, \ldots$ satisfies $!L$, denoted $\pi, i \models !L$, if there is no $j \geq i$ such that $\pi, i, j \models L$. Note that $\pi, i \models !L$ iff $\pi, i \not\models L$; note that both are different from $\pi, i \models \Sigma^* \setminus L$. For a regular assertion θ, we say that $\pi_i, \pi_{i+1}, \ldots$ satisfies θ, denoted $\pi, i \models \theta$ if there is a set $Y \subseteq \mathcal{R}$ such that Y satisfies θ, $\pi, i \models L(r)$ for all $r \in Y$, and $\pi, i \models !L(r)$ for all $!r \in Y$,

We can now define the semantics of the $\mapsto$ operator.

– $\pi, i \models (r \mapsto \theta)$ if for all $j \geq i$ such that $\pi, i, j \models L(r)$, we have $\pi, j \models \theta$.

For a TRIGGER LOGIC formula ψ, a path π satisfies ψ in index i, denoted $\pi, i \models \psi$, if π, i satisfies a set X of regular formulas such that X satisfies ψ. Finally, π satisfies ψ if π satisfies ψ in index 0.

Thus, the formula $(\mathbf{true}[*]; p) \mapsto (\mathbf{true}[*]; q)$ holds in an infinite word $\pi \in 2^{\{p,q\}}$ if every p is eventually followed by q. Indeed, for all $j \geq 0$, if $\pi, 0, j \models L(\mathbf{true}[*]; p)$, which holds iff $\pi_j \models p$, then $\pi, j \models \mathbf{true}[*]; q$. The latter holds iff there is $k \geq j$ such that $\pi, j, k \models \mathbf{true}[*]; q$, which holds iff $\pi_k \models q$.

3 Automata on Words and Trees

An *automaton on infinite words* is a tuple $\mathcal{A} = \langle \Sigma, Q, q_0, \rho, \alpha \rangle$, where Σ is the input alphabet, Q is a finite set of states, $\rho : Q \times \Sigma \to 2^Q$ is a transition function, $q_0 \in Q$ is an initial state, and α is an acceptance condition (a condition that defines a subset of Q^ω). Intuitively, $\rho(q, \sigma)$ is the set of states that $\mathcal{A}$ can move into when it is in state q and it reads the letter σ. Since the transition function of $\mathcal{A}$ may specify many possible transitions for each state and letter, $\mathcal{A}$ is not *deterministic*. If ρ is such that for every $q \in Q$ and $\sigma \in \Sigma$, we have that $|\rho(q, \sigma)| = 1$, then $\mathcal{A}$ is a deterministic automaton. We extend ρ to sets of states in the expected way, thus, for $S \subseteq Q$, we have that $\rho(S, \sigma) = \bigcup_{s \in S} \rho(s, \sigma)$.

A *run* of $\mathcal{A}$ on w is a function $r : \mathbb{N} \to Q$ where $r(0) = q_0$ (i.e., the run starts in the initial state) and for every $l \geq 0$, we have $r(l+1) \in \rho(r(l), \sigma_l)$ (i.e., the run obeys the transition function). In automata over finite words, acceptance is defined according to the last state visited by the run. When the words are infinite, there is no such thing "last state", and acceptance is defined according to the set $Inf(r)$ of states that r visits *infinitely often*, i.e., $Inf(r) = \{q \in Q : \text{for infinitely many } l \in \mathbb{N}, \text{we have } r(l) = q\}$. As Q is finite, it is guaranteed that $Inf(r) \neq \emptyset$. The way we refer to $Inf(r)$ depends on the acceptance condition of $\mathcal{A}$. In *Büchi automata*, $\alpha \subseteq Q$, and r is accepting iff $Inf(r) \cap \alpha \neq \emptyset$. Dually, In *co-Büchi automata*, $\alpha \subseteq Q$, and r is accepting iff $Inf(r) \cap \alpha = \emptyset$.

Since $\mathcal{A}$ is not deterministic, it may have many runs on w. In contrast, a deterministic automaton has a single run on w. There are two dual ways in which we can refer to the many runs. When $\mathcal{A}$ is an *existential* automaton (or simply a *nondeterministic* automaton, as we shall call it in the sequel), it accepts an input word w iff there exists an accepting run of $\mathcal{A}$ on w.

Automata can also run on trees. For our application, we only need deterministic Büchi tree automata. Given a set D of directions, a *D-tree* is a set $T \subseteq D^*$ such that if $x \cdot c \in T$, where $x \in D^*$ and $c \in D$, then also $x \in T$. If $T = D^*$, we say that T is a full D-tree. The elements of T are called *nodes*, and the empty word ε is the *root* of T.

For every $x \in T$, the nodes $x \cdot c$, for $c \in D$, are the *successors* of x. Each node D^* has a *direction* in D. The direction of the root is d_0, for some designated $d_0 \in D$, called the *root direction*. The direction of a node $x \cdot d$ is d. We denote by $dir(x)$ the direction of node x. A *path* π of a tree T is a set $\pi \subseteq T$ such that $\varepsilon \in \pi$ and for every $x \in \pi$, either x is a leaf or there exists a unique $c \in D$ such that $x \cdot c \in \pi$. Given an alphabet Σ, a *Σ-labeled D-tree* is a pair $\langle T, \tau \rangle$ where T is a tree and $\tau : T \to \Sigma$ maps each node of T to a letter in Σ.

A *deterministic Büchi tree automaton* is $\mathcal{A} = \langle \Sigma, D, Q, \delta, q_0, \alpha \rangle$, where Σ, Q, q_0, and α, are as in Büchi word automata, and $\delta : Q \times \Sigma \to Q^{|D|}$ is a (deterministic) transition function. Intuitively, in each of its transitions, $\mathcal{A}$ splits into $|D|$ copies, each proceeding to a subtree whose root is the successor of the current node. For a direction $d \in D$, having $\delta(q, \sigma)(d) = q'$ means that if $\mathcal{A}$ is now in state q and it reads the letter σ, then the copy that proceeds to direction d moves to state q'.

Formally, a *run* of $\mathcal{A}$ on an input Σ-labeled D-tree $\langle D^*, \tau \rangle$, is a Q-labeled tree $\langle D^*, r \rangle$ such that $r(\epsilon) = q_0$ and for every $x \in D^*$, and direction $d \in D$, we have that $r(x \cdot d) = \delta(r(x), \tau(x))(d)$. If, for instance, $D = \{0, 1\}$, $r(0) = q_2$, $\tau(0) = a$, and $\delta(q_2, a)(0) = q_1$ and $\delta(q_2, a)(1) = q_2$, then $r(0 \cdot 0) = q_1$ and $r(0 \cdot 1) = q_2$. Given a run $\langle D^*, r \rangle$ and a path $\pi \subset D^*$, we define $inf(r|\pi) = \{q \in Q \ : \text{for infinitely many } x \in \pi, \text{we have } r(x) = q\}$. A run r is accepting iff for all paths $\pi \subset D^*$, we have $inf(r|\pi) \cap \alpha \neq \emptyset$. That is, iff for each path $\pi \subset D^*$ there exists a state in α that r visits infinitely often along π. An automaton $\mathcal{A}$ accepts $\langle D^*, \tau \rangle$ its run on it is accepting.

We use three-letter acronyms in $\{D, N\} \times \{B, C\} \times \{W, T\}$ to describe types of automata. The first letter describes the transition structure (deterministic or nondeterministic), the second letter describes the acceptance condition (Büchi or co-Büchi), and the third letter designates the objects recognized by the automata (words or trees). Thus, for example, NCW stands for nondeterministic Büchi word automata and NBT stands for nondeterministic Büchi tree automata.

4 Expressiveness

In this section we characterize the expressive power of TRIGGER LOGIC and show that is equivalent to that of DBW.

Proposition 1. *Given a regular formula ψ of the form $r \mapsto \theta(e_1, \ldots, e_k, !e'_1, \ldots, !e'_{k'})$, we can construct an NCW with $|r| + (2^{|e_1|+\cdots+|e_k|}|e'_1| \cdots |e'_{k'}|)$ states that accepts exactly all computations that violate ψ.*

Proof. We start with the special case where $k = k' = 1$ and θ is a disjunction, thus the formula we consider is $\psi = r \mapsto (e \vee !e')$. A path $\pi = \pi_0, \pi_1, \ldots$ violates the formula $r \mapsto (e \vee !e')$ iff there is $i \geq 0$ such that $\pi, 0, i \models L(r)$, $\pi, i \models L(e')$, and $\pi, i \models !L(e)$.

We describe an NCW $\mathcal{U}$ that accepts paths that violate ψ. Let $\mathcal{A}_1, \mathcal{A}_2$, and $\mathcal{A}_3$ be NFWs for r, e, and e', respectively. Let $\mathcal{A}'_2$ be the DCW obtained by determinizing $\mathcal{A}_2$, replacing its accepting states by rejecting sinks, and making all other states accepting. Also, let $\mathcal{A}'_3$ be the NCW obtained by replacing the accepting states of $\mathcal{A}_3$ by accepting sinks. Finally, Let $\mathcal{A}$ be the product of $\mathcal{A}'_2$ and $\mathcal{A}'_3$. The NCW $\mathcal{U}$ starts with $\mathcal{A}_1$. From every accepting state of $\mathcal{A}_1$, it can start executing $\mathcal{A}$. The acceptance

condition requires a run to eventually get stuck in an accepting sink of $\mathcal{A}'_3$ that is not a rejecting sink of $\mathcal{A}'_2$. Formally, for $i \in \{1,2,3\}$, let $\mathcal{A}_i = \langle \Sigma, Q_i, \delta_i, Q_i^0, \alpha_i \rangle$. Then, $\mathcal{A}'_2 = \langle \Sigma, 2^{Q_2}, \delta'_2, \{Q_2^0\}, \alpha'_2 \rangle$, where $\alpha'_2 = \{S : S \cap \alpha_2 = \emptyset\}$, and for all $S \in 2^{Q_2}$ and $\sigma \in \Sigma$, we have

$$\delta'_2(S,\sigma) = \begin{cases} \delta_2(S,\sigma) & \text{if } S \cap \alpha_2 = \emptyset \\ S & \text{otherwise} \end{cases}$$

Note that $\mathcal{A}'_2$ accepts exactly all infinite words none of whose prefixes are accepted by $\mathcal{A}_2$. Also, $\mathcal{A}'_3 = \langle \Sigma, Q_3, \delta'_3, Q_3^0, \alpha_3 \rangle$, where for all $q \in Q_3$ and $\sigma \in \Sigma$, we have

$$\delta'_3(q,\sigma) = \begin{cases} \delta_3(q,\sigma) & \text{if } q \notin \alpha_3 \\ \{q\} & \text{otherwise} \end{cases}$$

Note that $\mathcal{A}'_2$ accepts exactly all infinite words that have a prefix accepted by $\mathcal{A}_3$.

Now, $\mathcal{U} = \langle \Sigma, Q_1 \cup (2^{Q_2} \times Q_3), \delta, Q_1^0, \alpha \rangle$, where for all $q \in Q_1$ and $\sigma \in \Sigma$, we have

$$\delta(q,\sigma) = \begin{cases} \delta_1(q,\sigma) & \text{if } q \notin \alpha_1 \\ \delta_1(q,\sigma) \cup (\{\delta'_2(Q_2^0,\sigma)\} \times \delta'_3(Q_3^0,\sigma)) & \text{otherwise} \end{cases}$$

Also, for all $\langle S, q \rangle \in 2^{Q_2} \times Q_3$ and $\sigma \in \Sigma$, we have

$$\delta(\langle S, q \rangle, \sigma) = \{\delta'_2(S,\sigma)\} \times \delta'_3(q,\sigma).$$

We can partition the state space of $\mathcal{U}$ to three sets:

- $P_1 = Q_1 \cup \{\langle S, q \rangle : S \cap \alpha_2 = \emptyset \text{ and } q \notin \alpha_3\}$,
- $P_2 = \{\langle S, q \rangle : S \cap \alpha_2 = \emptyset \text{ and } q \in \alpha_3\}$, and
- $P_3 = \{\langle S, q \rangle : S \cap \alpha_2 \neq \emptyset \text{ and } q \in Q_3\}$.

The acceptance condition requires an accepting run to eventually leave the automaton $\mathcal{A}_1$ and then, in the product of $\mathcal{A}'_2$ with $\mathcal{A}'_3$, avoid the rejecting sinks of $\mathcal{A}'_2$ and get stuck in the accepting sinks of $\mathcal{A}'_3$. By the definition of δ, each run of $\mathcal{U}$ eventually gets trapped in a set P_i. Hence, the above goal can be achieved by defining the co-Büchi condition $\alpha = P_1 \cup P_3$.

In the general case, where the formula is of the form $r \mapsto \theta(e_1, \ldots, e_k, !e'_1, \ldots, !e'_{k'})$, we define, in a similar way, an NCW $\mathcal{U}$ for paths that violate the formula. As in the special case detailed above, we determinize the NFWs for $e_1, \ldots, e_k$ and make their accepting states rejecting sinks. Let $\mathcal{A}_2^1, \ldots, \mathcal{A}_2^k$ be the automata obtained as described above. Then, we take the NFWs for $e'_1, \ldots, e'_{k'}$ and make their accepting states accepting sinks. Let $\mathcal{A}_3^1, \ldots, \mathcal{A}_3^{k'}$ be the automata obtained as described above. The NCW $\mathcal{U}$ starts with the NFW $\mathcal{A}_1$ for r. From every accepting state of $\mathcal{A}_1$ it can take the transitions from the initial states of the product $\mathcal{A}$ of $\mathcal{A}_2^1, \ldots, \mathcal{A}_2^k, \mathcal{A}_3^1, \ldots, \mathcal{A}_3^{k'}$. In the product $\mathcal{A}$, each state is of the form $\langle S_1, \ldots, S_k, q_1, \ldots, q_{k'} \rangle$ and we partition the states to sets according to the membership of the underlying states in the sinks. Thus, $\mathcal{U}$ is partitioned to $1 + 2^{k+k'}$ sets: one for the states of $\mathcal{A}_1$, and then a set P_v, for each $v \in 2^{k+k'}$. For $v \in 2^{k+k'}$, the set P_v contains exactly all states $\langle S_1, \ldots, S_k, q_1, \ldots, q_{k'} \rangle$ such that for all $1 \leq i \leq k$, we have $S_i \cap \alpha_2^i \neq \emptyset$ iff $v[i] = 0$ and for all $1 \leq j \leq k'$, we have $q_j \in \alpha_3^j$ iff $v[k+j] = 0$.

It is not hard to see that the sets P_v are ordered: $P_v \geq P_{v'}$ (that is, a transition from P_v to $P_{v'}$ is possible) iff for each index $1 \leq i \leq k + k'$, we have $v[i] \geq v'[i]$. It is

left to define the acceptance condition. Recall that θ is a is positive Boolean formula over $e_1, \ldots, e_k, !e'_1, \ldots, !e'_{k'}$. In order to violate a requirement associated with e_i, the projection of a run of $\mathcal{U}$ on the component of ${\mathcal{A}'}_2^i$ has to avoid its rejecting sinks. In order to violate a requirement associated with $!e'_j$, the projection of a run of $\mathcal{U}$ on the component of ${\mathcal{A}'}_3^j$ has to reach an accepting sink. Accordingly, given θ, we say that $v \in 2^{k+k'}$ satisfies θ if assigning **true** to e_i, for $1 \leq i \leq k$, such that $v[i] = 0$ and to $!e'_j$, for $1 \leq j \leq k'$, such that $v[k+j] = 1$, and assigning **false** to all other atoms, satisfies θ. Now, we define the acceptance condition of $\mathcal{U}$ to that a run is accepting if it gets stack in a set P_v for which v satisfies θ. Thus, α is the union of the sets P_v for which v does not satisfies α. As required, $\mathcal{U}$ has $|r| + (2^{|e_1|+\cdots+|e_k|}|e'_1| \cdots |e'_{k'}|)$ states.

Note that we determinize only NFWs associated with regular expressions that are not in the scope of ! . Also, a tighter construction can take the structure of θ into an account and handle conjunctions in θ by nondeterminism rather than by taking the product.

Theorem 1. *A* TRIGGER LOGIC *formula can be translated to equivalent DBW. The blow-up in the translation is doubly exponential.*

Proof. Consider an NCW $\mathcal{A}$ with n states. By applying to $\mathcal{A}$ a construction dual to the break-point construction of [MH84] we get a DCW $\mathcal{A}'$ equivalent to $\mathcal{A}$ with $3^{O(n)}$ states. For completeness, we describe the construction below. Intuitively, $\mathcal{A}'$ follows the standard subset construction applied to $\mathcal{A}$. In order to make sure that every infinite path visits states in α only finitely often, $\mathcal{A}'$ maintains in addition to the set S that follows the subset construction, a subset O of S consisting of states along runs that have not visited α since the last time the O component was empty. The acceptance condition of $\mathcal{A}'$ then requires O to become empty only finitely often. Indeed, this captures the fact that there is a run of $\mathcal{A}$ that eventually prevents the set O from becoming empty, and thus it is a run along which α is visited only finitely often.

Formally, let $\mathcal{A} = \langle \Sigma, Q, q_0, \rho, \alpha \rangle$. Then, $\mathcal{A}' = \langle \Sigma, Q', q'_0, \rho', \alpha' \rangle$, where

- $Q' \subseteq 2^Q \times 2^Q$ is such that $\langle S, O \rangle \in Q'$ if $O \subseteq S \subseteq Q$.
- $q'_0 = \langle \{q_{in}\}, \emptyset \rangle$,
- $\rho' : Q' \times \Sigma \to Q'$ is defined, for all $\langle S, O \rangle \in Q'$ and $\sigma \in \Sigma$, as follows.

$$\rho'(\langle S, O \rangle, \sigma) = \begin{cases} \langle \rho(S, \sigma), \rho(O, \sigma) \setminus \alpha \rangle & \text{if } O \neq \emptyset \\ \langle \rho(S, \sigma), \rho(S, \sigma) \setminus \alpha \rangle & \text{if } O = \emptyset. \end{cases}$$

- $\alpha' = 2^Q \times \{\emptyset\}$.

Given a TRIGGER LOGIC formula Ψ, let $\psi_1, \ldots, \psi_n$ be its underlying regular formulas. Consider a regular formula ψ of the form $r \mapsto \theta(e_1, \ldots, e_i, !e'_k, \ldots, !e'_{k'})$. We saw in Proposition 1 that given ψ, we can construct an NCW with $|r| + (2^{|e_1|+\cdots+|e_k|}|e'_1| \cdots |e'_{k'}|)$ states that accepts exactly all computations that violate ψ. Let $\mathcal{A}_1, \ldots, \mathcal{A}_n$ be the NCWs corresponding to the negations of $\psi_1, \ldots, \psi_n$. For every $1 \leq i \leq n$, we can construct, as described above, a DCW $\mathcal{A}'_i$ equivalent to $\mathcal{A}_i$. By dualizing $\mathcal{A}'_i$, we get a DBW for ψ_i. Now, since DBWs are closed under union and intersection (cf. [Cho74]), we can construct a DBW $\mathcal{A}$ for Ψ. Note that $\mathcal{A}$ is doubly exponential in the size of Ψ.

It remains to show that we can translate from DBW to TRIGGER LOGIC.

Theorem 2. *Given a DBW $\mathcal{A}$, we can construct a* TRIGGER LOGIC *formulas of size exponential in $|\mathcal{A}|$ that is satisfed precisely by the computations that are accepted by $\mathcal{A}$.*

Proof. Let $\mathcal{A} = \langle \Sigma, Q, q_0, \rho, \alpha \rangle$. For $q \in \alpha$, let $\mathcal{A}_q$ be the DFW $\mathcal{A} = \langle \Sigma, Q, q_0, \rho, \{q\} \rangle$, and let $\mathcal{A}^q_q$ be the DFW $\mathcal{A} = \langle \Sigma, Q, q, \rho, \{q\} \rangle$. We do not want $\mathcal{A}_q$ and $\mathcal{A}^q_q$ to accept the empty word ε, so the initial state can be renamed if needed. Let e_q and e^q_q be regular expressions equivalent to $\mathcal{A}_q$ and $\mathcal{A}^q_q$. By [HU79], the lengths of e_q and q^q_q are exponential in $\mathcal{A}$.

A word $w \in \Sigma^\omega$ is accepted by $\mathcal{A}$ iff there is $q \in \alpha$ such that the run of $\mathcal{A}$ on w visits q infinitely often. Thus, the run visits q eventually, and all visits to q are followed by another visits in the (strict) future. We can therefore specifies the set of words that are accepted by $\mathcal{A}$ using the TRIGGER LOGIC formula

$$\bigvee_{q \in \alpha} ((\mathbf{true} \mapsto e_q) \wedge (e_q \mapsto e^q_q)).$$

The class of linear temporal properties that can be expressed by DBW was studied in [KV05b], where it is shown to be precisely the class of linear temporal properties that can be expressed in the alternation-free μ-calculus (AFMC). The translation is with respect to Kripke structures. A given DBW $\mathcal{A}$ can be translated to an AFMC formula φ_A such that for every Kripke structure K we have that $K \models \mathcal{A}$ iff $K \models \varphi_A$, where $K \models \mathcal{A}$ if all computations of K are accepted by $\mathcal{A}$. Generally, the translation to AFMC requires going through DBW, which may involve a doubly exponentially blow-up, as in our translation from TRIGGER LOGIC to DBW. For TRIGGER LOGIC, we can go to AFMC via the NCW constructed Proposition 1, with an exponential, rather than a doubly exponential blow-up.

5 Synthesis

A *transducer* is a labeled finite graph with a designated start node, where the edges are labeled by D ("input alphabet") and the nodes are labeled by Σ ("output alphabet"). A Σ-labeled D-tree is *regular* if it is the unwinding of some transducer. More formally, a transducer is a tuple $\mathcal{T} = \langle D, \Sigma, S, s_{in}, \eta, L \rangle$, where D is a finite set of directions, Σ is a finite alphabet, S is a finite set of states, $s_{in} \in S$ is an initial state, $\eta : S \times D \to S$ is a deterministic transition function, and $L : S \to \Sigma$ is a labeling function. We define $\eta : D^* \to S$ in the standard way: $\eta(\varepsilon) = s_{in}$, and for $x \in D^*$ and $d \in D$, we have $\eta(x \cdot d) = \eta(\eta(x), d)$. Now, a Σ-labeled D-tree $\langle D^*, \tau \rangle$ is regular if there exists a transducer $\mathcal{T} = \langle D, \Sigma, S, s_{in}, \eta, L \rangle$ such that for every $x \in D^*$, we have $\tau(x) = L(\eta(x))$. We then say that the size of the regular tree $\langle D^*, \tau \rangle$, denoted $\|\tau\|$, is $|S|$, the number of states of $\mathcal{T}$.

Given a TRIGGER LOGIC formula ψ over sets I and O of input and output signals (that is, $AP = I \cup O$), the *realizability problem* for ψ is to decide whether there is a *strategy* $f : (2^I)^* \to 2^O$, generated by a transducer[4] such that all the computations of

[4] It is known that if some transducer that generates f exists, then there is also a finite-state transducer [PR89a].

the system generated by f satisfy ψ [PR89a]. Formally, a computation $\rho \in (2^{I \cup O})^\omega$ is generated by f if $\rho = (i_0 \cup o_0), (i_1 \cup o_1), (i_2 \cup o_2), \ldots$ and for all $j \geq 1$, we have $o_j = f(i_0 \cdot i_1 \cdots i_{j-1})$.

5.1 Upper Bound

In this section we show that the translation of TRIGGER LOGIC formulas to automata, described earlier, yields a 2EXPTIME synthesis algorithm for TRIGGER LOGIC.

Theorem 3. *The synthesis problem for* TRIGGER LOGIC *is in 2EXPTIME.*

Proof. Consider a TRIGGER LOGIC formula Ψ over $I \cup O$. By Theorem 1, the formula Ψ can be translated to a DBW $\mathcal{A}$. The size of $\mathcal{A}$ is doubly exponential in the length of Ψ, and its alphabet is $\Sigma = 2^{I \cup O}$. Let $\mathcal{A} = \langle 2^{I \cup O}, Q, q_0, \delta, \alpha \rangle$, and let $\mathcal{A}_t = \langle 2^O, 2^I, Q, q_0, \delta_t, \alpha \rangle$ be the DBT obtained by expanding $\mathcal{A}$ to 2^O-labeled 2^I-trees. Thus, for every $q \in Q$ and and $o \in 2^O$, we have[5]

$$\delta_t(q, o) = \wedge_{i \in 2^I} (i, \delta(q, i \cup o)).$$

We now have that $\mathcal{A}$ is realizable iff $\mathcal{A}_t$ is not empty. Indeed, $\mathcal{A}_t$ accepts exactly all 2^O-labeled 2^I-trees all of whose computations are in $L(\mathcal{A})$. Furthermore, by the nonemptiness-test algorithm of [VW86], the DBT $\mathcal{A}_t$ is not empty iff there is a finite state transducer that realizes $L(\mathcal{A})$.

We discuss the practical advantages of our synthesis algorithm for TRIGGER LOGIC in Section 6.

5.2 Lower Bound

The doubly-exponential lower bound for LTL synthesis is tightly related to the fact a translation of an LTL formula to a deterministic automaton may involve a doubly-exponential blow-up [KV98a]. For TRIGGER LOGIC formulas, such a blow-up seems less likely, as the translation of regular expressions to nondeterministic automata is linear, while the translation of LTL to automata is exponential [VW94]. As we show below, the translation does involve a doubly exponential blow-up, even for formulas of the form $r \mapsto e$, that is, when the underlying regular expressions appear positively. Intuitively, it follows from the need to monitor all the possible runs of an NFW for e on different suffixes (these whose corresponding prefix satisfies r) of the input word.

Theorem 4. *There is a regular expression r and a family of regular expression e_1, $e_2, \ldots$ such that for all $n \geq 1$, the length of e_n is polynomial in n and the smallest DBW for the* TRIGGER LOGIC *formula $r \mapsto e_n$ is doubly-exponential in n.*

Proof. Let $\psi_n = r \mapsto e_n$. We define r and e_n over $\Sigma = \{0, 1, \#, \$\}$ so that the language of $!\psi_n$ constains exactly all words w such that there is a position j with $w[j] = \#$,

[5] Note that the fact $\mathcal{A}$ is deterministic is crucial. A similar construction for a nondeterministic $\mathcal{A}$ results in $\mathcal{A}_t$ whose language may be strictly contained in the required language.

$w[j+1, j+n+1] \in (0|1)^n$, and either there is no position $k > j$ with $w[k] = \$$, or $w[j+1, j+n+1] = w[k+1, k+n+1]$ for the minimal position $k > j$ with $w[k] = \$$.

By [CKS81], the smallest deterministic automaton that recognizes $!\psi_n$ has at least 2^{2^n} states. The proof in [CKS81] considers a language of the finite words. The key idea, however, is valid also for our setting, and implies that the smallest DBW for ψ has at least 2^{2^n} states: whenever the automaton reads \$, it should remember the set of words in $\#; (0|1)^n$ that have appeared since the last \$ (or the beginning of the word, if we are in the first \$).

We define $r = (0|1|\#)[*]; \#$, and e_n is the union of the following REs:

- $\mathbf{true}^i; (\#|\$)$, for $1 \leq i \leq n$: the suffix does not begin with a word in $(0|1)^n$.
- $(\mathbf{true}^i; 0; (!\$)[*]; \$; \mathbf{true}^i; !0)$, for $1 \leq i \leq n$: there is $1 \leq i \leq n$ such that the letter in the i-th position is 0 and is different from the letter in the i-th position after the first \$ in the suffix.
- $(\mathbf{true}^i; 1; (!\$)[*]; \$; \mathbf{true}^i; !1)$, for $1 \leq i \leq n$: there is $1 \leq i \leq n$ such that the letter in the i-th position is 1 and is different from the letter in the i-th position after the first \$ in the suffix.

It is not hard to see that a word w satisfies ψ_n if for every position j, if $w[j] = \#$, then either $w[j+1, j+n+1] \notin (0|1)^n$ or there is $k > j$ such that $w[k] = \$$ and $w[j+1, j+n+1] \neq w[k+1, k+n+1]$, for the minimal $k > j$ with $w[k] = \$$. Thus, as required, a word w satisfies $!\psi_n$ if there is a position j with $w[j] = \#$, $w[j+1, j+n+1] \in (0|1)^n$, and either there is no position $k > j$ with $w[k] = \$$, or $w[j+1, j+n+1] = w[k+1, k+n+1]$ for the minimal position $k > j$ with $w[k] = \$$.

Theorem 4 implies that our algorithm, which involves a translation of TRIGGER LOGIC formulas to DBWs, may indeed have a doubly-exponential time complexity. In Theorem 5 below we show that one cannot do better, as the synthesis problem is 2EXPTIME-hard. Thus, our algorithm is optimal and the synthesis problem for TRIGGER LOGIC is 2EXPTIME-complete.

Theorem 5. *The synthesis problem for* TRIGGER LOGIC *formulas is 2EXPTIME-hard.*

Proof. As in the 2EXPTIME-hardness for CLT* satisfiability [VS85], we do a reduction from the problem whether an alternating exponential-space Turing machine T accepts the empty tape. That is, given such a machine T and a number n in unary, we construct a trigger formula ψ such that T accepts the empty tape using space 2^n iff ψ is realizable. Let $T = \langle \Gamma, Q_u, Q_e, \rightarrow, q_0, q_{acc} \rangle$, where Γ is the tape alphabet, the sets Q_u and Q_e of states are disjoint, and contain the universal and the existential states, respectively, q_0 is the initial state, and q_{acc} is the accepting state. We denote the union $Q_u \cup Q_e$ by Q. Our model of alternation prescribes that the transition relation $\rightarrow \subseteq Q \times \Gamma \times Q \times \Gamma \times \{L, R\}$ has branching degree two, $q_0 \in Q_e$, and the machine T alternates between existential and universal set. When a universal or an existential state of T branches into two states, we distinguish between the left and the right branches. Accordingly, we use $(q, \sigma) \rightarrow \langle (q_l, b_l, \Delta_l), (q_r, b_r, \Delta_r) \rangle$ to indicate that when T is in state $q \in Q_u \cup Q_e$ reading a symbol σ, it branches to the left with (q_l, b_l, Δ_l) and to the

right with (q_r, b_r, Δ_r). (Note that the directions left and right here have nothing to do with the movement direction of the head; these are determined by Δ_l and Δ_r.)

For a configuration c of T, let $succ_l(c)$ and $succ_r(c)$ be the successors of c when applying to it the left and right choices in $\rightarrow$, respectively. Given an input w, a computation tree of T on w is a tree in which each node corresponds to a configuration of T. The root of the tree corresponds to the initial configuration. A node that corresponds to a universal configuration c has two successors, corresponding to $succ_l(c)$ and $succ_r(c)$. A node that corresponds to an existential configuration c has a single successor, corresponding to either $succ_l(c)$ or $succ_r(c)$. The tree is an accepting computation tree if all its branches eventually reach an accepting configuration – one in which the state is q_{acc}. We assume that once a computation reaches an accepting configuration it stays in q_{acc} forever.

We encode a configuration of T by a word $\gamma_1\gamma_2 \ldots (q, \gamma_i) \ldots \gamma_{2^n}$. That is, all the letters in the configuration are in Γ, except for one letter in $Q \times \Gamma$. The meaning of such a configuration is that the j's cell of T, for $1 \leq j \leq 2^n$, is labeled γ_j, the reading head points on cell i, and T is in state q. For example, the initial configuration of T on the empty tape is $@_1, (q_0, \#), \# \cdots \#, @_2$, where $\#$ stands for the empty cell, and $@_1$ and $@_2$ are special tape-end symbols. We can now encode a computation of T by a sequence of configurations.

Let $\Sigma = \Gamma \cup (Q \times \Gamma)$. We can encode the letters in Σ by a set $AP(T) = \{p_1, \ldots, p_m, p'_1, \ldots, p'_m\}$ (with $m = \lceil log|\Sigma| \rceil$) of atomic propositions. The propositions $p'_1, \ldots, p'_m$ are auxiliary; their roles is made clear shortly. We define our formulas over the set $AP = AP(T) \cup \{v_1, \ldots, v_n, v'_1, \ldots, v'_n\} \cup \{real, left_{in}, left_{out}, e\}$ of atomic propositions. The propositions $v_1, \ldots, v_n$ encode the locations of the cells in a configuration. The propositions $v'_1, \ldots, v'_n$ help in increasing the value encoded by $v_1, \ldots, v_n$ properly. The task of the last four atoms is explained shortly.

The set AP of propositions is divided into input and output propositions. The input propositions are $real$ and $left_{in}$. All other propositions are output propositions. With two input propositions, a strategy can be viewed as a 4-ary tree. Recall that the branching degree of T is 2. Why then do we need a 4-ary tree? Intuitively, the strategy should describe a legal and accepting computation tree of T in a "real" 2-ary tree embodied in the strategy tree. This real 2-ary tree is the one in which the input proposition $real$ always holds. Branches in which $real$ is eventually false do not correspond to computations of T and have a different role. Within the real tree, the input proposition $left_{in}$ is used in order to distinguish between the left and right successors of a configurations.

The propositions $v_1, \ldots, v_n$ encode the location of a cell in a configuration of T, with v_1 being the most significant bit. Since T is an exponential-space Turing machine, this location is a number between 0 and $2^n - 1$. To ensure that $v_1, \ldots, v_n$ act as an n-bit counters we need the following formulas:

1. The counter starts at 0.
   - $\mathbf{true} \mapsto \&\&_{i=1}^{n} !v_i$
2. The counter is increased properly. For this we use $v'_1, \ldots, v'_n$ as carry bits.
   - $\mathbf{true}[+] \mapsto v'_n$
   - $(\mathbf{true}[*]; (v_i \&\& v'_i)) \mapsto \mathbf{true}; (!v_i \&\& v'_{i-1})$, for $i = 2, \ldots, n$
   - $(\mathbf{true}[*]; ((v_i \&\& (!v'_i)) || ((!v_i) \&\& v'_i))) \mapsto \mathbf{true}; (v_i \&\& (!v'_{i-1}))$, for $i = 2, \ldots, n$

- $(\mathbf{true}[*]; ((!v_i)\&\&(!v'_i))) \mapsto \mathbf{true}; ((!v_i)\&\&(!v'_{i-1}))$, for $i = 2, \ldots, n$
- $(\mathbf{true}[*]; ((v_1\&\&v'_1)||((!v_1)\&\&(!v'_1)))) \mapsto \mathbf{true}; !v_1$
- $(\mathbf{true}[*]; ((v_1\&\&(!v'_1))||((!v_1)\&\&v'_1))) \mapsto \mathbf{true}; v_1$

Let $\sigma_1 \ldots \sigma_{2^n}, \sigma'_1 \ldots \sigma'_{2^n}$ be two successive configurations of T. For each triple $\langle \sigma_{i-1}, \sigma_i, \sigma_{i+1} \rangle$ with $1 < i < 2^n$, we know for each transition relation of T, what σ'_i should be. Let $next(\langle \sigma_{i-1}, \sigma_i, \sigma_{i+1} \rangle)$ denote our expectation for σ'_i. I.e.[6]

- $next(\langle \gamma_{i-1}, \gamma_i, \gamma_{i+1} \rangle) = \gamma_i$.
- $next(\langle (q, \gamma_{i-1}), \gamma_i, \gamma_{i+1} \rangle) = \begin{cases} \gamma_i & \text{If } (q, \gamma_{i-1}) \to (q', \gamma'_{i-1}, L). \\ (q', \gamma_i) & \text{If } (q, \gamma_{i-1}) \to (q', \gamma'_{i-1}, R). \end{cases}$
- $next(\langle \gamma_{i-1}, (q, \gamma_i), \gamma_{i+1} \rangle) = \gamma'_i$ where $(q, \gamma_i) \to (q', \gamma'_i, \Delta)$.
- $next(\langle \gamma_{i-1}, \gamma_i, (q, \gamma_{i+1}) \rangle) = \begin{cases} \gamma_i & \text{If } (q, \gamma_{i+1}) \to (q', \gamma'_{i+1}, R). \\ (q', \gamma_i) & \text{If } (q, \gamma_{i+1}) \to (q', \gamma'_i, L). \end{cases}$

Since we have two transitions relations, we actually obtain two functions, $next_l$ and $next_r$.

Consistency with $next$ gives us a necessary condition for a path in the computation tree to encode a legal computation. In addition, the computation should start with the initial configuration and reach an accepting state. It is easy to specify the requirements about the initial and accepting configurations. For a letter $\sigma \in \Sigma$, let $\eta(\sigma)$ be the propositional formula over AP in which $p_1, \ldots, p_n$ encode σ. That is, $\eta(\sigma)$ holds in a node iff the truth value of the propositions $p_1 \ldots, p_m$ in that node encodes the symbol σ. Similarly, $\eta'(\sigma)$ is the propositional formula over AP in which $p'_1, \ldots, p'_n$ encode σ. Thus, to say that the first configuration correspond to the empty word we use the following formulas, where $ones$ abbreviates $\wedge_{i=1}^{n} v_i$, and $\#$ denotes the empty symbol:

- $\mathbf{true} \mapsto \eta(@_1); \eta(\langle q_0, \# \rangle$
- $(\mathbf{true}; \mathbf{true}; (!ones)[+]) \mapsto \eta(\#)$
- $((!ones)[+]; ones) \mapsto \mathbf{true}; \eta(@_2)$

We come back to the acceptance condition shortly.

The propositions $p'_1, \ldots, p'_m$ capture the symbol encoded in the previous cell, and special symbols at initial cells. We use the following formula, where $zeros$ abbreviates $\wedge_{i=1}^{n}(!v_i)$.

- $(\mathbf{true}[*]; zero) \mapsto \eta'(@_2)$
- $(\mathbf{true}[*]; ((!ones)\&\&p_j) \mapsto (\mathbf{true}; p'_j)$
- $(\mathbf{true}[*]; ((!ones)\&\&(!p_j)) \mapsto (\mathbf{true}; (!p'_j))$

The output proposition e marks existential configurations. Recall that computations of T start in existential configurations and alternate between universal and existential configurations. The value of e is maintained throughout the configuration. This is expressed using the following formulas:

- $\mathbf{true} \mapsto e$
- $(\mathbf{true}[*]; ((!ones)\&\&e)) \mapsto (\mathbf{true}; e)$
- $(\mathbf{true}[*]; ((!ones)\&\&(!e)) \mapsto (\mathbf{true}; (!e))$
- $(\mathbf{true}[*]; (ones\&\&e))) \mapsto (\mathbf{true}; (!e)$
- $(\mathbf{true}[*]; (ones\&\&(!e)) \mapsto (\mathbf{true}; e)$

[6] Special handling of end cases is needed, when the head of T read the left or right end markers. For simplicity, we ignore this technicality here.

The output proposition $left_{out}$ marks configurations that are left successors. The value of $left_{out}$ is determined according to the value of $left_{in}$ at the end of the previous configuration, and is maintained throughout the configuration, where it is used in order to decide whether the configuration should be consistent with $next_l$ or with $next_r$. The following formulas ensure that the value is indeed maintained and that universal configurations are followed by both left and right configurations. On the other hand, for the successors of existential configurations, the strategy has no restrictions on the value of $left_{out}$, and can choose the same value for the two successors.

- $(\mathbf{true}[*]; ((!ones)\&\&left_{out})) \mapsto (\mathbf{true}; left_{out})$
- $(\mathbf{true}[*]; ((!ones)\&\&(!left_{out})) \mapsto (\mathbf{true}; (!left_{out}))$
- $(\mathbf{true}[*]; (ones\&\&(!e)\&\&(!left_{in}))) \mapsto (\mathbf{true}; left_{out})$
- $(\mathbf{true}[*]; (ones\&\&(!e)\&\&left_{in})) \mapsto (\mathbf{true}; (!left_{out}))$

The difficult part in the reduction is in guaranteeing that the sequence of configurations is indeed consistent with $next_l$ and $next_r$. To enforce this, we have to relate σ_{i-1}, σ_i, and σ_{i+1} with σ'_i for each i in every two successive configurations $\sigma_1 \ldots \sigma_{2^n}, \sigma'_1 \ldots \sigma'_{2^n}$. One natural way to do so is by a conjunction of formulas like "whenever we meet a cell at location $i-1$ and the labeling of the next three cells forms the triple $\langle \sigma_{i-1}, \sigma_i, \sigma_{i+1} \rangle$, then the next time we meet a cell at location i, this cell is labeled $next(\langle \sigma_{i-1}, \sigma_i, \sigma_{i+1} \rangle)$". The problem is that, as i can take a value from 0 to $2^n - 1$, there are exponentially many such conjuncts. This is where the non-real part of the tree is helpful [VS85].

Recall that the input proposition $real$ is used to labeled the "real" part of the strategy tree – the one that corresponds to the computation tree of T. Once we branch according to $!real$, we move to the auxiliary part of the tree. Consider now an arbitrary trace, either it is a real trace, on which $real$ is always true, or it reaches the auxiliary part of the tree, where $real$ is false. We refer to the latter trace as an *auxiliary trace*. The point at which $real$ is true for the last time is the *end* of this auxiliary trace.

Consider a point x on an auxiliary trace that is followed by the end point y. There are the following possibilities:

1. $ones$ holds less than or more than once between x and y, which means that x and y do not belong to successive configurations.
2. $ones$ holds once between x and y, which means that they belong to successive configurations, but the assignment to $p_1, \ldots, p_n$ at x and y disagree, which means that they are not corresponding cells.
3. $ones$ holds once between x and y, which means that they belong to success configurations, and the assignments to $p_1, \ldots, p_n$ at x and y agree, which means that they are corresponding cells.

Accordingly, in order to ensure correct succession of configurations of T, we use the formula

- $real[+] \mapsto \psi$,

where ψ is a union of the following regular expressions:

- $(\mathbf{true}[+]; \bigvee_{\gamma \in \Gamma} \eta(\langle s_a, \gamma \rangle)$: the trace reaches an accepting configuration;
- $(!ones\&\&real)[+]; ones$: the points x and y belong to same configuration;
- $real[*]; ones\&\ℜ real[+]; ones\&\&real$: the points belong to non-successive configurations;

- $v_i \&\& real; real[+]; (!v_i)\&\& real; !real$, for $i = 1, \ldots, n$: the points do not agree on the value of the i-th bit in the encoding of their address and therefore they have different cell locations;
- $\eta'(\sigma_1)\&\&\eta(\sigma_2)\&\& real; \eta(\sigma_3)\&\& real; real[+]; left_{out}\&\& next_l(\sigma_1, \sigma_2, \sigma_3)\&\& real; !real$, for $\sigma_1, \sigma_2, \sigma_3 \in \Sigma$: the points x and y are in the same cell of a configuration and its left successor, and $next_l$ is respected. Note that the propositions p'_i are used in order to refer to the cell before x.
- $\eta'(\sigma_1)\&\&\eta(\sigma_2)\&\& real; \eta(\sigma_3)\&\& real; real[+]; (!left_{out})\&\& next_r(\sigma_1, \sigma_2, \sigma_3)\&\& real; !real$, for $\sigma_1, \sigma_2, \sigma_3 \in \Sigma$: the points x and y are in the same cell of a configuration and its right successor, and $next_l$ is respected.

Note that the TRIGGER LOGIC formula constructed in the reduction is a conjunction of formulas of the form $r \mapsto e$. Thus, the problem is 2EXPTIME-hard already for this fragment of TRIGGER LOGIC.

6 Practice Issues

In Section 5.1, we proved that the synthesis problem for TRIGGER LOGIC can be solved in doubly-exponential time. This bound is no better on its face than the doubly-exponential time upper bound proved in [PR89a, KV05c] for LTL synthesis. A closer examination reveals, however, that the algorithms in [PR89a, KV05c] have time complexity of the form 4^{4^n}, while the algorithm described here has time complexity of the form 4^{2^n}. This, however, is not what we view as the main advantage of this algorithm. Rather, its main advantage is that it is significantly simpler. Unlike the algorithm in [PR89a], we need not apply Safra's determinization construction nor solving complex games. Unlike [KV05b], we need not use progress measures. Our algorithm is based solely on using the subset construction and solving the emptiness problem for Büchi tree automata.

In this section we show that our algorithm for TRIGGER LOGIC has additional appealing properties in practice.

A symbolic implementation. Theorem 3 reduces the TRIGGER LOGIC synthesis problem to the nonemptiness problem of a DBT obtained by dualizing a DCW that is the result of applying the break-point construction of [MH84] to the NCW that corresponds to the negation of the TRIGGER LOGIC formula. In [MS08a, MS08b], the authors described a symbolic implementation of the break-point construction for word automata. For tree automata, the symbolic algorithm for the nonemptiness construction is not more difficult, as both word emptiness and tree emptiness for Büchi automata are based on nested-fixpoint algorithms [EL86, VW86], using a quadratic number of symbolic operations.

In more details, the state space of the DBT consists of sets of states, it can be encoded by Boolean variables, and the DBT's transitions can be encoded by relations on these variables and a primed version of them. The fixpoint solution for the nonemptiness problem of DBT (c.f., [VW86]) then yields a symbolic solution to the synthesis problem. Moreover, the BDDs that are generated by the symbolic decision procedure can be used to generate a symbolic witness strategy. The Boolean nature of BDDs then

makes it very easy to go from this BDD to a sequential circuit for the strategy. It is known that a BDD can be viewed as an expression (in DAG form) that uses the "if then else" as a single ternary operator. Thus, a BDD can be viewed as a circuit built from if-then-else gates. More advantages of the symbolic approach are described in [HRS05]. As mentioned above, [HRS05] also suggests a symbolic solution for the LTL synthesis problem. However, the need to circumvent Safra's determinization causes the algorithm in [HRS05] to be complete only for a subset of LTL. Likewise, the need to implement the progress ranks of [KV05a] using a binary encoding challenges BDD-based implementations [TV07]. Our approach here circumvents both Safra's determinization and ranks, facilitating a symbolic implementation.

Incremental synthesis. A serious drawback of current synthesis algorithms is that they assume a comprehensive set of temporal assertions as a starting point. In practice, however, specifications are evolving: temporal assertions are added, deleted, or modified during the design process. Here, we describe how our synthesis algorithm can support *incremental* synthesis, where the temporal assertions are given one by one. We show how working with DBWs enables us, when we check the realizability of $\psi \&\& \psi'$, to use much of the work done in checking the realizability of ψ and ψ' in isolation.

Essentially, we show that when we construct and check the emptiness of the DBT to which realizability of $\psi \&\& \psi'$ is reduced, we can use much of the work done in the process of checking the emptiness of the two (much smaller) DBTs to which realizability of ψ and ψ' is reduced (in isolation). Let $\mathcal{A}$ and $\mathcal{A}'$ be the DBTs to which realizability of ψ and ψ' is reduced, respectively. Recall that $\mathcal{A}$ and $\mathcal{A}'$ are obtained from NCWs with state spaces Q and Q'. A non-incremental approach generates the DBT that corresponds to $\psi \&\& \psi'$. By Theorem 3, this results in a DBT $\mathcal{U}$ with state space $3^{Q \cup Q'}$. On the other hand, the state spaces of $\mathcal{A}$ and $\mathcal{A}'$ are much smaller, and are 3^Q and $3^{Q'}$, respectively.

Let us examine the structure of the state space of $\mathcal{U}$ more carefully. Each of its states can be viewed as a pair $\langle S \cup S', O \cup O' \rangle$, for $O \subseteq S \subseteq Q$ and $O' \subseteq S' \subseteq Q'$. The state corresponds to the states $\langle S, O \rangle$ of $\mathcal{A}$ and $\langle S', O' \rangle$ of $\mathcal{A}'$. Clearly, if one of these states is empty (that is, if the automaton accept no tree starting from these states), then so is $\langle S \cup S', O \cup O' \rangle$. Thus, an incremental algorithm can start by marking all such states as empty and continue the emptiness check only with the (hopefully much smaller) state space.

(Note that this idea does not apply to disjunctions. Suppose that neither ψ nor ψ' is realizable, and we want to check if $\psi || \psi'$ is realizable. It is not clear how to leverage realizability checking of ψ and ψ', when we check realizability of $\psi || \psi'$.)

Adding assumptions. The method described above cannot be applied for formulas of the form $\psi' \rightarrow \psi$, with ψ' and ψ formulas in TRIGGER LOGIC. Note that since TRIGGER LOGIC is not closed under negation, the specification $\psi' \rightarrow \psi$ is not a TRIGGER LOGIC formula. Still, such an implication arises naturally when we want to synthesize ψ with respect to environments satisfying ψ'. To handle such specifications, we apply the automata-theoretic constructions of Section 5.1 to both ψ' and ψ obtaining DBT $\mathcal{A}_t^{\psi'}$ and $\mathcal{A}_t^{\psi}$, with acceptance conditions α' and α. We now take the product of $\mathcal{A}_t^{\psi'}$ and

$\mathcal{A}_t^{\psi}$, and use as acceptance condition the Streett pair $\langle \alpha', \alpha \rangle$. A symbolic algorithm for Streett tree automata is described in [KV98b]. For Street(1) condition, that is, a single pair Streett condition, the algorithm requires a cubic number of symbolic operations.

References

[AFF+02] Armoni, R., Fix, L., Flaisher, A., Gerth, R., Ginsburg, B., Kanza, T., Landver, A., Mador-Haim, S., Singerman, E., Tiemeyer, A., Vardi, M.Y., Zbar, Y.: The ForSpec temporal logic: A new temporal property-specification logic. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 196–211. Springer, Heidelberg (2002)

[ALW89] Abadi, M., Lamport, L., Wolper, P.: Realizable and unrealizable concurrent program specifications. In: Ronchi Della Rocca, S., Ausiello, G., Dezani-Ciancaglini, M. (eds.) ICALP 1989. LNCS, vol. 372, pp. 1–17. Springer, Heidelberg (1989)

[AMPS98] Asarin, E., Maler, O., Pnueli, A., Sifakis, J.: Controller synthesis for timed automata. In: IFAC Symposium on System Structure and Control, pp. 469–474. Elsevier, Amsterdam (1998)

[AT04] Alur, R., La Torre, S.: Deterministic generators and games for ltl fragments. ACM Transactions on Computational Logic 5(1), 1–25 (2004)

[ATW05] Althoff, C.S., Thomas, W., Wallmeier, N.: Observations on determinization of Büchi automata. In: Farré, J., Litovsky, I., Schmitz, S. (eds.) CIAA 2005. LNCS, vol. 3845, pp. 262–272. Springer, Heidelberg (2006)

[BBE+01] Beer, I., Ben-David, S., Eisner, C., Fisman, D., Gringauze, A., Rodeh, Y.: The temporal logic Sugar. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 363–367. Springer, Heidelberg (2001)

[BDF+04] Basin, D.A., Deville, Y., Flener, P., Hamfelt, A., Nilsson, J.F.: Synthesis of programs in computational logic. In: Bruynooghe, M., Lau, K.-K. (eds.) Program Development in Computational Logic. LNCS, vol. 3049, pp. 30–65. Springer, Heidelberg (2004)

[BFG+05] Bustan, D., Flaisher, A., Grumberg, O., Kupferman, O., Vardi, M.Y.: Regular vacuity. In: Borrione, D., Paul, W. (eds.) CHARME 2005. LNCS, vol. 3725, pp. 191–206. Springer, Heidelberg (2005)

[BGJ+07] Bloem, R., Galler, S., Jobstmann, B., Piterman, N., Pnueli, A., Weiglhofer, M.: Automatic hardware synthesis from specifications: a case study. In: Proc. Conference on Design, Automation and Test in Europe, pp. 1188–1193. ACM, New York (2007)

[BL69] Büchi, J.R., Landweber, L.H.: Solving sequential conditions by finite-state strategies. Trans. AMS 138, 295–311 (1969)

[CGP99] Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)

[Cho74] Choueka, Y.: Theories of automata on ω-tapes: A simplified approach. Journal of Computer and Systems Science 8, 117–141 (1974)

[Chu63] Church, A.: Logic, arithmetics, and automata. In: Proc. Int. Congress of Mathematicians, 1962, pp. 23–35. Institut Mittag-Leffler (1963)

[CKS81] Chandra, A.K., Kozen, D.C., Stockmeyer, L.J.: Alternation. Journal of the Association for Computing Machinery 28(1), 114–133 (1981)

[Dil89] Dill, D.L.: Trace theory for automatic hierarchical verification of speed independent circuits. MIT Press, Cambridge (1989)

[DR09] Doyen, L., Raskin, J.-F.: Antichains for the automata-based approach to model-checking. Logical Methods in Computer Science 5(1) (2009)

[EC82] Emerson, E.A., Clarke, E.M.: Using branching time logic to synthesize synchronization skeletons. Science of Computer Programming 2, 241–266 (1982)

[EF06] Eisner, C., Fisman, D.: A Practical Introduction to PSL. Springer, Heidelberg (2006)

[EKM98] Elgaard, J., Klarlund, N., Möller, A.: Mona 1.x: new techniques for WS1S and WS2S. In: Vardi, M.Y. (ed.) CAV 1998. LNCS, vol. 1427, pp. 516–520. Springer, Heidelberg (1998)

[EL86] Emerson, E.A., Lei, C.-L.: Efficient model checking in fragments of the propositional μ-calculus. In: Proc. 1st IEEE Symp. on Logic in Computer Science, pp. 267–278 (1986)

[HRS05] Harding, A., Ryan, M., Schobbens, P.: A new algorithm for strategy synthesis in LTL games. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 477–492. Springer, Heidelberg (2005)

[HU79] Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, Reading (1979)

[Jur00] Jurdzinski, M.: Small progress measures for solving parity games. In: Reichel, H., Tison, S. (eds.) STACS 2000. LNCS, vol. 1770, pp. 290–301. Springer, Heidelberg (2000)

[Kla98] Klarlund, N.: Mona & Fido: The logic-automaton connection in practice. In: Nielsen, M. (ed.) CSL 1997. LNCS, vol. 1414. Springer, Heidelberg (1998)

[KPV06] Kupferman, O., Piterman, N., Vardi, M.Y.: Safraless compositional synthesis. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 31–44. Springer, Heidelberg (2006)

[KV98a] Kupferman, O., Vardi, M.Y.: Freedom, weakness, and determinism: from linear-time to branching-time. In: Proc. 13th IEEE Symp. on Logic in Computer Science, pp. 81–92 (1998)

[KV98b] Kupferman, O., Vardi, M.Y.: Weak alternating automata and tree automata emptiness. In: Proc. 30th ACM Symp. on Theory of Computing, pp. 224–233 (1998)

[KV00] Kupferman, O., Vardi, M.Y.: Synthesis with incomplete information. In: Advances in Temporal Logic, pp. 109–127. Kluwer Academic Publishers, Dordrecht (2000)

[KV05a] Kupferman, O., Vardi, M.Y.: Complementation constructions for nondeterministic automata on infinite words. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 206–221. Springer, Heidelberg (2005)

[KV05b] Kupferman, O., Vardi, M.Y.: From linear time to branching time. ACM Transactions on Computational Logic 6(2), 273–294 (2005)

[KV05c] Kupferman, O., Vardi, M.Y.: Safraless decision procedures. In: Proc. 46th IEEE Symp. on Foundations of Computer Science, pp. 531–540 (2005)

[LP85] Lichtenstein, O., Pnueli, A.: Checking that finite state concurrent programs satisfy their linear specification. In: Proc. 12th ACM Symp. on Principles of Programming Languages, pp. 97–107 (1985)

[MH84] Miyano, S., Hayashi, T.: Alternating finite automata on ω-words. Theoretical Computer Science 32, 321–330 (1984)

[MS08a] Morgenstern, A., Schneider, K.: From ltl to symbolically represented deterministic automata. In: Logozzo, F., Peled, D.A., Zuck, L.D. (eds.) VMCAI 2008. LNCS, vol. 4905, pp. 279–293. Springer, Heidelberg (2008)

[MS08b] Morgenstern, A., Schneider, K.: Generating deterministic ω-automata for most LTL formulas by the breakpoint construction. In: Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen, Freiburg, Germany (2008)

[MW80] Manna, Z., Waldinger, R.: A deductive approach to program synthesis. ACM Transactions on Programming Languagues and Systems 2(1), 90–121 (1980)

[MW84] Manna, Z., Wolper, P.: Synthesis of communicating processes from temporal logic specifications. ACM Transactions on Programming Languagues and Systems 6(1), 68–93 (1984)

[Pit07] Piterman, N.: From nondeterministic Büchi and Streett automata to deterministic parity automata. Logical Methods in Computer Science 3(3), 5 (2007)

[Pnu77] Pnueli, A.: The temporal logic of programs. In: Proc. 18th IEEE Symp. on Foundations of Computer Science, pp. 46–57 (1977)

[PPS06] Piterman, N., Pnueli, A., Saar, Y.: Synthesis of reactive(1) designs. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 364–380. Springer, Heidelberg (2006)

[PR89a] Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Proc. 16th ACM Symp. on Principles of Programming Languages, pp. 179–190 (1989)

[PR89b] Pnueli, A., Rosner, R.: On the synthesis of an asynchronous reactive module. In: Ronchi Della Rocca, S., Ausiello, G., Dezani-Ciancaglini, M. (eds.) ICALP 1989. LNCS, vol. 372, pp. 652–671. Springer, Heidelberg (1989)

[Rab72] Rabin, M.O.: Automata on infinite objects and Church's problem. Amer. Mathematical Society, Providence (1972)

[Ros92] Rosner, R.: Modular Synthesis of Reactive Systems. PhD thesis, Weizmann Institute of Science (1992)

[RS59] Rabin, M.O., Scott, D.: Finite automata and their decision problems. IBM Journal of Research and Development 3, 115–125 (1959)

[SDC01] Shimizu, K., Dill, D.L., Chou, C.-T.: A specification methodology by a collection of compact properties as applied to the intel itanium processor bus protocol. In: Margaria, T., Melham, T.F. (eds.) CHARME 2001. LNCS, vol. 2144, pp. 340–354. Springer, Heidelberg (2001)

[THB95] Tasiran, S., Hojati, R., Brayton, R.K.: Language containment using non-deterministic omega-automata. In: Camurati, P.E., Eveking, H. (eds.) CHARME 1995. LNCS, vol. 987, pp. 261–277. Springer, Heidelberg (1995)

[TV07] Tabakov, D., Vardi, M.Y.: Model checking Büechi specifications. In: 1st International Conference on Language and Automata Theory and Application Principles of Programming Languages (2007)

[Var95] Vardi, M.Y.: An automata-theoretic approach to fair realizability and synthesis. In: Wolper, P. (ed.) CAV 1995. LNCS, vol. 939, pp. 267–292. Springer, Heidelberg (1995)

[VR05] Vijayaraghavan, S., Ramanathan, M.: A Practical Guide for SystemVerilog Assertions. Springer, Heidelberg (2005)

[VS85] Vardi, M.Y., Stockmeyer, L.: Improved upper and lower bounds for modal logics of programs. In: Proc. 17th ACM Symp. on Theory of Computing, pp. 240–251 (1985)

[VW86] Vardi, M.Y., Wolper, P.: Automata-theoretic techniques for modal logics of programs. Journal of Computer and Systems Science 32(2), 182–221 (1986)

[VW94] Vardi, M.Y., Wolper, P.: Reasoning about infinite computations. Information and Computation 115(1), 1–37 (1994)

[WD91] Wong-Toi, H., Dill, D.L.: Synthesizing processes and schedulers from temporal specifications. In: Clarke, E.M., Kurshan, R.P. (eds.) Proc. 2nd Int. Conf. on Computer Aided Verification. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 3, pp. 177–186. AMS, Providence (1991)

Semiring-Induced Propositional Logic: Definition and Basic Algorithms *

Javier Larrosa, Albert Oliveras, and Enric Rodríguez-Carbonell**

Technical University of Catalonia, Barcelona

Abstract. In this paper we introduce an extension of propositional logic that allows clauses to be weighted with values from a generic semiring. The main interest of this extension is that different instantiations of the semiring model different interesting computational problems such as *finding a model*, *counting the number of models*, *finding the best model* with respect to an objective function, *finding the best model* with respect to several independent objective functions, or *finding the set of pareto-optimal models* with respect to several objective functions.

Then we show that this framework unifies several solving techniques and, even more importantly, rephrases them from an algorithmic language to a logical language. As a result, several solving techniques can be trivially and elegantly transferred from one computational problem to another. As an example, we extend the basic DPLL algorithm to our framework producing an algorithm that we call SDPLL. Then we enhance the basic SDPLL in order to incorporate the three features that are common in all modern SAT solvers: *backjumping*, *learning* and *restarts*.

As a result, we obtain an extremely simple algorithm that captures, unifies and extends in a well-defined logical language several techniques that are valid for arbitrary semirings.

Keywords: semiring, marginalization problem, DPLL.

1 Introduction

The importance of semirings to unify apparently unrelated combinatorial computational problems has been known and studied for a long time [28,27,5,19,1]. Some well-known unifiable problems occur in (soft) *constraint networks*, *probabilistic networks* or *relational databases*, each of them having many real-life domains of application.

There are many advantages for semiring unification. On the one hand, it provides a very general formalism for algorithmic development: instead of re-discovering the same technique for each particular type of problem, it can be formulated in an abstract form and immediately applied to any problem that fits into the framework (e.g. Adaptive Consistency [12], Directional Resolution [11], Nonserial Dynamic Programming [3], the basic pseudo-boolean method [9], and many others [1] are essentially independent

* A version of this paper extended with proofs is available at `http://www.lsi.upc.edu/~erodri/lpar16ex.pdf`

** Authors partially supported by Spanish Min. of Science and Innovation through the projects TIN2006-15387-C03-02 and TIN2007-68093-C02-01 (LogicTools-2).

E.M. Clarke and A. Voronkov (Eds.): LPAR-16, LNAI 6355, pp. 332–347, 2010.

developments of the same algorithm). On the other hand, the unification provides a convenient formalism for algorithmic generalization (see e.g. [8] and [29]).

In this paper we study this idea in the context of propositional logic. First of all, we extend propositional logic by incorporating a generic semiring and allowing boolean formulas to be weighted with semiring values. We define its semantics and extend classical notions such as logical implication ($\models$) or equivalence ($\equiv$).

Then we show that semiring-induced propositional logic can model in a natural way very important combinatorial problems over boolean variables such as *finding a model* (SAT), *counting the number of models* (#SAT), *finding the best model with respect to an objective function*, *finding the best model with respect to several independent objective functions* or *finding the set of pareto-optimal models with respect to several objective functions*.

The principal advantage of the extension is conceptual because techniques for these problems are mostly defined in a procedural way and it is difficult to see the logic that is behind the execution of the procedure (see e.g. [7,13,30]). With our approach, solving techniques can be explained in logical terms. As an example, we extend the basic DPLL algorithm [10] to semiring-induced logic producing an algorithm that we call SDPLL. When SDPLL is instantiated with different semirings to model, for example, SAT, #SAT or Max-SAT, it is faithful to the simplest algorithms for each problem [10,4,7]. Therefore, we show that these algorithms were in fact the same algorithm *modulo the corresponding semiring*. Additionally, it immediately provides basic enumeration algorithms to not-so-studied problems such as multi-objective model optimization. Then, we enhance the basic SDPLL with three features that are present in all modern SAT solvers [22]: backjumping, learning and restarts. Thus, we show that they are also valid in our much more general framework.

2 Semirings

In this paper, a semiring $\mathcal{A} = (A, \oplus, \otimes)$ consists of a non-empty set A together with two binary operations $\oplus$ and $\otimes$ such that both operations are *commutative* and *associative*, and $\otimes$ *distributes over* $\oplus$[1].

If there is an element $\mathbf{0} \in A$ such that $\mathbf{0} \oplus a = a$ and $\mathbf{0} \otimes a = \mathbf{0}$ for all $a \in A$ then A is a semiring with *zero* element. Similarly, if there is an element $\mathbf{1} \in A$ such that $\mathbf{1} \otimes a = a$ for all $a \in A$ then A is a semiring with *unit* element. It can be assumed without loss of generality that a semiring has a zero element, as noted in [19]. Semirings admit at most one zero and one unit element.

Given a semiring $\mathcal{A}$, a binary relation $\leq_{\mathcal{A}}$ can be defined as follows: for any $a, b \in A$, $a \leq_{\mathcal{A}} b$ holds if there exists $c \in A$ such that $a \oplus c = b$. This relation can be shown to be a pre-order [19]; i.e., $i)$ for all $a \in A$, $a \leq_{\mathcal{A}} a$ (reflexivity), and $ii)$ if $a \leq_{\mathcal{A}} b$ and $b \leq_{\mathcal{A}} c$ then $a \leq_{\mathcal{A}} c$ (transitivity). In this paper we will restrict ourselves to semirings with zero and unit elements, noted $\mathcal{A} = (A, \oplus, \otimes, \mathbf{0}, \mathbf{1})$, whose pre-order is a partial order (i.e., it holds that $a \leq_{\mathcal{A}} b$ and $b \leq_{\mathcal{A}} a$ implies $a = b$).

The semiring order also has the properties that $iii)$ $a \leq_{\mathcal{A}} b$ and $a' \leq_{\mathcal{A}} b'$ imply $a \oplus a' \leq_{\mathcal{A}} b \oplus b'$ and $a \otimes a' \leq_{\mathcal{A}} b \otimes b'$; and $iv)$ for all $a \in A$, $\mathbf{0} \leq_{\mathcal{A}} a$. As a

[1] This definition corresponds to what is called a *commutative semiring* elsewhere [15].

semiring	A	$\oplus$	$\otimes$	$\mathbf{0}$	$\mathbf{1}$	applic.
$\mathcal{A}_{\texttt{bool}}$	$\{0,1\}$	$\vee$	$\wedge$	0	1	SAT
$\mathcal{A}_{\texttt{count}}$	$\mathbb{R}^+$	$+$	$\times$	0	1	#SAT
$\mathcal{A}_{\texttt{max}\times}$	$\mathbb{R}^+$	max	$\times$	0	1	Max-SAT
$\mathcal{A}_{\texttt{min}+}$	$\mathbb{R}^+ \cup \{\infty\}$	min	$+$	∞	0	Max-SAT
$\mathcal{A}^n$	A^n	$\oplus^n$	$\otimes^n$	$(\mathbf{0},\ldots,\mathbf{0})$	$(\mathbf{1},\ldots,\mathbf{1})$	
$\mathcal{A}^f$	A^f	$\oplus^f$	$\otimes^f$	$\{\mathbf{0}\}$	$\{\mathbf{1}\}$	

Fig. 1. The first four rows show four different semirings with immediate application. The last two rows show two different semiring constructors (multidimensional and frontier extensions) which are relevant to model multi-criteria problems.

consequence, $\oplus$ increases monotonically (i.e., $a \leq_{\mathcal{A}} a \oplus b$); and when applied to values smaller than or equal to $\mathbf{1}$, then $\otimes$ decreases monotonically (i.e., if $a, b \leq_{\mathcal{A}} \mathbf{1}$ then $a \otimes b \leq_{\mathcal{A}} a$). As usual, $a \neq b$ and $a \leq_{\mathcal{A}} b$ will be noted as $a <_{\mathcal{A}} b$.

The first four rows of Figure 1 summarize well-known semirings that will be used to highlight the expressivity of semiring-induced logic. The first column indicates the semiring name, columns $2-6$ show their components and column 7 indicates their paradigmatic application (to be seen in Section 4). In all of them the induced order $\leq_{\mathcal{A}}$ is the usual (total) order, except for $\mathcal{A}_{\texttt{min}+}$ where it is reversed (e.g. $5 \leq_{\mathcal{A}_{\texttt{min}+}} 2$).

Sometimes it is useful to derive a new semiring from an already existing one. In the following we consider two useful extensions (they are summarized in the last two rows of Figure 1). The *multidimensional extension* generates a new semiring whose values are vectors of semiring values.

Definition 1. *Let $\mathcal{A} = (A, \oplus, \otimes, \mathbf{0}, \mathbf{1})$ be a semiring. Its* multidimensional extension *[19] is $\mathcal{A}^n = (\, A^n, \oplus^n, \otimes^n, \mathbf{0}^n, \mathbf{1}^n)$ where*

- $A^n = A \times \ldots \times A$
- $(a_1,\ldots,a_n) \oplus^n (b_1,\ldots,b_n) = (a_1 \oplus b_1, \ldots, a_n \oplus b_n)$
- $(a_1,\ldots,a_n) \otimes^n (b_1,\ldots,b_n) = (a_1 \otimes b_1, \ldots, a_n \otimes b_n)$
- $\mathbf{0}^n = (\mathbf{0},\ldots,\mathbf{0})$
- $\mathbf{1}^n = (\mathbf{1},\ldots,\mathbf{1})$

In this case, $(a_1,\ldots,a_n) \leq_{\mathcal{A}^n} (b_1,\ldots,b_n)$ if and only if $\forall_{1\leq j\leq n}, a_j \leq_{\mathcal{A}} b_j$. Observe that if $\leq_{\mathcal{A}}$ is a total order then $\leq_{\mathcal{A}^n}$ is the usual product order in a product of posets. If $a \leq_{\mathcal{A}^n} b$ one says that b *dominates* a.

Given a semiring, the *frontier extension* [6] generates a new semiring whose values are sets of non-dominated values from the original semiring.

Definition 2. *Let $\mathcal{A} = (A, \oplus, \otimes, \mathbf{0}, \mathbf{1})$ be a semiring. The set of non-dominated elements of $S \subseteq A$ is defined as*

$$||S|| = \{v \in S \mid \forall_{w \in S}\ v \not< w\}\ .$$

The frontier extension *of $\mathcal{A}$ is $\mathcal{A}^f = (A^f, \oplus^f, \otimes^f, \mathbf{0}^f, \mathbf{1}^f\ )$, where*

- $A^f = \{||S|| \mid S \subseteq A\}$
- $S \oplus^f R = ||S \cup R||$

- $S \otimes^f R = ||\{a \otimes b \mid a \in S, b \in R\}||$
- $\mathbf{0}^f = \{\mathbf{0}\}$
- $\mathbf{1}^f = \{\mathbf{1}\}$

In this case $S \leq_{\mathcal{A}^f} R$ holds if and only if for all $a \in S$ there is $b \in R$ such that $a \leq_{\mathcal{A}} b$. This is the so-called *frontier order* widely used in multi-objective optimization.

Example 1. Consider semiring $\mathcal{A}_{\texttt{bool}} = (\{0,1\}, \vee, \wedge, 0, 1)$. Its bidimensional extension $\mathcal{A}^2_{\texttt{bool}}$ is the set of two-dimensional bit vectors such as $a = (0,1)$ and $b = (1,1)$. Note that $a \leq_{\mathcal{A}^2_{\texttt{bool}}} b$ as $0 \leq_{\mathcal{A}_{\texttt{bool}}} 1$ and $1 \leq_{\mathcal{A}_{\texttt{bool}}} 1$, $a \vee^2 b = (0 \vee 1, 1 \vee 1) = (1,1)$ and $a \wedge^2 b = (0 \wedge 1, 1 \wedge 1) = (0,1)$. The frontier extension of $\mathcal{A}^2_{\texttt{bool}}$ is $(\mathcal{A}^2_{\texttt{bool}})^f$. Its values are sets of non-dominated two-dimensional bit vectors such as $a = \{(0,1),(1,0)\}$ or $b = \{(1,1)\}$. The set $\{(0,1),(1,1)\}$ does not belong to the semiring as $(1,1)$ dominates $(0,1)$. Note that $a \leq_{(\mathcal{A}^2_{\texttt{bool}})^f} b$ as every element of a is dominated by an element of b. Furthermore, $a \vee^{2f} b = ||\{(0,1),(1,0),(1,1)\}|| = \{(1,1)\}$ and $a \wedge^{2f} b = \{(0,1),(1,0)\}$.

3 Semiring-Induced Propositional Logic

3.1 Syntax

Let P be a finite set of propositional symbols that will remain fixed throughout the paper. If $p \in P$, then p and $\neg p$ are *literals*. The *negation* of a literal l, written $\neg l$, denotes $\neg p$ if l is p, and p if l is $\neg p$. A *clause* C is a (possibly empty) finite disjunction of literals. A *unit clause* consists of a single literal. The *empty clause* is noted $\Box$.

A (partial truth) *assignment* M is a set of literals such that if l is in M, then $\neg l$ is not. A literal l is *true* in M if $l \in M$, is *false* in M if $\neg l \in M$, and is *undefined* in M otherwise. The assignment M is *total* if every symbol of P is defined in M. The set of total assignments is noted $\mathcal{M}$.

An assignment M satisfies a clause C if at least one of its literals is true in M. It falsifies C if all the literals of C are false in M. Otherwise, C is undefined in M. Note that the empty clause is falsified by any M.

Let $\mathcal{A} = (A, \oplus, \otimes, \mathbf{0}, \mathbf{1})$ be a semiring. A *weighted clause* is a pair (C, w) such that C is a clause and $w \in A$ with $w <_{\mathcal{A}} \mathbf{1}$ denotes its weight. A *semiring-induced propositional formula* is a set of weighted clauses $F = \{(C_1, w_1), \ldots, (C_e, w_e)\}$[2].

3.2 Semantics

Definition 3. *Consider a formula* $F = \{(C_1, w_1), \ldots, (C_e, w_e)\}$. *Each weighted clause* (C_i, w_i) *defines a function over total assignments,*

$$\phi_i(M) = \begin{cases} w_i & : \quad M \text{ falsifies } C_i \\ \mathbf{1} & : \quad \text{otherwise} \end{cases}$$

[2] For the sake of simplicity, we will restrict ourselves to clausal form formulas. The extension to the general case is direct: just let each C_i be an arbitrary boolean expression.

The formula F defines an evaluation function *as*

$$\phi_{\mathcal{A},F}(M) = \bigotimes_{i=1}^{e} \phi_i(M) \ ,$$

where M is a total assignment and ϕ_i is the function induced by (C_i, w_i). A total assignment M such that $\phi_{\mathcal{A},F}(M) > \mathbf{0}$ is called a model *of F.*

Definition 4. *The pair $(\mathcal{A}, F)$ defines the* marginalization problem *consisting in finding*

$$mrg(\mathcal{A}, F) = \bigoplus_{M \in \mathcal{M}} \phi_{\mathcal{A},F}(M)$$

As we will see, mrg is a very general computational problem. Note that, since $\forall_{a \in A}\ \mathbf{0} \oplus a = a$, only models of F contribute to mrg.

The following two definitions present two important relations among formulas. The effect of the α value on the definitions will be clear in Section 5. For the moment, it is just fine to ignore it or, what is equivalent, assume $\alpha = \mathbf{0}$.

Definition 5. *Let F and F' be two formulas over a common set of propositional symbols, and α a semiring value. We say that F $\mathcal{A}$-implies F' subject to threshold α, noted $F \models_\alpha^{\mathcal{A}} F'$, if for all total assignment M, $\alpha \oplus \phi_{\mathcal{A},F}(M) \leq \alpha \oplus \phi_{\mathcal{A},F'}(M)$.*

Definition 6. *When $F \models_\alpha^{\mathcal{A}} F'$ and $F' \models_\alpha^{\mathcal{A}} F$ we say that F and F' are $\mathcal{A}$-equivalent subject to threshold α and we note it $F \equiv_\alpha^{\mathcal{A}} F'$.*

If $F \models_\alpha^{\mathcal{A}} \{(\square, \mathbf{0})\}$ we say that F is an *α-contradiction*. In Section 5 we will take advantage of the following property.

Property 1. If $F \models_\alpha^{\mathcal{A}} \{(\square, \mathbf{0})\}$ and M is a total assignment, then $\alpha \oplus \phi_{\mathcal{A},F}(M) = \alpha$.

Zero-weighted clauses (i.e, of the form $(C, \mathbf{0})$) are called *hard* clauses. Note that, since $\forall_{a \in A}\ \mathbf{0} \otimes a = \mathbf{0}$, if M is a total assignment that falsifies a hard clause, then M is not a model. If we restrict ourselves to hard clauses and assume $\alpha = \mathbf{0}$, implication and equivalence correspond to the usual definitions in classical propositional logic.

For simplicity we will drop the semiring superscript when there is no ambiguity.

4 Applications

This section is devoted to illustrate the richness of semiring-induced propositional logic. It can be considered semi-tutorial because similar applications have been already identified in different contexts (e.g. [29,27,5,19,6]). Consider a formula $F = \{(C_1, w_1), \ldots, (C_e, w_e)\}$ defined over the symbols P where weights belong to a semiring $\mathcal{A}$.

4.1 Decision Problems ($\mathcal{A}_{\texttt{bool}}$)

If we consider semiring $\mathcal{A}_{\texttt{bool}}$, the weight of all clauses must be zero as $A = \{0, 1\}$ and, by definition, weights are smaller than 1. The corresponding evaluation function

is $\phi_F(M) = \phi_1(M) \wedge \ldots \wedge \phi_e(M)$. It is easy to see that $\phi_F(M) = 1$ iff M satisfies every clause in F. Moreover, the marginalization problem $mrg(F) = \vee_{M \in \mathcal{M}} \ \phi_F(M)$ is 1 iff F is satisfiable. In other words, $mrg(\mathcal{A}_{\texttt{bool}}, F)$ is equivalent to the boolean satisfiability problem (SAT) [24]. As logical consequence and logical equivalence can be reduced to SAT testing, the logic induced by semiring $\mathcal{A}_{\texttt{bool}}$ can also be used to model such problems.

Example 2. Consider a set of three boolean variables $P = \{x_1, x_2, x_3\}$, and the problem of assigning them in such a way that $x_1 = x_2$ and $x_2 \neq x_3$. If we want to know if the problem has any solution (i.e. it is satisfiable) we can use semiring $\mathcal{A}_{\texttt{bool}}$. This problem is encoded in the formula $F = \{(x_1 \vee \neg x_2, 0), (\neg x_1 \vee x_2, 0), (x_2 \vee x_3, 0), (\neg x_2 \vee \neg x_3, 0)\}$. The first column in Figure 2 shows the set of total assignments $\mathcal{M}$. The second column shows, for each assignment M, the value $\phi_F(M)$ of the evaluation function. For instance $\phi_F(x_1, \neg x_2, x_3) = 0$ since this assignment does not satisfy clause $\neg x_1 \vee x_2$. The bottom row shows the result of the marginalization problem $mrg(F)$. In this case it is the logical OR of all $\phi_F(M)$ values. It is 1 as there are assignments for which the evaluation function is 1.

4.2 Summation Problems ($\mathcal{A}_{\texttt{count}}$)

If we consider semiring $\mathcal{A}_{\texttt{count}}$, the corresponding evaluation function and the marginalization problem are $\phi_F(M) = \prod_{i=1}^{e} \phi_i(M)$ and $mrg(F) = \sum_{M \in \mathcal{M}} \phi_F(M)$, respectively. If the weight of all clauses is $w_i = 0 \in \mathbb{R}^+$, then $\phi_F(M) = 1$ iff M satisfies all clauses. So computing $mrg(F)$ is equivalent to the model counting problem, #SAT [4].

Example 3. Consider the same problem as in the previous example. If we want to know the number of solutions we should use semiring $\mathcal{A}_{\texttt{count}}$. The encoding of the problem is the same as before. The third column in Figure 2 shows the values of the evaluation function $\phi_F(M)$. As $\wedge$ and $\times$ are equivalent when restricted to $\{0, 1\}$, the evaluations do not change from the previous example. The last cell of the column shows the result of computing $mrg(F)$. It is 2 as there are two assignments whose evaluation is 1.

Alternatively, if we allow different clauses to have different weights, mrg can model important problems such as the computation of marginals in Bayesian networks [26].

4.3 Optimization Problems ($\mathcal{A}_{\texttt{min}+}$, $\mathcal{A}_{\texttt{max}\times}$)

If we consider semiring $\mathcal{A}_{\texttt{min}+}$, the corresponding evaluation function and marginalization problem are $\phi_F(M) = \sum_{i=1}^{e} \phi_i(M)$ and $mrg(F) = \min_{M \in \mathcal{M}}\{\phi_F(M)\}$, respectively. If the weight of all clauses is $w_i = 1 \in \mathbb{R}^+$, then $\phi_F(M) =$ "number of clauses falsified by M". Therefore computing $mrg(F)$ is equivalent to the problem of maximizing the number of satisfied clauses (Max-SAT) [24].

If we allow different clauses to have different weights, mrg is equivalent to the *partial weighted* Max-SAT problem [24, 20], which models a variety of interesting additive optimization problems with applications in bioinformatics, circuit design, electronic markets, resource allocation, etc. [16].

Example 4. Consider the same problem as in the previous examples. Let $\mathcal{P}$ be the set of models of the formula F. Suppose we want to find the model with the least number of variables set to true, $\min_{M\in\mathcal{P}} \sum_{1\leq i\leq 3} x_i$. We can express this problem with semiring $\mathcal{A}_{\texttt{min+}}$. The clauses already existing in the previous example should be made hard and for each symbol x_i a unit clause $(\neg x_i, 1)$ should be added. The fourth column in Figure 2 shows the values of the resulting evaluation function $\phi_F(M)$. For instance, $\phi_F(\{x_1, x_2, \neg x_3\}) = 2$ as $\{x_1, x_2, \neg x_3\}$ falsifies clauses $(\neg x_1, 1)$ and $(\neg x_2, 1)$. The bottom cell of the column shows $mrg(F)$. In this case $mrg(F) = 1$, the minimum over all evaluations of total assignments.

Semirings $\mathcal{A}_{\texttt{max}\times}$ and $\mathcal{A}_{\texttt{min+}}$ are isomorphic since we can transform the former into the latter via a logarithmic mapping [25]. Therefore, they have the same expressive power. Nevertheless, semiring $\mathcal{A}_{\texttt{max}\times}$ seems to be a more natural choice for modeling probabilistic problems such as the most probable explanation problem (MPE) in Bayesian networks [26] or the MAP inference problem in Markov random fields [18] with applications in diagnosis, vision, signal encoding, etc.

4.4 Multi-criteria Optimization Problems ($\mathcal{A}^n$, $\mathcal{A}^f$)

Consider the multidimensional extension $\mathcal{A}^n$ of a semiring $\mathcal{A}=(A, \oplus, \otimes, \mathbf{0}, \mathbf{1})$. Weights are now n-dimension vectors, $w_i = (w_i^1, \ldots, w_i^n)$. It is easy to see that the resulting evaluation function satisfies $\phi_{\mathcal{A}^n,F}(M) = (\phi_{\mathcal{A},F^1}(M), \ldots, \phi_{\mathcal{A},F^n}(M))$, where $F^j = \{(C_1, w_1^j), \ldots, (C_e, w_e^j)\}$ is the projection of F onto the j-th vector dimension. In words, the evaluation function treats each dimension independently from the others. Further, the marginalization satisfies $mrg(\mathcal{A}^n, F) = (mrg(\mathcal{A}, F^1), \ldots, mrg(\mathcal{A}, F^n))$, which corresponds to the independent marginalization problem of each dimension. Therefore, one can use $\mathcal{A}^n$ to model (and, as we will see in Section 5, solve) in one shot n independent problems over the same set of symbols.

Example 5. Consider again our running example, now with two objectives. The first one, as in the previous example, is to minimize the variables set to true. The second one is to minimize the number of variable pairs simultaneously set to false. Formally,

$$\min_{M\in\mathcal{P}} \Big(\sum_{1\leq i\leq 3} x_i, \sum_{1\leq i<j\leq 3} (1-x_i)(1-x_j) \Big) .$$

We can model it using semiring $\mathcal{A}^2_{\texttt{min+}}$. Note that its zero and unit elements are (∞, ∞) and $(0, 0)$, respectively. Hard clauses must have the new zero element,

$$\{(x_1 \vee \neg x_2, (\infty, \infty)), (\neg x_1 \vee x_2, (\infty, \infty)), (x_2 \vee x_3, (\infty, \infty)), (\neg x_2 \vee \neg x_3, (\infty, \infty))\} .$$

The first objective can be encoded with the following set of clauses,

$$\{(\neg x_1, (1, 0)), (\neg x_2, (1, 0)), (\neg x_3, (1, 0))\} .$$

M	$\mathcal{A}_{\texttt{bool}}$	$\mathcal{A}_{\texttt{count}}$	$\mathcal{A}_{\texttt{min}+}$	$\mathcal{A}^2_{\texttt{min}+}$	$(\mathcal{A}^2_{\texttt{min}+})^f$
$\{x_1, x_2, x_3\}$	0	0	∞	(∞, ∞)	$\{(\infty, \infty)\}$
$\{x_1, x_2, \neg x_3\}$	1	1	2	$(2, 0)$	$\{(2, 0)\}$
$\{x_1, \neg x_2, x_3\}$	0	0	∞	(∞, ∞)	$\{(\infty, \infty)\}$
$\{x_1, \neg x_2, \neg x_3\}$	0	0	∞	(∞, ∞)	$\{(\infty, \infty)\}$
$\{\neg x_1, x_2, x_3\}$	0	0	∞	(∞, ∞)	$\{(\infty, \infty)\}$
$\{\neg x_1, x_2, \neg x_3\}$	0	0	∞	(∞, ∞)	$\{(\infty, \infty)\}$
$\{\neg x_1, \neg x_2, x_3\}$	1	1	1	$(1, 1)$	$\{(1, 1)\}$
$\{\neg x_1, \neg x_2, \neg x_3\}$	0	0	∞	(∞, ∞)	$\{(\infty, \infty)\}$
$mrg(\mathcal{A}, F)$	1	2	1	$(1, 0)$	$\{(2, 0), (1, 1)\}$

Fig. 2. Evaluation functions and marginalization problems induced by different semirings. The set of symbols is $P = \{x_1, x_2, x_3\}$. The set of clauses changes from one case to another.

The second objective can be encoded with the following set of clauses,

$$\{(x_1 \vee x_2, (0, 1)), (x_1 \vee x_3, (0, 1)), (x_2 \vee x_3, (0, 1))\} .$$

The fifth column in Figure 2 shows the values of the corresponding evaluation function. For instance, $\phi_F(\{\neg x_1, \neg x_2, x_3\}) = (1, 1)$ as it falsifies clauses $(\neg x_3, (1, 0))$ and $(x_1 \vee x_2, (0, 1))$, and $(1, 0) +^2 (0, 1) = (1, 1)$. The last cell of the column shows the result of the marginalization problem. In this case it is the point-wise minimum over all the column entries.

Given a formula $F = \{(C_1, w_1), \ldots, (C_e, w_e)\}$ its frontier extension is $F' = \{(C_1, w'_1), \ldots, (C_e, w'_e)\}$, where $w'_j = \{w_j\}$. It can be proved [6] that $\phi_{\mathcal{A}^f, F'}(M) = \{\phi_{\mathcal{A},F}(M)\}$ and

$$mrg(\mathcal{A}^f, F') = \{\phi_{\mathcal{A},F}(M) \mid \forall_{N \in \mathcal{M}}, \phi_{\mathcal{A},F}(M) \not\leq \phi_{\mathcal{A},F}(N)\},$$

which is the set of maximal evaluations of $\phi_{\mathcal{A},F}(M)$.

An immediate application is to model a multi-objective problem with n objectives with semiring $(\mathcal{A}^n)^f$. Each objective is encoded in one dimension of $\mathcal{A}^n$. The marginalization problem corresponds to the so-called *efficient frontier* of the problem, which is the most general notion of optimality in multi-objective optimization.

Example 6. If we want to compute the efficient frontier of the previous bi-objective problem, we can use the frontier extension of the previous semiring, $(\mathcal{A}^2_{\texttt{min}+})^f$, and replace vector weights by singleton vector weights. The sixth column in Figure 2 shows the values of the corresponding evaluation function. The last cell of the column shows the result of the corresponding marginalization problem. In this case it is the union of all the values followed by the removal of the non optimal elements, which is the efficient frontier of the original bi-objective optimization problem.

5 A DPLL Algorithm for Semiring-Induced Propositional Logic

In this section we show how DPLL for satisfiability testing and its most prominent enhancements can be naturally generalized to compute the marginalization problem of a semiring-induced propositional logic formula. Following [23], we describe the algorithm using a transition system.

5.1 Transition Systems

We will model our semiring-induced DPLL procedures by means of a set of *states* together with a binary relation $\Rightarrow$ over these states, called the *transition relation*. If $S \Rightarrow S'$ we say that there is a *transition* from S to S'. We call any sequence of transitions of the form $S_0 \Rightarrow S_1,\ S_1 \Rightarrow S_2,\ \ldots$ a *derivation*, and denote it by $S_0 \Rightarrow S_1 \Rightarrow S_2 \Rightarrow \ldots$ We call any subsequence of a derivation a *subderivation*.

In what follows, transition relations will be defined by means of conditional *transition rules*. For a given state S, a transition rule precisely defines whether there is a transition from S by this rule and, if so, to which state S'. Such a transition is called an *application step* of the rule.

A *transition system* is a set of transition rules defined over some given set of states. Note that if more than one transition is possible from S, any option is valid. If there is no transition from S, we will say that S is *final*.

5.2 States in SDPLL Transition Systems

Semiring-induced DPLL (SDPLL) can be fully described by simply considering that a state of the procedure is of the form (α, M, F), where F is a formula, M is a (partial) assignment and $\alpha \in A$ is a semiring element. Additionally, we define states of the form $(\alpha, done, F)$ to represent final states.

More precisely, M is a *sequence* of literals, never containing both a literal and its negation. Each literal has an *annotation*, a bit that marks it as a *decision* literal or not. Essentially, a decision literal is a literal that is added in the context of a split case and, at some point, its negation needs to be considered. The concatenation of two such sequences will be denoted by simple juxtaposition. When we want to emphasize that a literal l is a decision literal we will write it as l^{d}. We will denote the empty sequence of literals (or the empty assignment) by $\emptyset$.

Adding a literal l to M means that we are conditioning the formula F with l. This is equivalent to adding a unit hard clause $(l, \mathbf{0})$ to F. Accordingly, we will frequently consider M as a set of unit hard clauses, ignoring the annotations, the order between its elements and assuming an implicit zero weight. Therefore, $M \cup F$ with $M = l_1 \ldots l_n$ will be a shorthand for $\{(l_1, \mathbf{0}), \ldots (l_n, \mathbf{0})\} \cup F$.

5.3 The Basic SDPLL Procedure

A basic backtracking-based algorithm that enumerates all total assignments can be defined with the following three rules. Since none of them changes the input formula F, we do not include it in the state descriptions.

Definition 7. *Backtracking-based enumeration rules.*

$$\textit{Decide}: \quad (\alpha, M) \Rightarrow (\alpha, Ml^{\mathsf{d}}) \quad \textbf{if} \ \{ \ l \textit{ is undefined in } M$$

$$\textit{BacktrackSuccess}: (\alpha, Ml^{\mathsf{d}}N) \Rightarrow (\alpha', M\neg l) \ \textbf{if} \begin{cases} \alpha' = \alpha \oplus \phi_F(M \ l^{\mathsf{d}} \ N) \\ M \ l^{\mathsf{d}} \ N \textit{ is total} \\ N \textit{ has no decision literals} \end{cases}$$

$$\textit{DoneSuccess}: \quad (\alpha, M) \Rightarrow (\alpha', \textit{done}) \ \textbf{if} \begin{cases} \alpha' = \alpha \oplus \phi_F(M) \\ M \quad \textit{is total} \\ M \textit{ has no decision literals} \end{cases}$$

One can use the previous system for solving a semiring-induced marginalization problem $mrg(F)$ by simply generating an arbitrary derivation $(\mathbf{0}, \emptyset) \Rightarrow \ \ldots \ \Rightarrow \ (\alpha, done)$. The α value of the final state is the solution of the marginalization problem. The algorithm generates all total assignments M and adds their $\phi_F(M)$ contribution to the semiring value.

Consider an arbitrary state (α, M). If M is a partial assignment, rule Decide models the split case. The assignment M is extended with a so far undefined literal l. The literal is annotated as a decision literal to denote that once all the extensions of Ml have been taken into account, the extensions of $M\neg l$ must still be considered. If M is a total assignment, then the contribution of $\phi_F(M)$ must be added to α (i.e, $\alpha' = \alpha \oplus \phi_F(M)$). If M does not contain any decision literals, it means that all total assignments have already been considered, so the value of α is the final result and we can end the application of rules. This is the situation considered by the DoneSuccess rule. Alternatively, if M contains decision literals the algorithm backtracks by replacing the most recent decision literal by its negation and removing all subsequent literals in M. This is the situation considered by the BacktrackSuccess rule. Clearly, the algorithm terminates in a finite number of steps and, in a final state, α contains the result of the marginalization problem. Note that the α value increases monotonically during the execution.

Example 7. Consider an arbitrary intermediate state (α, M). If the semiring is $\mathcal{A}_{bool}$, then $\alpha = 1$ iff "some total assignment in a previous state was a model". If the semiring is $\mathcal{A}_{count}$, then $\alpha =$"number of models found so far". If the semiring is $\mathcal{A}_{min+}$, then $\alpha =$"evaluation of the best (i.e, minimum cost) model so far". Finally, if the semiring is $\mathcal{A}^{nf}_{min+}$, then $\alpha =$"set of evaluations of pareto-optimal models with respect to already inspected total assignments".

Obviously, the previous algorithm is extremely inefficient. It can be improved with the addition of *pruning*. We say that state (α, M) is *in a conflict* if $M \cup F$ is an α-contradiction (i.e, $M \cup F \models_\alpha \{(\square, \mathbf{0})\}$). Property 1 indicates that the algorithm can discard (i.e. prune) all the extensions of the assignment M since they will not contribute to the solution. Pruning is specified with the following rules.

Definition 8. *The basic SDPLL algorithm is the system defined by the previous three and the following three pruning rules.*

$$\textit{Propagate}: \quad (\alpha, M) \Rightarrow (\alpha, Ml) \quad \textbf{if} \begin{cases} (M\neg l) \cup F \models_\alpha \{(\square, \mathbf{0})\} \\ l \textit{ is undefined in } M \end{cases}$$

$$\textit{BacktrackFail}: (\alpha, Ml^{\mathsf{d}}N) \Rightarrow (\alpha, M\neg l) \textbf{ if} \begin{cases} (Ml^{\mathsf{d}}N) \cup F \models_\alpha \{(\square, \mathbf{0})\} \\ N \textit{ has no decision literals} \end{cases}$$

$$\textit{DoneFail}: \quad (\alpha, M) \Rightarrow (\alpha, \textit{done}) \textbf{ if} \begin{cases} M \cup F \models_\alpha \{(\square, \mathbf{0})\} \\ M \textit{ has no decision literals} \end{cases}$$

Rule DoneFail considers the case in which the current assignment M is in a conflict and does not contain any decision literal. In that case we can end the execution. Rule BacktrackFail considers the case in which M is in a conflict and contains decision literals. In that case the algorithm backtracks by replacing the most recent decision literal by its negation and removing all subsequent literals of M. Rule Propagate considers the case when M is not in a conflict, but $M\neg l$ is. In that case, the algorithm extends M with l. Note that l is not marked as a decision, as its negation needs not to be considered.

Unlike the three earlier rules, the applicability of the pruning rules is not easy to check since detecting conflicts is in general NP-hard. Therefore, practical algorithms rely on sufficient conditions that can be efficiently computed. The following property presents a very naive, but still widely used one.

Property 2. Consider a transition state (α, M). Let $V \subseteq F$ be the set of clauses falsified by M, and $\beta = \bigotimes_{(C,w)\in V} w$ the product of costs of violated clauses. Then, $\alpha \oplus \beta = \alpha$ is a sufficient condition for $M \cup F \models_\alpha \{(\square, \mathbf{0})\}$.

Example 8. Consider the simplest semiring $\mathcal{A}_{bool}$. The previous pruning condition becomes $\alpha = 1 \vee V \neq \varnothing$. It occurs when either a model has already been found or the current assignment M falsifies some clause. With semiring $\mathcal{A}_{\min+}$, the previous pruning condition becomes $\min\{\alpha, \beta\} = \alpha$. It will occur if either $\alpha = 0$ (a solution that cannot be improved has already been found) or $\alpha \leq \beta$ (the current assignment M is already worse than the best solution found so far). Consider the more complex semiring $(\mathcal{A}^2_{\min+})^f$ (i.e, the frontier extension of a two-dimensional optimization semiring). In this case, α is the set of optimal evaluations found so far and $\beta = \{(w_1, w_2)\}$ is a singleton that adds up the violations of M. The pruning condition is $||\alpha \cup \{(w_1, w_2)\}|| = \alpha$, which occurs when (w_1, w_2) is dominated by some element of α.

Observe that when Propagate can be applied, so is Decide. For efficiency reasons, it is desirable to always chose Propagate.

The SDPLL algorithm using the previous pruning condition is a faithful generalization of several algorithms: If the semiring is $\mathcal{A}_{bool}$ it becomes DPLL [10] for satisfiability testing. If the semiring is $\mathcal{A}_{count}$ it becomes the algorithm in [4] for model counting. If the semiring is $\mathcal{A}_{min+}$ it becomes the algorithm in [7] for Max-SAT. Finally, if the semiring is $(\mathcal{A}^n_{min+})^f$ it becomes essentially equivalent to the algorithm described in [14] for multi-objective optimization.

6 Extending SAT Techniques

In the previous section we showed how our formalism allows one to unify several basic enumeration algorithms by abstracting away algorithmic details. Here we show its convenience for generalizing more sophisticated techniques. We consider three features common to all modern DPLL-based SAT solvers: backjumping, learning and restarts.

6.1 Backjumping, Learning and Restarts

The purpose of *backjumping* is to undo *several* decisions at once, going back to a lower decision level than the previous level and adding some new literal to that lower level. The Backjump rule below models this idea.

$$\textsf{Backjump}: \quad (\alpha, Ml^{\mathsf{d}}N) \Rightarrow (\alpha', Ml') \textbf{ if } \begin{cases} \text{there is a previous state} \\ (\alpha', M) \text{ such that:} \\ (M\neg l') \cup F \models_{\alpha'} \{(\Box, \mathbf{0})\} \\ l' \text{ is undefined in } M \end{cases}$$

It can be seen as a delayed propagation. In words, the rule is triggered if the algorithm detects that a propagation instead of a decision could have been done at an earlier state (α', M). This occurs when the condition for propagation, i.e. $(M\neg l') \cup F \models_{\alpha'} \{(\Box, \mathbf{0})\}$, was not detected at the earlier state (recall that simple sufficient conditions are usually used) but can be detected now (possibly, from an analysis of the current state). Note that $\alpha' \leq \alpha$ as α' is taken from a previous state. Therefore with the Backjump rule the semiring value does not grow monotonically anymore during the execution of the algorithm.

The purpose of *learning* is to make explicit in the original formula implicit clauses, because they may help in the future identification of conflicts. Similarly, when a clause seems not to be useful for that purpose according to some measure, it can be removed. The Learn and Forget rules below generalize this idea. Since these rules modify the formula F, we add it to the state description.

$$\textsf{Learn}: \quad (\alpha, M, F) \Rightarrow (\alpha, M, F \cup \{(C, w)\}) \textbf{ if } \left\{ F \models_{\mathbf{0}} F \cup \{(C, w)\} \right.$$

$$\textsf{Forget}: (\alpha, M, F \cup \{(C, w)\}) \Rightarrow (\alpha, M, F) \quad \textbf{if } \left\{ F \models_{\mathbf{0}} F \cup \{(C, w)\} \right.$$

Observe that the Learn and Forget rules allow one to add and remove from the current formula F an arbitrary clause C as long as it is $\mathbf{0}$-entailed by F. The addition and removal is safe, even in combination with Backjump and Restart (to be seen later) which decrease the α value, precisely because the entailment is required with respect to the lowest possible α value.

Finally, it may be useful to *restart* the DPLL procedure whenever the search is not making enough progress according to some measure. The rationale behind this idea is that upon each restart, the additional knowledge of the search space compiled into the newly learned clauses will lead the heuristics for Decide to behave differently, and possibly in a wiser way. The following rule models this idea.

$$\textsf{Restart}: \quad (\alpha, M, F) \;\Rightarrow\; (\mathbf{0}, \emptyset, F)$$

If Learn and Forget are applied, termination of the procedure can be achieved by avoiding infinite subderivations with only Learn and Forget steps. On the other hand, if the Restart rule is also applied, in order to get termination in practice one periodically increases the minimum number of applications of the other rules between each pair of restart steps. This is formalized below.

Definition 9. *Let us consider a derivation by the basic SDPLL rules together with the Backjump, Learn, Forget and Restart rules. We say that Restart* has increasing periodicity *in the derivation if, for each subderivation $S_i \Rightarrow \ldots \Rightarrow S_j \Rightarrow \ldots \Rightarrow S_k$ where the steps producing S_i, S_j, and S_k are the only Restart steps, the number of steps of the other rules in $S_i \Rightarrow \ldots \Rightarrow S_j$ is strictly smaller than in $S_j \Rightarrow \ldots \Rightarrow S_k$.*

Finally, the following theorem shows how the SDPLL algorithm can be used to effectively compute the marginalization of a given formula.

Theorem 1. *Let us consider the basic SDPLL rules together with the Backjump, Learn, Forget and Restart rules. If infinite subderivations consisting of only Learn and Forget steps are not allowed and Restart has increasing periodicity, any derivation $(\mathbf{0}, \emptyset, F) \Rightarrow \ldots \Rightarrow S$ is finite. Moreover, if S is final then it is of the form $(mrg(F), done, G)$.*

6.2 Conflict-Driven Backjumping and Learning

The previous four rules have been presented in their most general form. In principle, they can be applied independently. Still, the experience from modern SAT solvers is that it is their combination what produces the best results. Besides, their application should be *driven by conflicting states*. In our framework the description of this idea requires the specialization of the Backjump rule as follows.

ConflictDrivenBackjump :

$$(\alpha, Ml^{\mathsf{d}}N) \;\Rightarrow\; (\alpha', Ml') \qquad \textbf{if} \begin{cases} Ml^{\mathsf{d}}N \cup F \models_{\alpha} \{(\Box, \mathbf{0})\} \\ \text{there exists a previous state} (\alpha', M) \text{ and} \\ \text{some clause } l_1 \vee \cdots \vee l_n \vee l' \text{ such that:} \\ \quad F \models_{\alpha'} \{(l_1 \vee \cdots \vee l_n \vee l', \mathbf{0})\} \\ \quad \forall_{1 \leq i \leq n},\ M \cup F \models_{\alpha'} \{(\neg l_i, \mathbf{0})\} \\ \quad l' \text{ is undefined in } M \end{cases}$$

We call the clause $l_1 \vee \ldots \vee l_n \vee l'$ in ConflictDrivenBackjump a *backjump* clause. This rule is more specific than the previous Backjump because it can be only applied when the current state is a conflict. Furthermore, the analysis of the conflict has to reveal the existence of a backjump clause. It can be easily proved that the two conditions for a backjump clause imply the condition $(M\neg l') \cup F \models_{\alpha'} \{(\Box, \mathbf{0})\}$ of the more general Backjump rule. Conflict-driven-learning restricts the Learn and Forget rules to add and remove only backjump clauses, which in turn restricts the new knowledge after each restart.

Conflict analysis [31] is the efficient detection of useful backjump clauses. It is only well-studied in the SAT case. However, the following example shows that our abstract description provides direct generalizations to other problems.

Example 9. Consider semiring $\mathcal{A}_{min+}$ and a formula with, among others, the following clauses, $\{(\neg x_1 \vee x_2, \infty), (\neg x_2 \vee x_3, 8), (\neg x_4 \vee x_5, 7), (\neg x_7 \vee x_8, \infty), (\neg x_7 \vee \neg x_8, 7), (\neg x_2, 1), (\neg x_3, 1), (\neg x_5, 1)\}$. Suppose that an execution of SDPLL that uses the pruning condition of Prop. 2 has generated state $(9, M)$ such that M does not violate any clause and $x_1, \ldots, x_8$ are undefined in M. A possible subderivation is

$$\ldots \Rightarrow (9, M) \Rightarrow (9, Mx_1^d) \Rightarrow (9, Mx_1^d x_2) \Rightarrow (9, Mx_1^d x_2 x_3) \Rightarrow (9, Mx_1^d x_2 x_3 x_4^d) \Rightarrow$$
$$\Rightarrow (9, Mx_1^d x_2 x_3 x_4^d x_5) \Rightarrow (9, Mx_1^d x_2 x_3 x_4^d x_5 x_6^d) \Rightarrow\Rightarrow (9, Mx_1^d x_2 x_3 x_4^d x_5 x_6^d x_7^d) \Rightarrow$$
$$\Rightarrow (9, Mx_1^d x_2 x_3 x_4^d x_5 x_6^d x_7^d x_8)$$

The current state is in a conflict since it satisfies the pruning condition (the sum of weights of clauses falsified by the current assignment is 10 and $\min\{9, 10\} = 9$). If we analyze the conflict, we observe that decisions x_4^d and x_6^d are irrelevant for the pruning condition (if we remove their contribution and the contribution of their implications, the state is still in a conflict). From the analysis of the conflict, we can obtain backjump clause $\neg x_2 \vee \neg x_7$. The application of the rule ConflictDrivenBackjump produces the following state $(9, Mx_1^d x_2 x_3 \neg x_7)$.

It is important to note that state-of-the-art Max-SAT solvers [16,21] are very naive in terms of backjumping (they only backjump when the conflict is exclusively caused by hard clauses). Thus, our backjump rule does not only cover this basic case, but also opens a new perspective for new cases as the one in this example.

6.3 Semirings with Idempotent $\oplus$

Semirings with idempotent $\oplus$ include all applications discussed in Section 4, except for counting problems ($\mathcal{A}_{count}$). Although the purpose of this paper is to consider general definitions and techniques, the $\oplus$-idempotent case is still so general that we will mention some algorithmic improvements for it. They are all based on the fact that during the execution of SDPLL, it is always possible to replace the semiring value of a previous state by the higher semiring value of the current state and resume the execution from that earlier state. Formally,

Property 3. Let F be a formula defined over a $\oplus$-idempotent semiring. Consider an arbitrary derivation of the basic SDPLL (without the enhancements of Section 6.1), $(\mathbf{0}, \varnothing) \Rightarrow (\alpha_1, M_1) \Rightarrow \ldots \Rightarrow (\alpha_j, M_j)$. For any $1 \leq i \leq j$, any derivation from the state (α_j, M_i) to a final state $(\alpha_n, done)$ satisfies that $\alpha_n = mrg(F)$.

The first implication of this is that in the Backjump (and ConflictDrivenBackjump) rule the occurrences of α' can be replaced by α. Therefore, the rule models a return to an earlier state but preserving the better semiring value of the current state. This allows one to take advantage of the work done so far for, e.g., propagating unassigned literals or detecting new conflicts after backjumping. An interesting feature of the resulting rule is that ConflictDrivenBackjump subsumes chronological backtracking, as the negation of all decision literals of the current assignment M is a backjump clause.

The second implication of Property 3 is that the Restart rule can restart the search preserving the semiring value of the current state. Finally, the Learn and Forget rules can add and remove α-implied clauses where α is not $\mathbf{0}$ but the semiring value of the current state, which broadens the range of clauses that can be used for learning.

7 Conclusions and Future Work

In this paper we introduce semiring-induced propositional logic, which extends propositional logic by allowing clauses to be weighted with semiring values. We show that it provides a convenient formalism for modeling a variety of important computational problems. Further, it serves as an elegant and well-defined presentation of general solving techniques by focusing on the general idea and abstracting away algorithmic details.

In our future work we want to incorporate to the SDPLL algorithm decomposition techniques, which have proven fundamental in counting problems [17,2].

This paper has focused on enumeration-based algorithmic techniques. We want to investigate which inference-based algorithms can also be unified. In particular, we want to study under which conditions the resolution rule for Max-SAT introduced in [20] can be generalized to arbitrary semirings, while preserving completeness.

References

1. Aji, S., McEliece, R.: The generalized distributive law. IEEE Trans. on Information Theory 46(2), 325–343 (2000)
2. Bacchus, F., Dalmao, S., Pitassi, T.: Algorithms and complexity results for #SAT and Bayesian inference. In: FOCS, pp. 340–351 (2003)
3. Bertele, U., Brioschi, F.: Nonserial Dynamic Programming. Academic Press, London (1972)
4. Birnbaum, E., Lozinskii, E.: The good old Davis-Putnam procedure helps counting models. JAIR 10, 457–477 (1999)
5. Bistarelli, S., Montanari, U., Rossi, F.: Semiring-based constraint satisfaction and optimization. JACM 44(2), 201–236 (1997)
6. Bistarelli, S., Gadducci, M., Larrosa, J., Rollon, E.: A semiring-based approach to multi-objective optimization. In: Proc. of Intl. Workshop on Soft Constraints and Preferences (2008)
7. Borchers, B., Furman, J.: A two-phase exact algorithm for MAX-SAT and weighted MAX-SAT problems. Journal of Combinatorial Optimization 2, 299–306 (1999)
8. Cooper, M., Schiex, T.: Arc consistency for soft constraints. Artif. Intell. 154(1-2), 199–227 (2004)
9. Crama, Y., Hansen, P., Jaumard, B.: The basic algorithm for pseudo-boolean programming revisited. Discrete Applied Mathmatics 29, 171–185 (1990)
10. Davis, M., Logemann, G., Loveland, G.: A machine program for theorem proving. Communications of the ACM 5, 394–397 (1962)
11. Davis, M., Putnam, H.: A computing procedure for quantification theory. JACM 3 (1960)
12. Dechter, R., Pearl, J.: Network-based heuristics for constraint-satisfaction problems. Artif. Intell. 34, 1–38 (1988)
13. Fu, Z., Malik, S.: On solving the partial Max-SAT problem. In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 252–265. Springer, Heidelberg (2006)
14. Gavanelli, M.: An algorithm for multi-criteria optimization in CSPs. In: ECAI, pp. 136–140 (2002)
15. Golan, J.: Semirings and their applications. Kluwer Academic Publishers, Dordrecht (1999)
16. Heras, F., Larrosa, J., Oliveras, A.: MiniMaxSAT: An efficient Weighted Max-SAT Solver. JAIR 31, 1–32 (2008)
17. Bayardo, R., Pehoushek, J.: Counting models using connected components. In: AAAI/IAAI, pp. 157–162 (2000)

18. Kindermann, R., Snell, L.: Markov Random Fields and Their Applications. AMS, Providence (1980)
19. Kohlas, J., Wilson, N.: Semiring induced valuation algebras: Exact and approximate local computation algorithms. Artif. Intell. 172(11), 1360–1399 (2008)
20. Larrosa, J., Heras, F., de Givry, S.: A logical approach to efficient Max-SAT solving. Artif. Intell. 172(2-3), 204–233 (2008)
21. Li, C., Manyà, F., Planes, J.: New inference rules for Max-SAT. JAIR 30, 321–359 (2007)
22. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an Efficient SAT Solver. In: DAC 2001, pp. 530–535. ACM Press, New York (2001)
23. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT modulo theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL. JACM 53(6), 937–977 (2006)
24. Papadimitriou, C.: Computational Complexity. Addison-Wesley, USA (1994)
25. Park, J.: Using weighted Max-SAT engines to solve MPE. In: AAAI 2002, pp. 682–687 (2002)
26. Pearl, J.: Probabilistic Inference in Intelligent Systems. Networks of Plausible Inference. Morgan Kaufmann, San Mateo (1988)
27. Schiex, T., Fargier, H., Verfaillie, G.: Valued constraint satisfaction problems: hard and easy problems. In: IJCAI 1995, pp. 631–637 (1995)
28. Shafer, G.R., Shenoy, P.P.: Probability propagation. Anals of Mathematics and Artificial Intelligence 2, 327–352 (1990)
29. Shenoy, P., Shafer, G.: Axioms for probability and belief-function propagation. In: UAI 1988 (1988)
30. Xing, Z., Zhang, W.: Maxsolver: An efficient exact algorithm for (weighted) maximum satisfiability. Artif. Intell. 164(1-2), 47–80 (2005)
31. Zhang, L., Madigan, C.F., Moskewicz, M.W., Malik, S.: Efficient Conflict Driven Learning in a Boolean Satisfiability Solver. In: ICCAD 2001, pp. 279–285 (2001)

Dafny: An Automatic Program Verifier for Functional Correctness

K. Rustan M. Leino

Microsoft Research, Redmond, WA, USA
leino@microsoft.com

Abstract. Traditionally, the full verification of a program's functional correctness has been obtained with pen and paper or with interactive proof assistants, whereas only reduced verification tasks, such as extended static checking, have enjoyed the automation offered by satisfiability-modulo-theories (SMT) solvers. More recently, powerful SMT solvers and well-designed program verifiers are starting to break that tradition, thus reducing the effort involved in doing full verification.

This paper gives a tour of the language and verifier Dafny, which has been used to verify the functional correctness of a number of challenging pointer-based programs. The paper describes the features incorporated in Dafny, illustrating their use by small examples and giving a taste of how they are coded for an SMT solver. As a larger case study, the paper shows the full functional specification of the Schorr-Waite algorithm in Dafny.

0 Introduction

Applications of program verification technology fall into a spectrum of assurance levels, at one extreme proving that the program lives up to its functional specification (*e.g.*, [8,23,28]), at the other extreme just finding some likely bugs (*e.g.*, [19,24]). Traditionally, program verifiers at the high end of the spectrum have used interactive proof assistants, which require the user to have a high degree of prover expertise, invoking tactics or guiding the tool through its various symbolic manipulations. Because they limit which program properties they reason about, tools at the low end of the spectrum have been able to take advantage of satisfiability-modulo-theories (SMT) solvers, which offer some fixed set of automatic decision procedures [18,5].

An SMT-based program verifier is automatic in that it requires no user interaction with the solver. This is not to say it is effortless, for it usually requires interaction at the level of the program being analyzed. Used analogously to type checkers, the automatic verifier takes a program with specifications (analogously, type annotations) and produces error messages about where the program may be violating a rule of the language (like an index bounds error) or a programmer-supplied specification (like a failure to establish a declared postcondition). The error messages speak about the program (*e.g.*, "`MyProgram.dfy(212,23): loop invariant might not hold on entry`") and can be computed continuously in the

E.M. Clarke and A. Voronkov (Eds.): LPAR-16, LNAI 6355, pp. 348–370, 2010.

background of an integrated development environment. Compared with the use of an interactive proof assistant, the added automation and the interaction closer to the programmer's domain stand a chance of reducing the effort involved in using the verifier and reducing the amount of expertise needed to use it.

Recently, some functional-correctness verification has been performed using automatic program verifiers based on SMT solvers or other automatic decision procedures [45,52,14,33]. This has been made possible by increased power and speed of state-of-the-art SMT solvers and by carefully crafted program verifiers that make use of the SMT solver. For example, the input language for programs and specifications affects how effective the verifier is. Moreover, SMT solvers obtain an important kind of user extensionality by supporting quantifiers, and these quantifiers are steered by matching triggers. The design of good triggers is a central ingredient in making effective use of SMT solvers (for various issues in using matching triggers, see [34]).

In this paper, I describe the language and verifier Dafny. The language is imperative, sequential, supports generic classes and dynamic allocation, and builds in specification constructs (as in Eiffel [42], JML [29], and Spec# [4]). The specifications include standard pre- and postconditions, framing constructs, and termination metrics. Devoid of convenient but restricting ownership constructs for structuring the heap, the specification style (based on *dynamic frames* [27]) is flexible, if sometimes verbose. To aid in specifications, the language includes user-defined mathematical functions and ghost variables. These features permit programs to be specified for *modular verification*, so that the separate verification of each part of the program implies the correctness of the whole program. Finally, in addition to class types, the language supports sets, sequences, and algebraic datatypes.

Dafny's basic features and statements are presented in Marktoberdorf lectures notes [33]. Those lecture notes give a detailed account of the logical encoding of Dafny, including the modeling of the heap and objects, methods and statements, and user-defined functions. The additional features described in this paper and present in the current implementation of the language and verifier include generic types, algebraic datatypes, ghost constructs, and termination metrics.

Dafny's program verifier works by translating a given Dafny program into the intermediate verification language Boogie 2 [2,40,32] in such a way that the correctness of the Boogie program implies the correctness of the Dafny program. Thus, the semantics of Dafny are defined in terms of Boogie (a technique applied by many automatic program verifiers, *e.g.*, [14,21]). The Boogie tool is then used to generate first-order verification conditions that are passed to a theorem prover, in particular to the SMT solver Z3 [17].

Dafny has been used to specify and verify some challenging algorithms. The specifications are understandable and verification times are usually low. To showcase the composition of Dafny's features, I describe the Schorr-Waite algorithm in Dafny. In fact, I include its entire program text (including its full functional correctness specifications), which as far as I know is a first in a conference paper.

Feeding this program through the Dafny verifier, the verification takes less than 5 seconds.

Dafny is available as open source at boogie.codeplex.com.

1 Dafny Language Features

Dafny is an imperative, class-based language. In this section, I describe some of its more advanced features and sketch how these are encoded in Boogie 2 by the Dafny verifier.

In principle, the language can be compiled to executable code, but the current implementation includes only a verifier, not a compiler. For a verified program, a compiler would not need to generate any run-time representation for specifications and ghost state, which are needed only for the verification itself.

1.0 Types

The types available in Dafny are booleans, mathematical integers, (possibly null) references to instances of user-defined generic classes, sets, sequences, and user-defined algebraic datatypes. There is no subtyping, except that all class types are subtypes of the built-in type **object**. Programs are type safe, that is, the static type of an expression accurately describes the run-time values to which the expression can evaluate. Generic type instantiations and types of local variables are inferred. Because of the references and dynamic allocation, Dafny can be used to write interesting pointer algorithms. Sets are especially useful when specifying framing (described below), and sequences and algebraic datatypes are especially useful when writing specifications for functional correctness (more about that below, too).

1.1 Pre- and Postconditions

Methods have standard declarations for preconditions (keyword **requires**) and postconditions (keyword **ensures**), like those in Eiffel, JML, and Spec#. For example, the following method declaration promises to set the output parameter r to the integer square root of the input parameter n, provided n is non-negative.

```
method ISqrt(n: int) returns (r: int)
  requires 0 ≤ n;
  ensures r*r ≤ n ∧ n < (r+1)*(r+1);
{ /* method body goes here... */ }
```

It is the caller's responsibility to establish the precondition and the implementation's responsibility to establish the postcondition. As usual, failure by either side to live up to its responsibility is reported by the verifier as an error.

1.2 Ghost State

A simple and useful feature of the language is that variables can be marked as **ghost**. This says that the variables are used in the verification of the program but are not needed at run time. Thus, a compiler can omit allocating space and generating code for the ghost variables. For this to work, values of ghost variables are not allowed to flow into non-ghost ("physical") variables, which is enforced by syntactic checks.

Like other variables, ghost variables are updated by assignment statements. For example, the following program snippet maintains the relation `g = 2*x`:

```
class C {
  var x: int; var y: int; ghost var g: int;
  method Update() modifies {this};
  { x := x + 1;  g := g + 2; }
}
```

(I will explain the **modifies** clause in Sec. 1.3.) Dafny follows the standard convention of object-oriented languages to let **this**.x be abbreviated as x, where **this** denotes the implicit receiver parameter of the method.

As far as the verifier is concerned, there is no difference between ghost variables and physical variables. In particular, their types and the way they undergo change are the same as for physical variables. At the cost of the explicit updates, this makes them much easier to deal with than model variables or pure methods (*e.g.*, [29]), whose values change as a consequence of other variables changing.

1.3 Modifications

An important part of a method specification is to say which pieces of the program state are allowed to be changed. This is called *framing* and is specified in Dafny by a **modifies** clause, like the one in the `Update` example above. For simplicity, all framing in Dafny is done at the object granularity, not the object-field granularity. So, a **modifies** clause indicates a set of *objects*, and that allows the method to modify *any field* of any of those objects.

For example, the `Update` method above is also allowed to modify **this**.y. If callers need to know that the method has no effect on y, the method specification can be strengthened by a postcondition **ensures** `y = old(y);`, where **old**(E), which can be used in postconditions, stands for the expression E evaluated on entry to the method.

A method's **modifies** clause must account for *all* possible updates of methods. This may seem unwieldy, since the set of objects a method affects can be both large and dynamically determined. There are various solutions to this problem; for example, Spec# makes use of a programmer-specified ownership hierarchy among objects [3,38], JML uses ownership and data groups [43,31], and separation logic and permission-based verifiers infer the frame from the given precondition [44,49,37]. Inspired by *dynamic frames* [27], Dafny uses the crude and simple **modifies** clauses just described, which allows the frame to be specified by

the value of a ghost variable. The standard idiom is to declare a set-valued ghost field, say `Repr`, to dynamically maintain `Repr` as the set of objects that are part of the receiver's representation, and to use `Repr` in **`modifies`** clauses (see [33]). For example:

```
class MyClass {
  ghost var Repr: set<object>;
  method SomeMethod() modifies Repr; { /* ... */ }
}
```

Recall, this **`modifies`** clause gives the method license to modify any field of any object in `Repr`. If **`this`** is a member of the set `Repr`, then the **`modifies`** clause also gives the method license to modify the field `Repr` itself.

In retrospect, I find that this language design—explicit, set-valued **`modifies`** clauses that specify modifications at the granularity of objects—has contributed greatly to the flexibility and simplicity of Dafny.

1.4 Functions

A class can declare mathematical functions. These are given a signature, which can include type parameters, a function specification, and a body. For illustration, consider the following prototypical function, declared in a class `C`:

```
function F(x: T) requires P; reads R; decreases D; { Body }
```

where `T` is some type, `P` is a boolean expression, `R` is a set-valued expression, `D` is a list of expressions, and `Body` is an expression (not a statement). The **`requires`** clause says in which states the function is allowed to be used; in other words, the function can be partial and its domain is defined by the **`requires`** clause. Just like Dafny checks for division-by-zero errors, it also checks that invocations of a function satisfy the function's precondition. The **`reads`** clause gives a frame for the function, saying which objects the function may depend on. Analogously to **`modifies`** clauses of methods, the **`reads`** clause describes a set of objects and the function is then allowed to depend on any field of any of those objects. Dafny enforces the **`reads`** clause in the function body. The **`decreases`** clause gives a termination metric (also known as a variant function or a ranking function), which specifies a well-founded order among recursive function invocations. Finally, `Body` defines the value of the function.

By default, function are "ghost" and can be used in specifications only. But by declaring the function with **`function method`**, the function definition is checked to be free of specification-only constructs and the function can then be used in compiled code.

The Dafny function is translated into a Boogie function with the name $C.F$ (in Boogie, dot is just another character that can be used in identifier names). In addition to the explicit Dafny parameters of the function, the Boogie function takes as parameters the heap and the receiver parameter. The prototypical function above gives rise to a Boogie axiom of the form:

$$(\,\forall\,\mathcal{H}\colon \mathit{HeapType},\ \mathit{this}\colon [\![\texttt{C}]\!],\ x\colon [\![\texttt{T}]\!]\ \bullet\\ \quad \mathit{GoodHeap}(\mathcal{H}) \wedge \mathit{this} \neq \mathit{null} \wedge [\![\texttt{P}]\!] \ \Rightarrow\ C.F(\mathcal{H}, \mathit{this}, x) = [\![\texttt{Body}]\!]\,)$$

where *HeapType* denotes the type of the heap, $[\![\,\cdot\,]\!]$ denotes the translation function from Dafny types and expressions into Boogie types and expressions, and *GoodHeap* holds of well-formed heaps (see [33]). The Dafny function declaration is allowed to omit the body, in which case this definitional axiom is omitted and the function remains uninterpreted.

When generating axioms, one needs to be concerned about the logical consistency of those axioms. Unless the user-supplied body is restricted, the definitional axiom could easily be inconsistent, in particular if the function is defined (mutually) recursively. To guard against this, Dafny insists that any recursion be well-founded, which is enforced in two phases. First, Dafny builds a call graph from the syntactic declarations of functions. Then, for any function that may be (mutually) recursive, the language makes use of the termination metric supplied by the **decreases** clause. Such a metric is a lexicographic tuple whose components can be expressions of any type. Dafny enforces that any call between mutually recursive functions leads to a strictly smaller metric value. In doing the comparison, it first truncates the caller's tuple and callee's tuple to the longest commonly typed prefix. Integers are ordered as usual, **false** is ordered below **true**, **null** is ordered below all other references, sets are ordered by subset, sequences are ordered by their length, and algebraic datatypes are ordered by their rank (see Sec. 1.9). All of these are finite and naturally bounded from below, except integers, for which a lower bound of `0` is enforced. The lower bound is checked of the caller's **decreases** clause, but the check is performed at the time of a (mutually) recursive call, not on entry to the caller. This makes the specification of some **decreases** clauses more natural.

An omitted **decreases** clause defaults to the set of objects denoted by the **reads** clause.

There is one more detail about the encoding of the definitional axiom. In Boogie, all declared axioms are available when discharging proof obligations. Thus, if the axioms are inconsistent, then all proof obligations can be discharged trivially, even the proof obligations designed to ensure the consistency of the axioms! To avoid such circularities, Dafny adds an antecedent to each definitional axiom; this lets the axiom be activated selectively. Let the *height* of a function be its order in a topological sort of all functions according to the call graph. For example, mutually recursive functions have the same height. The antecedent added to the definitional axiom of a function `F` of height h is $h < \mathit{ContextHeight}$, where *ContextHeight* is an uninterpreted constant. To activate the definitional axioms of non-recursive calls, the consistency check of function `F` is given the assumption $\mathit{ContextHeight} = h$. Proof obligations related to methods get to assume what amounts to $\mathit{ContextHeight} = \infty$, thus activating all definitional axioms.

To further explain the Boogie encoding of Dafny's functions, it is necessary first to give more details of the encoding of the heap [33]. The Dafny verifier models the heap as a map from object references and field names to values. The

type of a Dafny field-selection expression `o.f` depends on the type of the field `f`, which is modeled directly using Boogie's polymorphic type system. All object references in Dafny are modeled by a single Boogie type *Ref*. A field `f` of type `T` declared in a class `C` is modeled as a constant $C.f$ of type *Field* $[\![\mathtt{T}]\!]$, where *Field* is a unary type constructor. The type of the heap, *HeapType*, is a polymorphic map type $\langle\alpha\rangle[\mathit{Ref}, \mathit{Field}\ \alpha]\alpha$; in words, for any type α, given a *Ref* and a *Field* α, the map returns an α [40]. For example, if `f` is of type **`int`** and `o` has type `C`, then the Boogie encoding declares a constant $C.f$ of type *Field* **int** and, in a heap $\mathcal{H}$, `o.f` is modeled as $\mathcal{H}[o, C.f]$, which has the Boogie type **int**.

Back to the encoding of functions. The **`reads`** clause of the prototypical function above produces the following frame axiom:

$$
\begin{array}{l}
(\,\forall\,\mathcal{H},\mathcal{K}\colon \mathit{HeapType},\ \mathit{this}\colon [\![\mathtt{C}]\!],\ x\colon [\![\mathtt{T}]\!]\ \bullet \\
\quad \mathit{GoodHeap}(\mathcal{H}) \wedge \mathit{GoodHeap}(\mathcal{K}) \wedge \\
\quad (\,\forall\,\langle\alpha\rangle\ o\colon \mathit{Ref},\ f\colon \mathit{Field}\ \alpha\ \bullet\ o \neq \mathit{null} \wedge [\![\mathtt{o} \in \mathtt{R}]\!] \Rightarrow \mathcal{H}[o,f] = \mathcal{K}[o,f]\,) \\
\quad \Rightarrow C.F(\mathcal{H}, \mathit{this}, x) = C.F(\mathcal{K}, \mathit{this}, x)\,)
\end{array}
$$

where "$\forall\langle\alpha\rangle\ o\colon \mathit{Ref},\ f\colon \mathit{Field}\ \alpha$" quantifies over all types α, all references o, and all fields names f of type *Field* α. The frame axiom says that if two heaps $\mathcal{H}$ and $\mathcal{K}$ agree on the values of all fields of all non-null objects in `R`, then $C.F$ returns the same value in the two heaps.

At first, it may seem odd to have a frame axiom, since the function's definitional axiom is more precise, but the frame axiom serves several purposes. First, if `Body` is omitted, the frame axiom still gives a partial interpretation of the function. Second, the frame axiom opens the possibility of using scope rules where Dafny hides the exact definition, except in certain restricted scopes, for example when verifying the enclosing class. By emitting the definitional axiom only when verifying the program text in the restricted scopes, other scopes then only get to know what the function depends on, not its exact definition. Such scope rules are still under experimentation in the current version of Dafny (but see, *e.g.*, [30,49]). Third, for certain recursively defined functions, the frame axiom can sometimes keep the underlying SMT solver away from matching loops.

Dafny allows the frame of a function to be given as **`reads`** `*;`, in which case the function is allowed to depend on anything and the frame axiom is not emitted.

The logical consistency of the frame axiom plus the definitional axiom is justified by reads check that are part of the function body's consistency check: each heap dereference is checked to be in the declared **`reads`** clause [33]. Meanwhile, the frame axiom by itself is consistent, so it is always activated.

1.5 Specifying Data-Structure Invariants

When specifying a program built as a layers of modules, it is important to have specifications that let one abstract over the details of each module. Several approaches exist, for example built around ownership systems [13,12], explicit validity bits [3], separation logic [46], region logic [1], and permissions [9]. Dafny follows an approach inspired by dynamic frames [27], where the idea is to specify

frames as dynamically changing sets and where the consistency of data structures (that is, *object invariants* [42,3]) are specified by validity functions [19,39,44]. The sets, functions, ghost variables, and **modifies** clauses of Dafny are all that is needed to provide idiomatic support for this approach.

Let me illustrate the idiom that encodes dynamic frames in Dafny with this program skeleton:

```
class C {
  ghost var Repr: set<object>;
  function Valid(): bool
    reads {this} ∪ Repr;
  { this ∈ Repr ∧ ... }
  method Init()
    modifies {this};
    ensures Valid() ∧ fresh(Repr - {this});
  { ... Repr := {this} ∪ ...; }
  method Update()
    requires Valid();
    modifies Repr;
    ensures Valid() ∧ fresh(Repr - old(Repr));
  { ... }
  ...
}
```

The ghost variable `Repr` is the dynamic frame of the object's representation. The Dafny program needs to explicitly update the variable `Repr` when the object's representation changes. Dafny does not build in any notion of an object invariant. Instead, the body of function `Valid()` is used to define what it means for an object to be in a consistent state.

Dafny does not have any special constructs to support object construction. Instead, one declares a method, named `Init` in the skeleton above, that performs the initialization. A client then typically allocates and initializes an object as follows:

```
var c := new C;  call c.Init();
```

Method `Init` says it may modify the fields of the object being initialized, which includes the ghost field `Repr`. Its postcondition says the method will return in a state where `Valid()` is **`true`**. Since `Init` is allowed to modify `Repr`, it declares a postcondition that says something about how it changes `Repr`. An expression **`fresh`**`(S)`, which is allowed in postconditions, says that all non-null objects in the set `S` have been allocated since the invocation of the method. The postcondition of `Init` says that all objects it adds to `Repr`, except **`this`** itself, have been allocated by `Init`. Thus, the typical client above can conclude that `c.Repr` is disjoint from any previous set of objects in the program.

The program skeleton also shows a typical mutating method, `Update`. It requires and maintains the object invariant, `Valid()`, and it only modifies objects

in the object's representation. Since `Update` can modify the ghost variable `Repr`, the postcondition **`fresh`**`(...)` promises to add only newly allocated objects to `Repr`, which lets clients conclude that the object's representation does not bleed into previous object representations.

More about this idiom and some examples are found in the Marktoberdorf lecture notes [33].

Note that the specifications in the program skeleton above are just idiomatic. It is easy to deviate from this idiom. For example, to specify that the `Append` method of a `List` class will reuse the linked-list representation of the argument, one might use the following specification:

```
method Append(that: List)
  requires Valid() ∧ that.Valid();
  modifies Repr ∪ that.Repr;
  ensures Valid() ∧ fresh(Repr - old(Repr) - old(that.Repr));
```

Here, nothing is said about the final value of `that.Repr`; in particular, the caller cannot assume **`this`** and `that` to have disjoint representations after the call. Moreover, the value of `that.Valid()` is also under-specified on return, so the caller cannot assume `that` to still be in a consistent state. One may need to strengthen the precondition above with `Repr ∩ that.Repr = {}`, which says that the representations of **`this`** and `that` are disjoint, or perhaps with `Repr ∩ that.Repr ⊆ {null}`, which says that they share at most the **`null`** reference.

The code in the skeleton shows only the specifications one needs to talk about the object structure. To also specify functional correctness, one typically adds more ghost variables (for example,

```
ghost var Contents: seq<T>;
```

for a linked list of `T` objects) and uses these variables in method pre- and postconditions.

1.6 Type Parameters

Classes, methods, functions, algebraic datatypes, and datatype constructors are generic, that is, they can take type parameters. Here is an example generic class, where `T` denotes a type parameter of the class:

```
class Pair<T> {
  var a: T; var b: T;
  method Flip() modifies {this}; ensures a = old(b) ∧ b = old(a);
  { var tmp := a; a := b; b := tmp; }
}
```

Uses of generic features require some type instantiation, which can often be inferred. For example, this code fragment allocates an integer pair and invokes the `Flip` method:

```
var p := new Pair<int>; p.b := 7; call p.Flip(); assert p.a = 7;
```

Expressions whose type is a type parameter can only be treated parametrically, that is, Dafny does not provide any type cast or type query operation for such expressions. For example, the body of the `Flip` method cannot use `a` and `b` as integers, but the client code above can, since, there, the type parameter has been instantiated with **`int`**.

Dafny types are translated into Boogie types, which generally are coarser. For example, **`bool`** and **`int`** translate to the same types in Boogie, and all class types translate into one user-defined Boogie type *Ref*, which is used to model all references. Procedures and functions in Boogie can also take type parameters, but the Dafny verifier does not make use of them for Dafny generics. The reason for this primarily has to do with the types of field values in the heap. As explained above in Sec. 1.4, each Dafny field gives rise to one Boogie constant that denotes the field name. This is important, because it allows generic code to be verified just once; for example, the implementation of the `Flip` method is verified without considering any specific instantiation of type parameter `T`. But in contexts where a use of a field like `a` or `b` is known to produce a specific type, like in the client code for `Flip` above, retrieving that field from the heap needs to produce the specific type. That goes beyond what the type constructor *Field* can do. So, rather than using type parameters in Boogie to deal with the generics in Dafny, the Dafny verifier introduces one Boogie type *Box* to stand for all values whose type is a type parameter. It then also introduces conversion functions from each type to *Box* and vice versa. The verifier is careful not to box an already boxed value, which can be ensured by looking at the static types of expressions.

In my personal experience with the Spec# program verifier, I have found the encoding of generic types to be an error prone enterprise. In retrospect, I think the reason has been that boxed entities in Spec# are encoded as references, which is what they look like in the .NET virtual machine. Admittedly, Dafny's generics are simpler, but my feeling is that the decision of encoding generic types using a separate Boogie type has led to a more straightforward encoding.

1.7 Sets

Dafny supports finite sets. A set of `T`-valued elements has type **`set`**`<T>`. Operations on sets are the usual ones, like membership, union, difference, and subset, but not complement and not cardinality. Sets are encoded as maps from values to booleans. The axiomatization defines all operations in terms of set membership; for example, there are no axioms that directly state the distribution of union over intersection. This axiomatization seems to work smoothly in practice—it is fast, simple, and gets the verification done.

In that spirit, set equality is translated into a Boogie function *SetEqual*, which is defined by equality in membership. Because Boogie does not promise extensionality of its maps [32], the reverse does not necessarily hold. For example, if `F` is a function on sets and `s` and `t` are sets, the Dafny verifier may not succeed in

proving `F(s) = F(t)` even if it has enough information to establish that `s` and `t` have the same members. The verifier therefore includes the axiom

$$(\forall\, a, b: Set \bullet\ SetEqual(a, b) \Rightarrow a = b\ )$$

but because of the way quantifiers are handled in SMT solvers like Z3, this axiom is put into play only if the prover has a ground term $SetEqual(s, t)$. If that term is not available, the Dafny user may need to help the verifier along by supplying the statement **`assert s = t;`**, which both introduces the term $SetEqual(s, t)$ and adds it as a proof obligation.

A final remark about sets is that the Dafny encoding only ever uses sets of *boxes*. That is, the translation boxes values before adding them to sets. The reason for this is similar to the reason for introducing boxes for type parameters described in Sec. 1.6.

1.8 Sequences

Dafny also supports sequences, with operations like member selection, concatenation, and length. They are encoded analogously to sets, except that they do not use Boogie's built-in maps, but instead use a user-defined type *Seq* with a separate member-select function and a function for retrieving the length of the sequence. However, my experience with sequences has not been as smooth as with sets.

In particular, quantifying over sequence elements like in `(∀ i ● ...s[i]...)`, but where the index into the sequence involves not just the quantified variable `i` but also some arithmetic, does not always lead to useful quantifier triggering. Although this problem has more to do with mixing triggers and interpreted symbols (like +), the problem can become noticeable when specifying properties of sequences. The workaround, once one has a hunch that this is the problem, is either to rewrite the quantifier or to supply an assertion that mentions an appropriate term to be triggered. For example, the list reversal program in the Marktoberdorf lecture notes [33] needs an assertion of the form:

```
assert list[0] = data;
```

1.9 Algebraic Datatypes

An algebraic datatype defines a set of structural values. For example, generic nonempty binaries trees with data stored at the leaves can be defined as follows:

```
datatype Tree<T> { Leaf(T); Branch(Tree<T>, Tree<T>); }
```

This declaration defines two constructors for `Tree` values, `Leaf` and `Branch`. A use of a constructor is written like `#Tree.Leaf(5)`, an expression whose type is `Tree<int>`.

The most useful feature of a datatype is provided by the **`match`** expression, which indicates one case per constructor of the datatype. For example,

```
function LeafCount<T>(d: Tree<T>): int  decreases d;
{
  match d
  case Leaf(t) ⇒ 1
  case Branch(u,v) ⇒ LeafCount(u) + LeafCount(v)
}
```

is a function that returns the number of leaves of a given tree.

All datatypes are modeled using a single user-defined Boogie type, *Datatype*, and constructors are modeled as Boogie functions. There are five properties of interest in the axiomatization of such functions:

0. each constructor is injective in each of its arguments,
1. different constructors produce different values,
2. every datatype value is produced from some constructor of its type,
3. datatype values are (partially) ordered, and
4. the ordering is well-founded.

Dafny emits Boogie axioms for three of these properties.

Properties (0) and (1) are axiomatized in the usual way, by giving the inverse functions and providing a category code, respectively. Property (2) is currently not encoded by the Dafny verifier, because it can give rise to enormously expensive disjunctions. Luckily, the property is usually not needed, because the only case-split facility that Dafny provides on datatypes is the **match** expression and Dafny insists, through a simple syntactic check, that all cases are covered (which means there is usually no need to prove in the logic that all cases are handled).

Property (3) is encoded using an integer function *rank* and axioms that postulate datatype arguments of a constructor to have a smaller rank than the value constructed. For example, Dafny emits the following axiom for `Branch`:

$$(\forall\, a0, a1\colon Datatype \bullet\ rank(a0) < rank(Tree.Branch(a0, a1)))$$

Property (4) is of interest when one wants to do induction on the structure of datatypes. It holds if the datatypes in a program can be stratified so that every datatype includes some constructor all of whose datatype arguments come from a lower stratum. Dafny enforces such a stratification, but it does not actually emit an axiom for this property (which otherwise would simply have postulated *rank* to return non-negative integers only), because the SMT solver never sees any proof obligation that requires it.

If the underlying SMT solver provides native support for algebraic datatypes (which Z3 actually does, as does CVC-3 [6]), then Dafny could tap into that support (which presumably provides all five properties) instead of rolling its own axioms. However, that would also require the intermediate verification language to support algebraic datatypes (which Boogie currently does not).

One final, important thing remains to be said about the encoding of datatypes, and it concerns the definitional axioms generated for Dafny functions. Recursive functions are delicate to define in an SMT solver, because of the possibility

that axioms will be endlessly instantiated (a phenomenon known as a *matching loop* [18], see also [34]). The problem can be mitigated by specifying triggers that are structurally larger than the new terms produced by the instantiations. Following VeriFast [26], Dafny emits, for any function whose body is a **match** expression on one of the function's arguments, a series of definitional axioms, one corresponding to each **case**. The crucial point is that the trigger of each axiom discriminates according to the form of the function's argument used in the **match**.

For example, one of the definitional axioms for function `LeafCount` above is:

$$(\forall u, v\colon Datatype \bullet \\ LeafCount(Tree.Branch(u, v)) \;=\; LeafCount(u) + LeafCount(v)\ )$$

where the trigger is specified to be the left-hand side of the equality (for brevity, I omitted the $\mathcal{H}$ and *this* arguments to *LeafCount*). Note that the new *LeafCount* terms introduced by instantiations of this axiom would cause further instantiations only if the SMT solver has already equated u or v with some *Tree.Branch* term. In contrast, consider the following axiom, where I write $b0$ and $b1$ for the inverse functions of *Tree.Branch*:

$$(\forall d\colon Datatype \bullet \;\; LeafCount(d) \;=\; LeafCount(b0(d)) + LeafCount(b1(d))\ )$$

If triggered on the term $LeafCount(d)$, this axiom is likely to lead to a matching loop, since each instantiation gives rise to new terms that also match the trigger.

1.10 Termination Metrics

As a final language-feature topic, let me say more about termination, and in particular about the termination of loops. Loops can be declared with loop invariants and a termination metric, the latter being supplied with a **decreases** clause that takes a lexicographic tuple, just as for functions. Dafny verifies that, each time the loop's back edge is taken (that is, each time control reaches the end of the body of the **while** statement and proceeds to the top of a new iteration where it will evaluate the loop guard), the ("post-iteration") metric value is strictly smaller than the ("pre-iteration") metric value at the top of the current iteration.

As I mentioned in Sec. 1.4, each Dafny type has an ordering, has finite values only, and, except for integers, is bounded from below. If the decrement of the metric involves decreasing an integer-valued component of the lexicographic tuple, then Dafny checks, at the time of the back edge, that the pre-iteration value of that component had been at least `0`. The complicated form of this rule (compared to, say, the simpler rule of enforcing as a loop invariant that every integer-valued component of the metric is non-negative) gives more freedom in choosing the termination metric. For example, the following loop verifies with the simple **decreases** clause given, despite the fact that `n` will be negative after the loop:

```
while (0 ≤ n) decreases n; { ... n := n - 1; }
```

To support applications where it is not desirable to insist on loop termination, Dafny lets a loop be declared with **decreases** *;, which skips the termination check for the loop.

Dafny also supports **decreases** clauses for methods, giving protection against infinite recursion. As for functions, Dafny proceeds in two phases, in the first phase building a call graph and in the second phase checking the **decreases** clauses of (mutually) recursive calls. Use of the two phases reduces the number of **decreases** clauses that programs need to contain.

2 Case Study: Schorr-Waite Algorithm

The famous Schorr-Waite algorithm marks all nodes reachable in a graph from a given root node [47]. What makes the algorithm attractive, and challenging for verification, is that it keeps track of most of the state of its depth-first search by reversing edges in the graph itself. This can be appropriate in the marking phase of a garbage collector, which is run at a time when space is low. The functional correctness of the algorithm has four parts: (C0) all reachable nodes are marked, (C1) only reachable nodes are marked, (C2) there is no net effect on the structure of the graph, and (C3) the algorithm terminates.

In this section, I present the entire Dafny program text for the Schorr-Waite algorithm. Using this 120-line program as input, Dafny (using Boogie 2 and Z3 version 2.4) verifies its correctness in less than 5 seconds. A large number of proofs have been constructed for the algorithm through both pen-and-paper proofs and mechanical verifications (see, *e.g.*, [10, 0, 41, 25, 11]). The previous shortest mechanical-verifier input for this algorithm appears to be 400 lines of Isabelle proof scripts by Mehta and Nipkow [41], whereas many other attempts have been far longer. To my knowledge, no previous Schorr-Waite proof has been carried out solely by an SMT solver, and never before has all necessary specifications and loop invariants been presented in one conference paper.

Here is a description of highlights of the program:

Class `Node` (lines 0–5 in the program below) represents the nodes in the graph, each with some arbitrary out-degree as represented by field `children`. The Schorr-Waite algorithm adds the `childrenVisited` field for bookkeeping, and the ghost field `pathFromRoot` is used only for the verification.

Datatype `Path` (lines 7–10) represents lists of `Node`'s and is used by function `Reachable` (line 13) to describe the list of intermediate nodes between `from` and `to`. Function `ReachableVia` (line 18) is defined recursively according to that list of intermediate nodes. Reachability predicates are notoriously difficult for first-order SMT solvers, but the trigger-aware encoding of the definitional axiom (explained in Sec. 1.9) makes it work honorably for this program.

The method itself is defined starting at line 28 and its specification is given on lines 29–41. The preconditions say that the method parameters describe a proper graph with marks and auxiliary fields cleared. Correctness property (C0)

is specified by the postconditions on lines 36–37, (C1) on line 39, and (C2) on line 41. Noteworthy about this specification is that (C0) is specified as a closure property, using quantifiers, whereas (C1) uses the reachability predicate. Alternatively, one could have also specified (C0) with the reachability predicate, as the dual of (C1), but that would have led to a more complicated proof (using a loop invariant like Hubert and Marché's *I4c* in [25], which they describe as "the trickiest annotation"). Correctness property (C3) is implicit, since Dafny checks that loops and methods terminate.

The heart of the algorithm is implemented by a loop with 3 cases (lines 85–112), one for going deeper into the graph ("push"), one for considering the next child of the current node, and one for backtracking to the previous node on the stack ("pop"). The program maintains a ghost variable `stackNodes` that stores the visitation stack of the depth-first traversal. This ghost variable, which is updated explicitly (lines 48, 95, and 106), plays a vital part in the proof. It is used in most of the loop invariants. Lines 74–76 declare the relation between `stackNodes` and the Schorr-Waite reversed edges.

The loop invariant helps establish the postconditions as follows: Correctness property (C0) is maintained as a loop invariant (lines 51 and 61–62), except for those nodes that are on the stack. Ditto for (C2) (lines 63–65). Correctness property (C1) is also maintained as a loop invariant (line 72). It uses function `Reachable`, which is defined in terms of an existential quantifier. The prover needs help in establishing this existential quantifier when a node is marked (lines 46 and 109), so the program supplies a witness by using a local ghost variable `path` and an associated loop invariant (line 70). To maintain that invariant in the pop case, intermediate reachability paths are recorded in the ghost field `pathFromRoot`, see the loop invariant on line 71.

The termination metric for the loop is given as a lexicographic triple on line 83. The first component of the triple is a set, the second a sequence, and the third an integer. Dafny verifies that each iteration of the loop decreases this triple and that its integer component is bounded from below. Since sets are finite in Dafny, this establishes a well-founded order and thus implies that the loop terminates. In comparison, proving termination of the Schorr-Waite algorithm using Caduceus [21], a verifier equipped to use SMT solvers when possible, involved using a second-order formula even just to express the well-foundedness of the termination metric used, which necessitated the use of an interactive proof assistant to complete the proof [25].

```
0   class Node {
1     var children: seq<Node>;
2     var marked: bool;
3     var childrenVisited: int;
4     ghost var pathFromRoot: Path;
5   }
6
7   datatype Path {
8     Empty;
9     Extend(Path, Node);
10  }
11
12  class Main {
13    function Reachable(from: Node, to: Node, S: set<Node>): bool
```

```
14      requires null ∉ S;
15      reads S;
16    { (∃ via: Path • ReachableVia(from, via, to, S)) }
17
18    function ReachableVia(from: Node, via: Path, to: Node, S: set<Node>): bool
19      requires null ∉ S;
20      reads S;
21      decreases via;
22    {
23      match via
24      case Empty ⇒ from = to
25      case Extend(prefix, n) ⇒ n ∈ S ∧ to ∈ n.children ∧ ReachableVia(from, prefix, n, S)
26    }
27
28    method SchorrWaite(root: Node, ghost S: set<Node>)
29      requires root ∈ S;
30      // S is closed under 'children':
31      requires (∀ n • n ∈ S ⟹ n ≠ null ∧ (∀ ch • ch ∈ n.children ⟹ ch = null ∨ ch ∈ S));
32      // graph starts with nothing marked and nothing being indicated as currently being visited:
33      requires (∀ n • n ∈ S ⟹ ¬n.marked ∧ n.childrenVisited = 0);
34      modifies S;
35      // nodes reachable from 'root' are marked:
36      ensures root.marked;
37      ensures (∀ n • n ∈ S ∧ n.marked ⟹ (∀ ch • ch ∈ n.children ∧ ch ≠ null ⟹ ch.marked));
38      // every marked node was reachable from 'root' in the pre-state:
39      ensures (∀ n • n ∈ S ∧ n.marked ⟹ old(Reachable(root, n, S)));
40      // the structure of the graph has not changed:
41      ensures (∀ n • n ∈ S ⟹ n.childrenVisited = old(n.childrenVisited) ∧
                                n.children = old(n.children));
42    {
43      var t := root;
44      var p: Node := null;  // parent of t in original graph
45      ghost var path := #Path.Empty;
46      t.marked := true;
47      t.pathFromRoot := path;
48      ghost var stackNodes := [];
49      ghost var unmarkedNodes := S - {t};
50      while (true)
51        invariant root.marked ∧ t ≠ null ∧ t ∈ S ∧ t ∉ stackNodes;
52        invariant |stackNodes| = 0 ⟺ p = null;
53        invariant 0 < |stackNodes| ⟹ p = stackNodes[|stackNodes|-1];
54        // stackNodes has no duplicates:
55        invariant (∀ i, j • 0 ≤ i ∧ i < j ∧ j < |stackNodes| ⟹ stackNodes[i] ≠ stackNodes[j]);
56        invariant (∀ n • n ∈ stackNodes ⟹ n ∈ S);
57        invariant (∀ n • n ∈ stackNodes ∨ n = t ⟹
58          n.marked ∧ 0 ≤ n.childrenVisited ∧ n.childrenVisited ≤ |n.children| ∧
59          (∀ j • 0 ≤ j ∧ j < n.childrenVisited ⟹ n.children[j] = null ∨ n.children[j].marked));
60        invariant (∀ n • n ∈ stackNodes ⟹ n.childrenVisited < |n.children|);
61        invariant (∀ n • n ∈ S ∧ n.marked ∧ n ∉ stackNodes ∧ n ≠ t ⟹
62          (∀ ch • ch ∈ n.children ∧ ch ≠ null ⟹ ch.marked));
63        invariant (∀ n • n ∈ S ∧ n ∉ stackNodes ∧ n ≠ t ⟹
64          n.childrenVisited = old(n.childrenVisited));
65        invariant (∀ n • n ∈ S ⟹ n ∈ stackNodes ∨ n.children = old(n.children));
66        invariant (∀ n • n ∈ stackNodes ⟹
67          |n.children| = old(|n.children|) ∧
68          (∀ j • 0 ≤ j ∧ j < |n.children| ⟹
                  j = n.childrenVisited ∨ n.children[j] = old(n.children[j])));
69        // every marked node is reachable:
70        invariant old(ReachableVia(root, path, t, S));
71        invariant (∀ n, pth • n ∈ S ∧ n.marked ∧ pth = n.pathFromRoot ⟹
                                old(ReachableVia(root, pth, n, S)));
72        invariant (∀ n • n ∈ S ∧ n.marked ⟹ old(Reachable(root, n, S)));
73        // the current values of m.children[m.childrenVisited] for m's on the stack:
74        invariant 0 < |stackNodes| ⟹
                  stackNodes[0].children[stackNodes[0].childrenVisited] = null;
75        invariant (∀ k • 0 < k ∧ k < |stackNodes| ⟹
76          stackNodes[k].children[stackNodes[k].childrenVisited] = stackNodes[k-1]);
77        // the original values of m.children[m.childrenVisited] for m's on the stack:
78        invariant (∀ k • 0 ≤ k ∧ k+1 < |stackNodes| ⟹
```

```
79          old(stackNodes[k].children)[stackNodes[k].childrenVisited] = stackNodes[k+1]);
80        invariant 0 < |stackNodes| ⟹
81          old(stackNodes[|stackNodes|-1].children)[stackNodes[|stackNodes|-1].childrenVisited] =
            t;
82        invariant (∀ n ● n ∈ S ∧ ¬n.marked ⟹ n ∈ unmarkedNodes);
83        decreases unmarkedNodes, stackNodes, |t.children| - t.childrenVisited;
84      {
85        if (t.childrenVisited = |t.children|) {
86          // pop
87          t.childrenVisited := 0;
88          if (p = null) {
89            return;
90          }
91          var oldP := p.children[p.childrenVisited];
92          p.children := p.children[..p.childrenVisited] + [t] +
                          p.children[p.childrenVisited + 1..];
93          t := p;
94          p := oldP;
95          stackNodes := stackNodes[..|stackNodes| - 1];
96          t.childrenVisited := t.childrenVisited + 1;
97          path := t.pathFromRoot;
98        } else if (t.children[t.childrenVisited] = null ∨ t.children[t.childrenVisited].marked) {
99          // just advance to next child
100         t.childrenVisited := t.childrenVisited + 1;
101       } else {
102         // push
103         var newT := t.children[t.childrenVisited];
104         t.children := t.children[..t.childrenVisited] + [p] +
                          t.children[t.childrenVisited + 1..];
105         p := t;
106         stackNodes := stackNodes + [t];
107         path := #Path.Extend(path, t);
108         t := newT;
109         t.marked := true;
110         t.pathFromRoot := path;
111         unmarkedNodes := unmarkedNodes - {t};
112       }
113     }
114   }
115 }
116
```

Here is a breakdown of the effort involved in constructing this Dafny program. I spent 5 hours one night, writing the algorithm (starting from a standard depth-first traversal with an explicit stack) and specifications (C0) and (C2), along with the loop invariants necessary for verification. The next day, I implemented **`decreases`** clauses for loops in Dafny, which let me write the lexicographic triple on line 83 to prove (C3). I then spent 2–3 days trying to define `ReachableVia` using a **`seq`**`<Node>`, after which I gave up and hand-coded algebraic datatypes into the Dafny-generated Boogie program. That seemed to lead to a proof. After a many-month hiatus from Dafny, I then added datatypes to Dafny and, within a few more hours, completed the full proof.

In conclusion, while the specification of the algorithm is clear and reading any one line of the loop invariants is likely to receive nods from a programmer, the 32 lines of quantifier-filled loop invariants can be a mouthful. The hardest thing in writing the program is deciphering the verifier's error messages so that one can figure out what loop invariant to add or change. That task is not yet for non-experts. Although I am pleased to have done the proof, I find the loop invariants to be complicated because they are so concrete, and think I would prefer a refinement approach like that used by Abrial [0].

3 Related Work

To a large extent, the language, specification constructs, and logical encoding of Dafny borrow from other languages and verifiers. The particular combination adds up to a flexible and powerful system. In this section, I give a more detailed comparison with some of the most closely related tools.

The Java Modeling Language (JML) [29] is a rich specification language for Java. It has many of the same specification features as Dafny. The biggest difference with Dafny lies on the tool side, where JML lacks an automatic verifier. The KeY tool [7] accepts JML specifications, but the tool uses an interactive verifier. ESC/Java [22,15] uses JML specifications and uses an underlying SMT solver, but it performs *extended static checking* (that is, it intentionally misses some program errors in order to reduce the cost of using the tool [19]), not full verification.

In the specification language itself, JML supports behavioral subtyping for subclasses in Java. It has advanced support for ghosts, including model classes and model code. It does not build in algebraic datatypes, and termination metrics are more developed in Dafny. A larger difference is the way modular verification is done: JML uses object invariants and data groups [43,31] whereas Dafny uses dynamic frames.

Spec# [4,38] is an object-oriented language with specifications, defined as a superset of C# 2.0. It has an SMT-based automatic verifier [2] and provides modular verification by enforcing an ownership discipline among objects in the heap. For programs that fit this discipline, the necessary specifications are concise and natural. Spec# also has rich support for subclasses and immutable classes. However, Spec# lacks the mathematical specification constructs needed to carry out full functional correctness verification. For example, it has no ghost variables, no built-in sets or sequences, no algebraic datatypes, no termination metrics, and quantifiers are restricted to be ones that are executable. As a further comparison with Dafny, its support for verifying generic classes is not nearly as developed.

JML and Spec# use *pure methods* instead of mathematical functions. A pure method is a side-effect free method, written using statements in the programming language. The advantage of pure methods is that they leverage an existing language feature, and programs often contain query methods that return the value a mathematical function would have. However, pure methods are surprisingly complicated to get right. A major problem is that pure methods *do* have effects; for example, a pure method may allocate a hashtable that it uses during its computation. Another problem is that pure methods often are not deterministic, because they may return a newly allocated object (perhaps a non-interned string or an object representing a set of integers). These problems make it tricky to provide the programming logic with the desirable illusion that pure methods are functions (see, *e.g.*, [16,36]). In contrast, the treatment of mathematical functions in Dafny, and the logic functions in Caduceus [20] from which they were first inspired, is simple (as is the treatment of functions in other logics that only provide functions, not statements and other programming constructs). Dafny's **`function method`** declaration achieves the advantage of using

one language mechanism for both (restricted) specifications and code, but does so by letting the function be used in code rather than letting code be used as a function. I conclude with a slogan: pure methods are hard, functions are easy.

VeriCool 1 [50] has provided much inspiration for Dafny. It is also based on dynamic frames, but the prevailing style is to use VeriCool's pure methods (which are mostly like functions) instead of ghost variables. When such frames are defined recursively, they can bring about problems with matching loops in the SMT solver. Whereas Dafny does framing at the granularity of objects, VeriCool uses the more detailed object-field granularity. VeriCool has been extended to concurrency [48]. It does not support generic classes, algebraic datatypes, or termination metrics.

The idea in Dafny of using set-valued expressions for framing comes from the original work on dynamic frames by Kassios [27]. The difference lies in how the dynamic frames are represented, which impacts automation. The original dynamic frames are represented by specification variables, which are functions of other variables in the program. They are therefore more like Dafny's functions than Dafny's ghost variables. Because ghost variables are independent coordinates in the heap, they avoid the problems with recursively defined functions that, like in VeriCool 1, can be an issue for the SMT solver.

As for the notation, Kassios's preservation operator Ξ can be written using quantifiers and **`old`** in Dafny, the modification operator Δ corresponds to **`modifies`** clauses in Dafny, and Kassios's operator Λ for the *swinging pivots requirement* [39] corresponds to the idiom that uses **`fresh`** in Dafny (Sec. 1.5). Specification variables correspond to functions in Dafny, and the **frames** predicate corresponds to **`reads`** clauses in Dafny.

A recent trend seems to be to add more specification features to programming languages. For example, the Jahob verification system admits specifications written in higher-order logic [52]. This differs from interactive higher-order proof assistants in that the input mostly looks like a program, not a series of proof steps. Jahob also includes some features that can be used to write proofs, as does, to a lesser extent, Boogie [32]. VeriFast [26] integrates into C features for writing and proving lemmas.

Others are using SMT solvers for functional correctness verification. Régis-Gianas and Pottier used an SMT solver in their proof of Kaplar and Tarjan's algorithm for functional double-ended queues [45]. VCC [14] is being used to verify the Microsoft Hyper-V hypervisor. As part of that project, VCC has been used to prove the functional correctness (sans termination) of several challenging data structures.

From the other direction, various interactive proof assistants are using SMT solvers as part of their *grind* tactics.

4 Conclusions

In this paper, I have shown the design of Dafny, a language and verifier. Although it does not support all functional-correctness verification tasks—to do so

is likely to require more data types and perhaps some higher-order features—it has already demonstrated its use in automatic functional-correctness verification. Rosemary Monahan and I have also used Dafny to complete the 8 verification benchmarks proposed by Weide *et al.* [51], except for one aspect of one benchmark, which requires a form of lambda closure [35].

Dafny had started as an experiment to encode dynamic frames, but it has grown to become more of a general-purpose specification language and verifier (where modular verification is achieved via dynamic frames). As part of future work on Dafny, I intend to build a compiler that generates executable code.

I expect that more full functional correctness verifications will be done by SMT-based verifiers in the future.

Acknowledgments. I wish to thank Daan Leijen, Peter Müller, and Wolfram Schulte for encouraging me to write this paper. Michał Moskal provided helpful comments on a draft of this paper. Jochen Hoenicke, Andreas Podelski, and the participants at the IFIP WG 2.3 meeting in Lachen were helpful in discussing termination specifications. Finally, I thank the reviewers for their helpful comments.

References

0. Abrial, J.-R.: Event based sequential program development: Application to constructing a pointer program. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 51–74. Springer, Heidelberg (2003)
1. Banerjee, A., Naumann, D.A., Rosenberg, S.: Regional logic for local reasoning about global invariants. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 387–411. Springer, Heidelberg (2008)
2. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006)
3. Barnett, M., DeLine, R., Fähndrich, M., Leino, K.R.M., Schulte, W.: Verification of object-oriented programs with invariants. Journal of Object Technology 3(6), 27–56 (2004)
4. 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)
5. Barrett, C., Ranise, S., Stump, A., Tinelli, C.: The Satisfiability Modulo Theories Library, SMT-LIB (2008), www.SMT-LIB.org
6. Barrett, C., Tinelli, C.: CVC3. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 298–302. Springer, Heidelberg (2007)
7. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software: The KeY Approach. LNCS (LNAI), vol. 4334. Springer, Heidelberg (2007)
8. Bevier, W.R., Hunt, Jr., W.A., Moore, J.S., Young, W.D.: Special issue on system verification. Journal of Automated Reasoning 5(4), 409–530 (1989)
9. Boyland, J.: Checking interference with fractional permissions. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 55–72. Springer, Heidelberg (2003)

10. Broy, M., Pepper, P.: Combining algebraic and algorithmic reasoning: An approach to the Schorr-Waite algorithm. ACM TOPLAS 4(3), 362–381 (1982)
11. Bubel, R.: The Schorr-Waite-algorithm. In: [7], ch. 15
12. Clarke, D., Drossopoulou, S.: Ownership, encapsulation and the disjointness of type and effect. In: OOPSLA 2002, pp. 292–310. ACM, New York (2002)
13. Clarke, D., Potter, J., Noble, J.: Ownership types for flexible alias protection. In: OOPSLA 1998, pp. 48–64. ACM, New York (1998)
14. Cohen, E., Dahlweid, M., Hillebrand, M.A., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A practical system for verifying concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) Theorem Proving in Higher Order Logics. LNCS, vol. 5674, pp. 23–42. Springer, Heidelberg (2009)
15. Cok, D.R., Kiniry, J.R.: ESC/Java2: Uniting ESC/Java and JML. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 108–128. Springer, Heidelberg (2005)
16. Darvas, Á.P.: Reasoning About Data Abstraction in Contract Languages. PhD thesis, ETH Zurich, Diss. ETH No. 18622 (2009)
17. 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)
18. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM 52(3), 365–473 (2005)
19. Detlefs, D.L., Leino, K.R.M., Nelson, G., Saxe, J.B.: Extended static checking. Research Report 159, Compaq Systems Research Center (1998)
20. Filliâtre, J.-C., Marché, C.: Multi-prover verification of C programs. In: Davies, J., Schulte, W., Barnett, M. (eds.) ICFEM 2004. LNCS, vol. 3308, pp. 15–29. Springer, Heidelberg (2004)
21. 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)
22. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: PLDI 2002, pp. 234–245. ACM, New York (2002)
23. Gonthier, G.: Verifying the safety of a practical concurrent garbage collector. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 462–465. Springer, Heidelberg (1996)
24. Hoenicke, J., Leino, K.R.M., Podelski, A., Schäf, M., Wies, T.: It's doomed; we can prove it. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 338–353. Springer, Heidelberg (2009)
25. Hubert, T., Marché, C.: A case study of C source code verification: the Schorr-Waite algorithm. In: SEFM 2005, pp. 190–199. IEEE, Los Alamitos (2005)
26. Jacobs, B., Piessens, F.: The VeriFast program verifier. Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven (2008)
27. Kassios, I.T.: Dynamic frames: Support for framing, dependencies and sharing without restrictions. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 268–283. Springer, Heidelberg (2006)
28. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: formal verification of an OS kernel. In: SOSP 2009, pp. 207–220. ACM, New York (2009)

29. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: A behavioral interface specification language for Java. ACM SIGSOFT Software Engineering Notes 31(3), 1–38 (2006)
30. Leino, K.R.M.: Toward Reliable Modular Programs. PhD thesis, California Institute of Technology, Technical Report Caltech-CS-TR-95-03 (1995)
31. Leino, K.R.M.: Data groups: Specifying the modification of extended state. In: OOPSLA 1998, pp. 144–153. ACM, New York (1998)
32. Leino, K.R.M.: This is Boogie 2. Manuscript KRML 178 (2008), http://research.microsoft.com/~leino/papers.html
33. Leino, K.R.M.: Specification and verification of object-oriented software. In: Engineering Methods and Tools for Software Safety and Security. NATO Science for Peace and Security Series D: Information and Communication Security, vol. 22, pp. 231–266. IOS Press, Amsterdam (2009); Summer School Marktoberdorf 2008 lecture notes
34. Leino, K.R.M., Monahan, R.: Reasoning about comprehensions with first-order SMT solvers. In: Shin, S.Y., Ossowski, S. (eds.) SAC 2009. ACM, NewYork (2009)
35. Leino, K.R.M., Monahan, R.: Dafny meets the Verification Benchmarks Challenge. In: Leavens, G.T., O'Hearn, P., Rajamani, S.K. (eds.) VSTTE 2010. LNCS, vol. 6217, pp. 112–126. Springer, Heidelberg (2010)
36. Leino, K.R.M., Müller, P.: Verification of equivalent-results methods. In: Drossopoulou, S. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 307–321. Springer, Heidelberg (2008)
37. Leino, K.R.M., Müller, P.: A basis for verifying multi-threaded programs. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, Springer, Heidelberg (2009)
38. Leino, K.R.M., Müller, P.: Using the Spec# language, methodology, and tools to write bug-free programs. In: Müller, P. (ed.) Advanced Lectures on Software Engineering. LNCS, vol. 6029, pp. 91–139. Springer, Heidelberg (2010)
39. Leino, K.R.M., Nelson, G.: Data abstraction and information hiding. ACM TOPLAS 24(5), 491–553 (2002)
40. Leino, K.R.M., Rümmer, P.: A polymorphic intermediate verification language: Design and logical encoding. In: Esparza, J., Majumdar, R. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. LNCS, vol. 6015, pp. 312–327. Springer, Heidelberg (2010)
41. Mehta, F., Nipkow, T.: Proving pointer programs in higher-order logic. Information and Computation 199(1-2), 200–227 (2005); 19th International Conference on Automated Deduction (CADE-19)
42. Meyer, B.: Object-oriented Software Construction. Series in Computer Science.International. Prentice-Hall International, Englewood Cliffs (1988)
43. Müller, P., Poetzsch-Heffter, A., Leavens, G.T.: Modular invariants for layered object structures. Science of Computer Programming 62, 253–286 (2006)
44. Parkinson, M.J., Bierman, G.M.: Separation logic and abstraction. In: POPL 2005, pp. 247–258. ACM, New York (2005)
45. Régis-Gianas, Y., Pottier, F.: A Hoare logic for call-by-value functional programs. In: Audebaud, P., Paulin-Mohring, C. (eds.) MPC 2008. LNCS, vol. 5133, pp. 305–335. Springer, Heidelberg (2008)
46. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS 2002, pp. 55–74. IEEE, Los Alamitos (2002)
47. Schorr, H., Waite, W.M.: An efficient machine-independent procedure for garbage collection in various list structures. Commun. ACM 10(8), 501–506 (1967)

48. Smans, J., Jacobs, B., Piessens, F.: VeriCool: An automatic verifier for a concurrent object-oriented language. In: Barthe, G., de Boer, F.S. (eds.) FMOODS 2008. LNCS, vol. 5051, pp. 220–239. Springer, Heidelberg (2008)
49. Smans, J., Jacobs, B., Piessens, F.: Implicit dynamic frames: Combining dynamic frames and separation logic. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 148–172. Springer, Heidelberg (2009)
50. Smans, J., Jacobs, B., Piessens, F., Schulte, W.: Automatic verifier for Java-like programs based on dynamic frames. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 261–275. Springer, Heidelberg (2008)
51. Weide, B.W., Sitaraman, M., Harton, H.K., Adcock, B., Bucci, P., Bronish, D., Heym, W.D., Kirschenbaum, J., Frazier, D.: Incremental benchmarks for software verification tools and techniques. In: Shankar, N., Woodcock, J. (eds.) VSTTE 2008. LNCS, vol. 5295, pp. 84–98. Springer, Heidelberg (2008)
52. Zee, K., Kuncak, V., Rinard, M.C.: Full functional verification of linked data structures. In: PLDI 2008, pp. 349–361. ACM, New York (2008)

Relentful Strategic Reasoning in Alternating-Time Temporal Logic

Fabio Mogavero[1,2,⋆,⋆⋆], Aniello Murano[1,⋆⋆], and Moshe Y. Vardi[2,⋆⋆⋆]

[1] Universitá degli Studi di Napoli "Federico II", I-80126 Napoli, Italy
[2] Rice University, Department of Computer Science, Houston, TX 77251-1892, U.S.A.
{mogavero,murano}@na.infn.it, vardi@cs.rice.edu

Abstract. Temporal logics are a well investigated formalism for the specification, verification, and synthesis of reactive systems. Within this family, alternating temporal logic, ATL*, has been introduced as a useful generalization of classical linear- and branching-time temporal logics by allowing temporal operators to be indexed by coalitions of agents. Classically, temporal logics are memoryless: once a path in the computation tree is quantified at a given node, the computation that has led to that node is forgotten. Recently, mCTL* has been defined as a memoryful variant of CTL*, where path quantification is memoryful. In the context of multi-agent planning, memoryful quantification enables agents to "relent" and change their goals and strategies depending on their past history. In this paper, we define mATL*, a memoryful extension of ATL*, in which a formula is satisfied at a certain node of a path by taking into account both the future and the past. We study the expressive power of mATL*, its succinctness, as well as related decision problems. We also investigate the relationship between memoryful quantification and past modalities and show their equivalence. We show that both the memoryful and the past extensions come without any computational price; indeed, we prove that both the satisfiability and the model-checking problems are 2EXPTIME-COMPLETE, as they are for ATL*.

1 Introduction

Multi-agent systems recently emerged as a new paradigm for better understanding distributed systems [FHMV95, WW01]. In multi-agent systems, different processes can have different goals and the interactions between them may be adversarial or cooperative. Interactions between processes in multi-agent systems can thus be seen as games in the classical framework of game theory, with adversarial coalitions [OR94]. Classical branching-time temporal logics, such as CTL* [EH86], turn out to be of limited power when applied to multi-agent systems. For example, consider the property `Prop`: "processes 1 and 2 cooperate to ensure that a system (having more than two processes) never

⋆ Part of this research was done while the author was visiting the Rice University. Partially supported by ESF "Games for Design and Verification" project short visit grant n. 3339.
⋆⋆ Partially supported by MIUR PRIN Project n.2007-9E5KM8 and Vigevani Research Project Prize 2010.
⋆⋆⋆ Work supported in part by NSF grants CCF-0613889, ANI-0216467, and CCF-0728882, by BSF grant 9800096, and by gift from Intel.

E.M. Clarke and A. Voronkov (Eds.): LPAR-16, LNAI 6355, pp. 371–386, 2010.

enters a fail state". It is well known that CTL* cannot express Prop [AHK02]. Rather, CTL* can only say whether the set of all agents can or cannot prevent the system from entering a fail state.

In order to allow the temporal-logic framework to work within the setting of multi-agent systems, Alur, Henzinger, and Kupferman introduced *Alternating-Time Temporal Logic* (ATL*, for short) [AHK02]. This is a generalization of CTL* obtained by replacing the path quantifiers, "E" (*there exists*) and "A" (*for all*), with "*cooperation modalities*" of the form $\langle\langle A \rangle\rangle$ and $[\![A]\!]$, where A is a set of *agents*, which can be used to represent the power that a coalition of agents has to achieve certain results. In particular, these modalities express selective quantifications over those paths that can be effected as outcomes of infinite games between the coalition and its complement. ATL* formulas are interpreted over *game structures* (closely related to *systems* in [FHMV95]), which model a set of interacting processes. Given a game structure $\mathcal{G}$ and a set A of agents, the ATL* formula $\langle\langle A \rangle\rangle \psi$ is satisfied at a state s iff there is a *strategy* for the agents in A such that, no matter the strategy that is executed by agents not in A, the resulting outcome of the interaction satisfies ψ at s. Coming back to the previous example, one can see that the property Prop can be expressed by the ATL* formula $\langle\langle \{1,2\} \rangle\rangle \mathsf{G}\, \neg \mathit{fail}$, where G is the classical temporal modality "*globally*".

Traditionally, temporal logics are *memoryless*: once a path in the underlying structure (usually a computation tree) is quantified at a given state, the computation that led to that state is forgotten [KV06]. In the case of ATL*, we have even more: the logic is also "relentless", in the sense that the agents are not able to formulate their strategies depending on the history of the computation; when $\langle\langle A \rangle\rangle \psi$ is asserted in a state s, its truth is independent of the path that led to s. Inspired by a work on *strong cyclic planning* [DTV00], Pistore and Vardi proposed a logic that can express the spectrum between strong goal $\mathsf{A}\psi$ and the weak goal $\mathsf{E}\psi$ in planning [PV07]. A novel aspect of the Pistore-Vardi logic is that it is "*memoryful*", in the sense that the satisfiability of a formula at a state s depends on the future as well as on the past, i.e., the trace starting from the initial state and leading to s. Nevertheless, this logic does not have a standard temporal logical syntax (for example, it is not closed under conjunction and disjunction). Also, it is less expressive than CTL*. This has lead Kupferman and Vardi [KV06] to introduce a memoryful variant of CTL* (mCTL*, for short), which unifies in a common framework both CTL* and the Pistore-Vardi logic. Syntactically, mCTL* is obtained from CTL* by simply adding a special proposition *present*, which is needed to emulate the ability of CTL* to talk about the "present" time. Semantically, mCTL* is obtained from CTL* by reinterpreting the path quantifiers of the logic to be memoryful.

Recently, ATL* has become very popular in the context of multi-agent system planning [vdHWW02, Jam04]. In such a framework, a memoryful enhancement of ATL* enables "relentful" planning, that is, agents can relent and change their goals, depending on their history[1]. That is, when a specific goal at a certain state is checked, agents may learn from the past to change their goals. Note that this does not mean that agents change their strategy, but that they can choose a strategy that allows them to change their goals. For example, consider the ATL* formula $\langle\langle \emptyset \rangle\rangle \mathsf{G}\, \langle\langle A \rangle\rangle \psi$. In the memoryful

[1] In Middle English to relent means to melt. In modern English it is used only in the combination of "relentless".

framework, this formula is satisfied by a game structure $\mathcal{G}$ (at its starting node) iff for each possible trace (history) ρ the agents in A can ensure that the evolution of $\mathcal{G}$ that extends ρ satisfies ψ from the start state.

In this paper, we introduce and study the logic mATL*, a memoryful extension of ATL*. Thus, mATL* can be thought of as a fusion of mCTL* and ATL* in a common framework. Similarly to mCTL*, the syntax of mATL* is obtained from ATL* by simply adding a special proposition *present*. Semantically, mATL* is obtained from ATL* by reinterpreting the path quantifiers of the logic to be memoryful. More specifically, for a game structure $\mathcal{G}$, the mATL* formula $\langle\langle A\rangle\rangle\psi$ holds at a state s of $\mathcal{G}$ if there is a strategy for agents in A such that, no matter which is the strategy of the agents not in A, the resulting outcome of the game, obtained by *extending* the execution trace of the system ending in s, satisfies ψ. As an example for the usefulness of the relentful reasoning, consider the situation in which the agents in a set A have the goal to eventually satisfy q and, if they see r, they can also change their goal to eventually satisfy v. It is easy to formalize this property in ATL* with the formula $\langle\langle A\rangle\rangle(\mathsf{F}(q \vee r) \wedge \mathsf{G} f)$, where f is $r \rightarrow \langle\langle A\rangle\rangle(\mathsf{F} v)$. Consider, instead, the situation in which the agents in A have the goal to satisfy p until q holds, unless they see r in which case they change their goal to satisfy u until v holds from the *start* of the computation. This cannot be easily handled in ATL*, since the specification depends on the past. On the other hand, it can be handled in mATL*, with the formula $\langle\langle A\rangle\rangle((p \mathsf{U} (q \vee r)) \wedge \mathsf{G} f)$, where f is $r \rightarrow \langle\langle A\rangle\rangle(u \mathsf{U} v)$.

In the paper, we also consider an extension of mATL* with *past operators* (mpATL*, for short). As for classical temporal logics, past operators allow reasoning about the past in a computation [LPZ85]. In mpATL*, we can further require that coalitions of agents had a memoryful goal in the past. In more details, we can write a formula whose satisfaction, at a state s, depends on the trace starting from the initial state and leading to a state s' occurring before s. Coming back to the previous example, by using P as the dual of F, we can change the alternative goal f of agents in A to be $r \rightarrow \mathsf{P}(h \wedge \langle\langle A\rangle\rangle(u \mathsf{U} v))$, which requires that once r occurs at a state s, at a previous state s' of s in which h holds, the subformula u until v from the start of the computation must be true.

An important contribuition of this work is to show for the first time a clear and complete picture of the relationships among ATL* and its various extensions with memoriful quantification and past modalities, which goes beyond the expressiveness results obtained in [KV06] for mCTL*. Since memoryfulness refers to behavior from the start of the computation, which occurred in the past, memoryfulness is intimately connected to the past. Indeed, we prove this formally. We study the expressive power and the succinctness of mATL* w.r.t ATL*, as well as the memoryless fragment of mpATL* (i.e., the extension of ATL* with past modalities), which we call pATL*. We show that the three logics have the same expressive power, but both mATL* and pATL* are at least exponentially more succinct than ATL*. As for m$^-$ATL* (where the minus stands for the variant of the logic without the "present" proposition but the path interpretation is still memoryful), we prove that it is strictly less expressive than ATL*. On the other hand, we prove that pATL* is equivalent to p$^-$ATL*, but exponentially more succinct.

From an algorithmic point of view, we examine two decision problems for mpATL*, *model checking* and *satisfiability*. We show that model checking is not easier than satisfiability and in particular that both are 2EXPTIME-COMPLETE, as for ATL*. We

recall that this is not the case for mCTL*, where the model checking is EXPSPACE-COMPLETE, while satisfiability is 2EXPTIME-COMPLETE. For upper bounds, we follow an *automata-theoretic approach* [KVW00]. In order to develop a decision procedure for a logic with the *tree-model property*, one first develops an appropriate notion of tree automata and studies their emptiness problem. Then, the decision problem for the logic can be reduced to the emptiness problem of such automata. To this aim, we introduce a new automaton model, the *agent-action tree automata with satellites* (AGCTAS, for short), which extends both *automata over concurrent game structures* in [SF06] and *alternating automata with satellites* in [KV06], in a common setting. For technical convenience, AGCTAS states are partitioned into states regarding the satellite and those regarding the rest of the automaton, which we call the *main automaton*. The complexity results then come from the fact that mpATL* formulas can be translated into an AGCTAS with an exponential number of states for the main automaton and doubly exponential number of states for the satellite, and from the fact that the emptiness problem for AGCTAS is solvable in EXPTIME w.r.t. both the size of the main automaton and the logarithm of the size of the satellite.

As for mCTL*, the interesting properties shown for mATL* make this logic not only useful to its own, but also advantageous to efficiently decide other logics (once it is shown a tight reduction to it). In the case of mCTL*, we recall that this logic has been useful to decide the *embedded* CTL* *logic* (EmCTL*, for short), recently introduced in [NPP08]. EmCTL* allows to quantify over good and bad system executions. In [NPP08], the authors also introduce a new model checking methodology, which allows to group the system executions as good and bad, w.r.t the satisfiability of a base LTL specification. By using an EmCTL* specification, this model checking algorithm allows checking not only whether the base specification holds or fails to hold in a system, but also how it does so. In [NPP08], the authors use a polynomial translation of EmCTL* into mCTL* to solve efficiently decision problems related to EmCTL*. In the context of coalition logics, the use of an "embedded" framework seems even more interesting. In particular, an embedded ATL* logic (EmATL*, for short) could allow to quantify coalition of agents over good and bad system executions. Analogously to EmCTL*, one may show a polynomial translation from EmATL* to mATL* and use this result to efficiently solve decision problems concerning EmATL*. We postpone the details to the full version of this paper.

The outline of the paper follows. In Section 2, we recall the basic notions regarding concurrent game structures, strategies, plays, trees, and unwinding. In Section 3, we first introduce mATL* and define its syntax and semantics. Then, we introduce its extension mpATL* and study the expressiveness and succinctness of both mATL* and mpATL*. Finally, in Section 4, we introduce AGCTAS and show how to solve the satisfiability and model-checking problems for both mATL* and mpATL*.

2 Preliminaries

A *concurrent game structure* (CGS, for short) is a tuple $\mathcal{G} = \langle \mathrm{AP}, \mathrm{Ag}, \mathrm{Ac}, \mathrm{St}, \lambda, \tau, s_0 \rangle$, where AP and Ag are finite non-empty sets of *atomic propositions* and *agents*, Ac and St are enumerable non-empty sets of *actions* and *states*, $\lambda : \mathrm{St} \mapsto 2^{\mathrm{AP}}$ is a *labeling* function that maps each state s to the set of atomic propositions true in that state,

$\tau : \mathrm{St} \times \mathrm{Ac}^{\mathrm{Ag}} \mapsto \mathrm{St}$ is a *transition* function that maps a state and a *global decision* d (i.e., a function from Ag to Ac) to a state, and $s_0 \in \mathrm{St}$ is a designated *initial state*. By $|\mathcal{G}| = |\mathrm{St}| \cdot |\mathrm{Ac}|^{|\mathrm{Ag}|}$ we denote the *size* of the $\mathcal{G}$. If the set of actions is finite, i.e., $b = |\mathrm{Ac}| < \infty$, we say that $\mathcal{G}$ is *b-bounded* or simply *bounded*. If both the sets of actions and states are finite, we say that $\mathcal{G}$ is *finite*. It is easy to note that $\mathcal{G}$ is finite iff it has a finite size. For a set of agents A, a *decision* for A is $\mathsf{d}_A \in \mathrm{Ac}^A$ and a *counterdecision* for A is a decision $\mathsf{d}^c_A \in \mathrm{Ac}^{\mathrm{Ag}\setminus A}$ for agents not in A. By $\mathsf{d} = (\mathsf{d}_A, \mathsf{d}^c_A)$, we denote the *composition* of d_A and d^c_A.

A *trace* (resp., a *path*) is a finite (resp., an infinite) sequence of states $\rho \in \mathrm{St}^*$ (resp., $\pi \in \mathrm{St}^\omega$) such that, for all $0 \leq i < |\rho| - 1$ (resp., $i \in \mathbb{N}$), there exists a global decision d_i such that $\rho_{i+1} = \tau(\rho_i, \mathsf{d}_i)$ (resp., $\pi_{i+1} = \tau(\pi_i, \mathsf{d}_i)$). Intuitively, traces and paths are legal sequences of reachable states. A trace ρ is said *non-empty* iff $|\rho| > 0$ and *initial* iff $\rho_0 = s_0$, i.e., if ρ starts in the initial state. Moreover, with $\pi_{\leq i}$ we indicate the *prefix* up to the state of index i of the path π, i.e., the trace built by the first $i+1$ states $\pi_0, \cdots, \pi_i$. Finally, we use $\mathrm{Trc} \subseteq \mathrm{St}^*$ to indicate the sets of all the non-empty traces.

A *strategy* for a set of agents $A \subseteq \mathrm{Ag}$ is a partial function $\mathsf{f}_A : \mathrm{Trc} \rightharpoonup \mathrm{Ac}^A$ that maps a non-empty trace ρ to a decision $\mathsf{f}_A(\rho)$ of agents in A. A strategy f_A is called *memoryless* iff all its values depend only on the last state of the trace; otherwise, it is called *memoryful*. Formally, f_A is memoryless iff, for all traces ρ and states s with $\rho \cdot s$ belonging to the domain $\mathsf{dom}(\mathsf{f}_A)$ of f_A, it holds that $\mathsf{f}_A(\rho \cdot s) = \mathsf{f}_A(s)$. For a state s, we also say that f_A is *s-defined* iff it is defined on all the non-empty traces starting in s that are reachable through f_A. Formally, f_A is s-defined if $s \in \mathsf{dom}(\mathsf{f}_A)$ and for all traces $\rho \in \mathsf{dom}(\mathsf{f}_A)$, it holds that $\rho_0 = s$ and, for all counterdecisions d^c_A, it holds that $\rho \cdot \tau(\rho_{|\rho|-1}, (\mathsf{f}_A(\rho), \mathsf{d}^c_A)) \in \mathsf{dom}(\mathsf{f}_A)$. A path π is a *play* w.r.t. a π_0-defined strategy f_A of agents in A (f_A-*play*, for short), iff for all $i \in \mathbb{N}$, there is a counterdecision $\mathsf{d}^c_{A,i}$ such that $\pi_{i+1} = \tau(\pi_i, \mathsf{d}_i)$, where $\mathsf{d}_i = (\mathsf{f}_A(\pi_{\leq i}), \mathsf{d}^c_{A,i})$.

For a set Δ, a Δ-*tree* is a prefix-closed set $\mathrm{T} \subseteq \Delta^*$, i.e., if $x \cdot x' \in \mathrm{T}$, with $x' \in \Delta$, then also $x \in \mathrm{T}$. Elements of T are *nodes* and ε is its *root*. For every $x \in \mathrm{T}$ and $x' \in \Delta$, the node $x \cdot x' \in \mathrm{T}$ is a *successor* of x in T. T is *b-bounded* if the maximal number b of its node successors is finite. For a finite set Σ, a Σ-*labeled* Δ-*tree* is a pair $\langle \mathrm{T}, \mathsf{v} \rangle$, where T is a Δ-tree and $\mathsf{v} : \mathrm{T} \mapsto \Sigma$ is a *labeling* function. We drop Δ and Σ when they are clear from the context. For a node $x = y_0 \cdots y_k \in \mathrm{T}$, we denote by $\mathsf{trcto}(x)$ and $\mathsf{wrdto}(x)$, respectively, the trace $(\varepsilon) \cdot (y_0) \cdots (y_0 \cdots y_k) \in \mathrm{T}^*$, and the word $\mathsf{v}(\varepsilon) \cdot \mathsf{v}(y_0) \cdots \mathsf{v}(y_0 \cdots y_k) \in \Sigma^*$. Finally, a Σ-*labeled agent-action tree* (AAT, for short) is a tuple $\mathcal{T} = \langle \mathrm{Ag}, \mathrm{Ac}, \mathrm{T}, \mathsf{v} \rangle$, where Ag and Ac are as in CGSs and $\langle \mathrm{T}, \mathsf{v} \rangle$ is a Σ-labeled $\mathrm{Ac}^{\mathrm{Ag}}$-tree.

A CGS $\mathcal{U} = \langle \mathrm{AP}, \mathrm{Ag}, \mathrm{Ac}, \mathrm{St}, \lambda, \tau, s_0 \rangle$, where St is an $\mathrm{Ac}^{\mathrm{Ag}}$-tree, $s_0 = \varepsilon$, and $\tau(s, \mathsf{d}) = s \cdot \mathsf{d}$, is called *concurrent game tree* (CGT, for short). With each CGT $\mathcal{U}$ we can associate a 2^{AP}-labeled AAT $\mathcal{T} = \langle \mathrm{Ag}, \mathrm{Ac}, \mathrm{T}, \mathsf{v} \rangle$, in which $\mathrm{T} = \mathrm{St}$ and $\mathsf{v}(x) = \lambda(x)$, for all nodes x. Note that a b-bounded CGT has as set of states a $b^{|\mathrm{Ag}|}$-bounded tree. Given a CGS $\mathcal{G} = \langle \mathrm{AP}, \mathrm{Ag}, \mathrm{Ac}, \mathrm{St}, \lambda, \tau, s_0 \rangle$, the *unwinding* $\mathcal{U}_\mathcal{G}$ of $\mathcal{G}$ is the CGT $\langle \mathrm{AP}, \mathrm{Ag}, \mathrm{Ac}, \mathrm{St}', \lambda', \tau', \varepsilon \rangle$ for which there is a surjective function $\mathsf{unw} : \mathrm{St}' \mapsto \mathrm{St}$ such that $\mathsf{unw}(\varepsilon) = s_0$ and, for all nodes x and decisions d, we have $\mathsf{unw}(x \cdot \mathsf{d}) = \tau(\mathsf{unw}(x), \mathsf{d})$ and $\lambda'(x) = \lambda(\mathsf{unw}(x))$. Note that each CGS $\mathcal{G}$ has a unique associated unwinding $\mathcal{U}_\mathcal{G}$ and so a unique AAT $\mathcal{T}_\mathcal{G}$. Finally, all above definitions of trace, path, strategy, and play easily extend to AAT.

3 Memoryful Alternating-Time Temporal Logic

In this section, we introduce the *memoryful alternating-time temporal logic* (mATL*, for short), obtained by allowing the alternating-time temporal logic ATL* [AHK02] to use memoryful quantification over paths, in a similar way it has been done for the memoryful branching-time temporal logic mCTL* [KV06]. mATL* inherits from ATL* the existential $\langle\!\langle A \rangle\!\rangle$ and the universal $[\![A]\!]$ *strategy(-play) quantifiers*, where A denotes a set of agents. We recall that these two quantifiers can be read as *"there exists a collective strategy for agents in A"* and *"for all collective strategies for agents in A"*, respectively. The syntax of mATL* is similar to that for ATL*: there are *state formulas* and *path formulas*. Strategy quantifiers can prefix an assertion composed of an arbitrary Boolean combination and nesting of the linear-time operators X (*"next"*), U (*"until"*), and R (*"release"*). The only syntactical difference between the two logics is that mATL* formulas can refer to a special atomic proposition *present*, which enables us to refer to the present. Readers familiar with mCTL* can see mATL* as mCTL* where strategy quantifiers substitute path quantifiers. The formal syntax of mATL* follows.

Definition 1. *Let* AP *and* Ag *be the sets of atomic propositions and agents.* mATL* state *(φ) and* path *(ψ)* formulas *are built inductively by the following context-free grammar, with $p \in \mathrm{AP}$ and $A \subseteq \mathrm{Ag}$:*

1. $\varphi ::= \mathrm{present} \mid p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \langle\!\langle A \rangle\!\rangle \psi \mid [\![A]\!] \psi$;
2. $\psi ::= \varphi \mid \neg\psi \mid \psi \wedge \psi \mid \psi \vee \psi \mid \mathsf{X}\psi \mid \psi \,\mathsf{U}\, \psi \mid \psi \,\mathsf{R}\, \psi$.

The class of mATL* *formulas is the set of all the state formulas generated by the above grammar, in which the occurrences of the special atomic proposition* present *is in the scope of a strategy quantifier.*

The *length* $|\varphi|$ of a formula φ is defined inductively on the structure of φ itself, in the classical way, by also considering $|\langle\!\langle A \rangle\!\rangle \varphi|$ and $|[\![A]\!]\varphi|$ to be equal to $1 + |A| + |\varphi|$.

As for ATL*, the semantics of mATL* is defined w.r.t. a concurrent game structure. However, the two logics differ on interpreting state formulas. First, in mATL* the satisfaction of a state formula is related to a specific trace, while in ATL* it is related only to a state. Moreover, path quantification in mATL* ranges over paths that start at the initial state and contain as prefix the trace that lead to the present state; we refer to this trace as the *present trace*. This is what we refer to as *memoryful quantification*. In contrast, in ATL* path quantification ranges over paths that start at the present state. For example, consider the formula $\varphi = [\![A]\!]\mathsf{G}\langle\!\langle B \rangle\!\rangle \psi$. Considered as an ATL* formula, φ holds in the initial state of a structure if the agents in B can force a path satisfying ψ from every state that can be reached by a strategy of the agents in A. In contrast, considered as an mATL* formula, φ holds in the initial state of the structure if the agents in B can extend to a path satisfying ψ every trace generated by a strategy of the agent in A. Thus, when evaluating path formulas in mATL* one cannot ignore the past, and satisfaction may depend on the event that preceded the point of quantification. In ATL*, state formulas are evaluated w.r.t. states in the structure and path formulas are evaluated w.r.t. paths in the structure. In mATL* we add an additional parameter, the *present trace*, which is the trace that led from the initial state to the point of quantification. Path formulas are

again evaluated w.r.t. paths, but state formulas are now evaluated w.r.t. traces, which are viewed as partial executions. We now formally define mATL* semantics w.r.t. a CGS $\mathcal{G}$.

For two non-empty initial traces ρ and ρ_p, where ρ_p is the present trace, we write $\mathcal{G}, \rho, \rho_p \models \varphi$ to indicate that the state formula φ holds at ρ, with ρ_p being the present. Similarly, for a path π, a non-empty present trace ρ_p and a natural number k, we write $\mathcal{G}, \pi, k, \rho_p \models \psi$ to indicate that the path formula ψ holds at the position k of π, with ρ_p being the present. The semantics of the mATL* state formulas involving $\neg$, $\wedge$, and $\vee$, as well as that for mATL* path formulas, except for the state formula case, is defined as usual in CTL* (see Appendix A, for a full definition). The semantics of the remaining part, which involves the memoryful feature, follows:

Definition 2. *Given a* CGS $\mathcal{G} = \langle \mathrm{AP}, \mathrm{Ag}, \mathrm{Ac}, \mathrm{St}, \lambda, \tau, s_0 \rangle$*, two initial traces* $\rho, \rho_p \in \mathrm{Trc}$*, a path* π*, and a number* $k \in \mathbb{N}$*, where* $\rho = \rho' \cdot s$*,* $\rho' \in \mathrm{Trc} \cup \{\varepsilon\}$*, and* $s \in \mathrm{St}$*, it holds that:*

1. $\mathcal{G}, \rho, \rho_p \models \mathrm{present}$ *iff* $\rho = \rho_p$*;*
2. $\mathcal{G}, \rho, \rho_p \models p$*, for* $p \in \mathrm{AP}$*, iff* $p \in \lambda(s)$*;*
3. $\mathcal{G}, \rho, \rho_p \models \langle\!\langle A \rangle\!\rangle \psi$ *iff there exists an s-defined strategy* f_A *of agents in A such that for all* f_A*-plays* π *it holds that* $\mathcal{G}, \rho' \cdot \pi, 0, \rho \models \psi$*;*
4. $\mathcal{G}, \rho, \rho_p \models [\![A]\!] \psi$ *iff for all the s-defined strategies* f_A *of agents in A there exists an* f_A*-play* π *such that* $\mathcal{G}, \rho' \cdot \pi, 0, \rho \models \psi$*;*
5. $\mathcal{G}, \pi, k, \rho_p \models \varphi$ *iff* $\mathcal{G}, \pi_{\leq k}, \rho_p \models \varphi$*.*

Note that the present trace ρ_p comes into the above definition only at item 1 and that formulas of the form $\langle\!\langle A \rangle\!\rangle \psi$ and $[\![A]\!] \psi$ "reset the present", i.e., their satisfaction w.r.t ρ and ρ_p is independent of ρ_p, and the present trace, for the path formula ψ, is set to ρ.

We say that a CGS $\mathcal{G}$ is a *model* of an mATL* formula φ, denoting this by $\mathcal{G} \models \varphi$, iff $\mathcal{G}, s_0, s_0 \models \varphi$. Moreover, φ is said *satisfiable* iff there exists a model $\mathcal{G}$ for it. For two mATL* formulas φ_1 and φ_2 we say that φ_1 is *equivalent* to φ_2, formally $\varphi_1 \equiv \varphi_2$, iff, for all CGSs $\mathcal{G}$, and non-empty traces ρ and ρ_p, it holds that $\mathcal{G}, \rho, \rho_p \models \varphi_1$ iff $\mathcal{G}, \rho, \rho_p \models \varphi_2$.

By induction on the syntactical structure of the sentences, it is possible to prove the following classical result. Note that this is a basic step towards the automata-theoretic approach we use to solve the model-checking and the satisfiability problems for mATL*.

Theorem 1. mATL* *satisfies the tree model property. In fact, for each* CGS $\mathcal{G}$ *and formula* φ*, it holds that* $\mathcal{G} \models \varphi$ *iff* $\mathcal{U}_\mathcal{G} \models \varphi$*.*

From this result and the one-to-one connection between the CGT $\mathcal{U}_\mathcal{G}$ (obtained as the unwinding of the CGS $\mathcal{G}$) and the related AAT $\mathcal{T}_\mathcal{G}$, we say that $\mathcal{T}_\mathcal{G}$ satisfies φ iff $\mathcal{G} \models \varphi$.

When we compare two logics, the basic comparison is in terms of *expressiveness*. A logic L_1 is *as expressive* as a logic L_2 iff every formula in L_2 is logically equivalent to some formula in L_1. If L_1 is as expressive as L_2, but there is a formula in L_1 that is not logically equivalent to any formula in L_2, then L_1 is *more expressive* than L_2. If L_1 is as expressive as L_2 and vice versa, then L_1 and L_2 are *expressively equivalent*. We can compare the logics L_1 and L_2 also in terms of *succinctness*, which measures the necessary blow-up when translating between the logics. Note that comparing logics in terms of succinctness makes sense, when the logics are not expressively equivalent, focusing then on their common fragment. In fact, a logic L_1 can be more expressive than a logic L_2, but at the same time, less succinct than the latter.

We now discuss expressiveness and succinctness of mATL* w.r.t. ATL* as well as some extensions/restrictions of mATL*. In particular we consider the logics mpATL* and pATL* to be, respectively, mATL* and ATL* augmented with the past-time operators *"previous"* and *"since"*, which dualize the future-time operators *"next"* and *"until"* as in pLTL [LPZ85] and pCTL* [KP95] (see Appendix A for more). Note that pATL* still contains the present proposition and that, as for pCTL*, the semantics of its quantifiers is as for ATL*, where the past is considered linear, i.e., deterministic. Moreover, we consider the logic m^{-}ATL*, p^{-}ATL*, and mp^{-}ATL* to be, respectively, the syntactical restriction of mATL*, pATL*, and mpATL* in which the use of the atomic proposition *present* is not allowed. On one hand, we have that all mentioned logics are expressively equivalent, except for m^{-}ATL* and p^{-}ATL*. On the other hand, the ability to refer to the past makes all of them at least exponentially more succinct than the corresponding ones without the past. For example, a pATL* formula φ can be translated into an equivalent ATL* one φ', but φ' may require a nonelementary space in $|\varphi|$ (shortly, we say that pATL* is nonelementary reducible to ATL*). Note that, to get a better complexity for this translation is not an easy question. Indeed, it would improve the non-elementary reduction from *first order logic* to LTL, which is an outstanding open problem [Gab87]. All the discussed results are reported in the following theorem.

Theorem 2. *The following properties hold:*

1. ATL* *(resp.,* pATL*) is linearly reducible to* mATL* *(resp.,* mpATL*);*
2. mpATL* *(resp.,* mp^{-}ATL*) is linearly reducible to* pATL* *(resp.,* p^{-}ATL*);*
3. mpATL* *(resp.,* mp^{-}ATL*) is nonelementarily reducible to* mATL* *(resp.,* m^{-}ATL*);*
4. pATL* *is nonelementarily reducible to* ATL*;*
5. m^{-}ATL* *and* p^{-}ATL* *are at least exponentially more succinct than* ATL*;*
6. m^{-}ATL* *is less expressive then* ATL*.*

Proof (Sketch). Let φ be an input formula for items 1-4. Items 1 and 2 follow by replacing each subformula $\langle\!\langle A\rangle\!\rangle \psi$ in φ by $\langle\!\langle A\rangle\!\rangle \mathsf{F}\,(\mathit{present} \wedge \psi)$ and $\langle\!\langle A\rangle\!\rangle \mathsf{P}\,((\widetilde{\mathsf{Y}}\,\mathfrak{false}) \wedge \psi)$, respectively, where $\mathsf{P}\,\psi'$ is the corresponding past-time operator for $\mathsf{F}\,\psi'$ and $\widetilde{\mathsf{Y}}\,\psi'$ is the hypothetical previous time operator, which is true if either ψ' is true in the previous time-step or such a time-step does not exist. Item 3 follows by replacing each subformula $\langle\!\langle A\rangle\!\rangle \psi$ in φ by $\langle\!\langle A\rangle\!\rangle \psi'$, where ψ' is obtained by the Separation Theorem (see Theorem 2.4 of [Gab87]), which allows to eliminate all pure-past formulas[2]. Note that all the above substitutions start from the innermost subformula. Item 4 proceeds as for the translation of pCTL* into CTL (see Lemma 3.3 and Theorem 3.4 of [KP95]). The only difference here is that, when we apply the Separation Theorem to obtain a path formula as a disjunction of formulas of the form $ps \wedge pr \wedge ft$, where ps, pr, ft are respectively pure-past, pure-present and pure-future formulas, we need to substitute *present* by $\mathfrak{false}$ in ps and ft and by $\mathfrak{true}$ in pr. For items 3 and 4 the non-elementary blow-up is inherited from the use of the Separation Theorem. Item 5 follows by using the formula $\varphi = \langle\!\langle A\rangle\!\rangle \mathsf{G}\,(\bigwedge_{i=1}^{n}(p_i \Leftrightarrow [\![\emptyset]\!] p_i) \Rightarrow (p_0 \Leftrightarrow [\![\emptyset]\!] p_0))$ (resp., $\varphi = \langle\!\langle A\rangle\!\rangle \mathsf{G}\,(\bigwedge_{i=1}^{n}(p_i \Leftrightarrow$

[2] A pure-past formula contains only past-time operators. In item 4, we also consider pure-future formulas, which contain only future-time operators, and pure-present formulas, which do not contain any temporal operator at all.

$\mathsf{P}((\widetilde{\mathsf{Y}}\,\mathfrak{false}) \wedge p_i)) \Rightarrow (p_0 \Leftrightarrow \mathsf{P}((\widetilde{\mathsf{Y}}\,\mathfrak{false}) \wedge p_0))))$, which is similar to that used to prove that pLTL is exponentially more succinct than LTL (see Theorem 3.1 of [LMS02]). By using an argument similar to that used in [LMS02], we obtain the desired result. Item 6 follows by using a proof similar to that used for m$^-$CTL* (see Theorem 3.4 of [KV06]), and so showing that the ATL formula $\varphi = \langle\!\langle A \rangle\!\rangle \mathsf{F}((\llbracket \emptyset \rrbracket \mathsf{X}\, p) \wedge (\llbracket \emptyset \rrbracket \mathsf{X} \neg p))$ has no m$^-$ATL* equivalent formula.

As an immediate consequence of combinations of the results shown into the previous theorem, it is easy to prove the following corollary.

Corollary 1. mATL*, p$^-$ATL*, pATL*, *and* mpATL* *have the same expressive power of* ATL*. m$^-$ATL* *and* mp$^-$ATL* *have the same expressive power, but are less expressive than* ATL*. *Moreover, all of them are at least exponentially more succinct than* ATL*.

Fig. 1 summarizes all the above results regarding expressiveness and succinctness. The acronym *"lin"* (resp., *"ne"*) means that the translation exists and it is linear (resp., nonelementarily) in the size of the formula, and *"/"* means that such a translation is impossible. The numbers in brackets represent the item of Theorem 2 in which the translation is shown. We use no numbers when the translation is trivial or comes by a composition of existing ones.

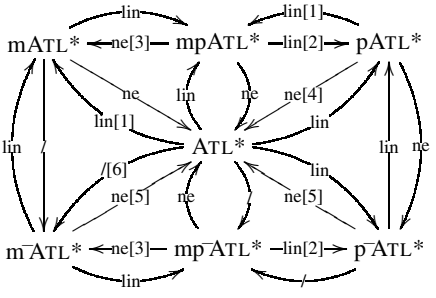


Fig. 1. Hierarchy of expressive power and succinctness

4 Decision Procedures

In this section, we study the satisfiability and model-checking problems for mpATL*. We directly study the richer mpATL* logic, since we prove the 2EXPTIME upper bound for this logic. To obtain such upper bounds, we use an automata-theoretic approach by introducing a novel automaton model: *agent-action tree automata with satellites*.

4.1 Agent-Action Tree Automata with Satellites

Alternating tree automata [MS87] are a generalization of nondeterministic tree automata. Intuitively, while a nondeterministic automaton that visits a node of the input tree sends exactly one copy of itself to each of the successors of the node, an alternating automaton can send several copies of itself to the same successor. *Symmetric automata* [JW95] are a variation of classical (asymmetric) alternating automata in which it is not necessary to specify the direction (i.e., the choice of the successors) of the tree on which a copy is sent. In fact, through two generalized directions (existential and universal moves), it is possible to send a copy of the automaton, starting from a node of the input tree, to some of its successors or to all its successors. Hence, the automaton does not distinguish between directions. As a generalization of alternating automata (both in the symmetric and asymmetric cases), here we consider *agent-action tree automata* (AGCTA, for short), which can send copies to successor nodes, according to

agents' decisions. These automata are a slight variation of *automata over concurrent game structures*, which were introduced in [SF06]. Moreover, we also consider AGCTA along with the satellite framework (AGCTAS, for short), in a similar way it has been done in [KV06]. The satellite is used to take a bounded memory of the evaluated part of a path in a given structure and it is kept apart from the main automaton as it allows to show easily a tight complexity of the considered problems w.r.t. the size of the specification. We use symmetric AGCTAS for the satisfiability and asymmetric AGCTAS for the model-checking. In the following, we simply write AGCTA when we indifferently refer to its symmetric or asymmetric version. The formal definitions of AGCTA and AGCTAS follow.

Definition 3. *A* symmetric AGCTA *is a tuple* $\mathcal{A} = \langle \Sigma, \mathrm{Ag}, \mathrm{Q}, \delta, q_0, \mathrm{F} \rangle$*, where* Σ*,* Ag*, and* Q *are non-empty finite sets of* input symbols*,* agents*, and* states*, respectively,* $q_0 \in \mathrm{Q}$ *is an* initial state*,* F *is an* acceptance condition *to be defined later, and* $\delta : \mathrm{Q} \times \Sigma \mapsto \mathtt{B}^+(\mathrm{D} \times \mathrm{Q})$ *is an* alternating transition function*, where* $\mathrm{D} = \{\Diamond, \Box\} \times 2^{\mathrm{Ag}}$ *is an extended set of abstract directions, which maps a state and an input symbol to a positive boolean combination of two kinds of* atoms*:* existential atoms $((\Diamond, A), q')$ *and* universal atoms $((\Box, A), q')$*. Moreover,* $\mathcal{A}$ *is* asymmetric *if it also contains a set* Ac *of actions (i.e.,* $\mathcal{A} = \langle \Sigma, \mathrm{Ag}, \mathrm{Ac}, \mathrm{Q}, \delta, q_0, \mathrm{F} \rangle$*) and* $\delta : \mathrm{Q} \times \Sigma \mapsto \mathtt{B}^+(\mathrm{Ac}^{\mathrm{Ag}} \times \mathrm{Q})$ *contains atoms of the form* (d, q')*, where* d *is a decision of the agents in* Ag*.*

Definition 4. *A* run *of a symmetric* AGCTA $\mathcal{A}$ *on a* Σ*-labeled* AAT $\mathcal{T} = \langle \mathrm{Ag}, \mathrm{Ac}, \mathrm{T}, \mathsf{v} \rangle$ *is a* $(\mathrm{Q} \times \mathrm{T})$*-labeled* $\mathbb{N}$*-tree* $\mathcal{R} = \langle \mathrm{Tr}, \mathsf{r} \rangle$ *such that* (i) $\mathsf{r}(\varepsilon) = (q_0, \varepsilon)$ *and* (ii) *for all* $y \in \mathrm{Tr}$*, with* $\mathsf{r}(y) = (q, x)$*, there is a set* $\mathrm{S} \subseteq \mathrm{D} \times \mathrm{Q}$*, with* $\mathrm{S} \models \delta(q, \mathsf{v}(x))$*, such that for all atoms* $(a, q') \in \mathrm{S}$ *it holds that*

- *if* $a = (\Diamond, A)$ *then there exists a decision* $\mathsf{d}_A \in \mathrm{Ac}^A$ *such that for all counterdecisions* $\mathsf{d}_A^c \in \mathrm{Ac}^{\mathrm{Ag} \setminus A}$ *it holds that* $(q', x \cdot (\mathsf{d}_A, \mathsf{d}_A^c)) \in \mathsf{L}(y)$*, where* $\mathsf{L}(y)$ *is the set* $\{\mathsf{r}(y \cdot y') \mid y' \in \mathbb{N}, y \cdot y' \in \mathrm{Tr}\}$ *of labels of successors of* y *in* $\mathcal{R}$*;*
- *if* $a = (\Box, A)$ *then for all decisions* $\mathsf{d}_A \in \mathrm{Ac}^A$ *there exists a counterdecision* $\mathsf{d}_A^c \in \mathrm{Ac}^{\mathrm{Ag} \setminus A}$ *such that* $(q', x \cdot (\mathsf{d}_A, \mathsf{d}_A^c)) \in \mathsf{L}(y)$*.*

If $\mathcal{A}$ *is asymmetric, then the above item* (ii) *is substituted by the following:* (ii') *for all* $y \in \mathrm{Tr}$*, with* $\mathsf{r}(y) = (q, x)$*, there exists a set* $\mathrm{S} \subseteq \mathrm{D} \times \mathrm{Q}$*, with* $\mathrm{S} \models \delta(q, \mathsf{v}(x))$*, such that for all atoms* $(\mathsf{d}, q') \in \mathrm{S}$ *it holds that* $(q', x \cdot \mathsf{d}) \in \mathsf{L}(y)$*.*

In this paper, we only consider automata along with a co-Büchi acceptance condition $\mathrm{F} \subseteq \mathrm{Q}$. A run $\mathcal{R}$ on a AAT $\mathcal{T}$ for an AGCTA $\mathcal{A}$ with a co-Büchi condition is accepting iff for all its paths all states in F only occur finitely often. A tree $\mathcal{T}$ is accepted by $\mathcal{A}$ iff there is an accepting run of $\mathcal{A}$ on it. By $\mathcal{L}(\mathcal{A})$ we denote the language accepted by the automaton $\mathcal{A}$, i.e., the set of all the AATs that $\mathcal{A}$ accepts. $\mathcal{A}$ is said *empty* if $\mathcal{L}(\mathcal{A}) = \emptyset$. The emptiness problem for $\mathcal{A}$ is to decide whether $\mathcal{L}(\mathcal{A}) = \emptyset$.

We now define AGCTA with satellite.

Definition 5. *An* asymmetric *(resp.,* symmetric*)* AGCTA with satellite *(*AGCTAS*) is a tuple* $\langle \mathcal{A}, \mathcal{D} \rangle$*, where* $\mathcal{A} = \langle \Sigma \times \mathrm{Q}', \mathrm{Ag}, \mathrm{Ac}, \mathrm{Q}, \delta, q_0, \mathrm{F} \rangle$ *(resp.,* $\mathcal{A} = \langle \Sigma \times \mathrm{Q}', \mathrm{Ag}, \mathrm{Q}, \delta, q_0, \mathrm{F} \rangle$*) is an* asymmetric *(resp.,* symmetric*)* AGCTA *and* $\mathcal{D}$ *is a* satellite $\langle \Sigma', \mathrm{Q}', \delta', q_0' \rangle$*, where* $\Sigma \subseteq \Sigma'$ *and* Q' *are non-empty finite sets of* input symbols *and* states*,* $q_0' \in \mathrm{Q}'$ *is an* initial state*, and* $\delta' : \mathrm{Q}' \times \Sigma' \mapsto \mathrm{Q}'$ *is a* deterministic transition function*.*

For the coming definition we need an extra notation. Let f be a Boolean formula, by $f[p/q]$ we denote the formula in which all occurrences of p in f are replaced by q.

Definition 6. *An* AAT $\mathcal{T}$ *is accepted by an* asymmetric *(resp.,* symmetric*)* AGCTAS $\langle \mathcal{A}, \mathcal{D} \rangle$ *iff* $\mathcal{T}$ *is accepted by the* AGCTA *product-automata* $\mathcal{A}^* = \langle \Sigma, \mathrm{Ag}, \mathrm{Ac}, \mathrm{Q} \times \mathrm{Q}', \delta^*, (q_0, q_0'), \mathrm{F}^* \rangle$ *(resp.,* $\mathcal{A}^* = \langle \Sigma, \mathrm{Ag}, \mathrm{Q} \times \mathrm{Q}', \delta^*, (q_0, q_0'), \mathrm{F}^* \rangle$*), where* F^* *is the acceptance condition directly derived from* F *and* δ^* *is such that:* $\delta^*((q,p),\sigma) = \delta(q,(\sigma, p))[q'/(q', \delta'(p,\sigma))]$*, for* $\sigma \in \Sigma$ *and* $(q,p) \in \mathrm{Q} \times \mathrm{Q}'$.

In words, $\delta^*((q,p),\sigma)$ is obtained by substituting in $\delta(q,(\sigma,p))$ each occurrence of a state q' with a tuple of the form (q',p'), where $p' = \delta'(p,\sigma)$ is the new state of the satellite. As for AGCTA, we consider AGCTAS along with a co-Büchi acceptance condition. W.r.t. Definition 6, we have that $\mathrm{F}^* = \mathrm{F} \times \mathrm{Q}'$. Moreover, we set $\mathcal{L}(\langle \mathcal{A}, \mathcal{U} \rangle) = \mathcal{L}(\mathcal{A}^*)$.

Note that satellites are just a convenient way to describe an AGCTA in which the state space can be partitioned into two components, one of which is deterministic and independent from the other, and has no influence on the acceptance. Indeed, it is just a matter of technicality to see that AGCTAS inherit all the closure properties of the alternating automata. In particular, the following theorem shows how the separation between $\mathcal{A}$ and $\mathcal{U}$ enables a tight analysis of the complexity of the relative emptiness problem.

Theorem 3. *The emptiness problem for a symmetric (resp., asymmetric) co-Büchi* AGCTAS $\langle \mathcal{A}, \mathcal{D} \rangle$*, where* $\mathcal{A}$ *has* m *agents and* n *states and* $\mathcal{D}$ *has* n' *states, can be decided in time* $2^{O((n \cdot \log(n \cdot n'))^m)}$.

Proof (Sketch). The proof proceeds as follow. First, we use the bounded model theorem for symmetric AGCTA (see Theorem 2 of [SF06]), which asserts that an AGCTA accepts an AAT iff it accepts a $|atom(\mathcal{A}) \times \mathrm{Ag}|^{|\mathrm{Ag}|}$-bounded AAT, in order to obtain a linear translation from the symmetric AGCTA $\mathcal{A}$ to an asymmetric one $\mathcal{A}'$ with the same sets of agents and states and $|atom(\mathcal{A}) \times \mathrm{Ag}|$ actions, such that $\mathcal{L}(\mathcal{A}') \subseteq \mathcal{L}(\mathcal{A})$, and $\mathcal{L}(\mathcal{A}') = \emptyset$ iff $\mathcal{L}(\mathcal{A}) = \emptyset$. The transition function of $\mathcal{A}'$ is obtained from that of $\mathcal{A}$ by substituting each existential atom $((\Diamond, A), q')$ (resp., universal atom $((\Box, A), q')$) with the formula $\bigvee_{\mathsf{d}_A \in \mathrm{Ac}^A} \bigwedge_{\mathsf{d}_A^c \in \mathrm{Ac}^{\mathrm{Ag} \setminus A}} ((\mathsf{d}_A, \mathsf{d}_A^c), q')$ (resp., $\bigwedge_{\mathsf{d}_A \in \mathrm{Ac}^A} \bigvee_{\mathsf{d}_A^c \in \mathrm{Ac}^{\mathrm{Ag} \setminus A}} ((\mathsf{d}_A, \mathsf{d}_A^c), q')$. As second step, since $\mathcal{A}'$ can be seen as a classical alternating co-Büchi tree automaton with $(atom(\mathcal{A}) \times \mathrm{Ag})^{\mathrm{Ag}}$ as set of directions, we use an exponential-time translation that leads to an asymmetric nondeterministic Büchi AGCTA $\mathcal{A}''$ with the same sets of agents and actions and $2^{O(n \cdot \log(n))}$ states such that $\mathcal{L}(\mathcal{A}'') = \mathcal{L}(\mathcal{A})$ (see Theorem 1.2 of [MS95]). At this point, taking the product-automata between $\mathcal{A}''$ and the satellite $\mathcal{D}$ we obtain another asymmetric nondeterministic Büchi AGCTA $\mathcal{A}'''$ with $2^{O(n \cdot \log(n \cdot n'))}$ states such that $\mathcal{L}(\langle \mathcal{A}'', \mathcal{D} \rangle) = \mathcal{L}(\mathcal{A}''')$. Now, by construction, it is evident that $\mathcal{L}(\langle \mathcal{A}, \mathcal{D} \rangle) = \emptyset$ iff $\mathcal{L}(\mathcal{A}''') = \emptyset$. Finally, the emptiness of $\mathcal{A}'''$ can be checked in a quadratic running-time in the size of the transition function, which is polynomial in the number of states and exponential in the number of directions (see Theorem 2.2 of [VW86]). Overall, with this procedure, we obtain that the emptiness problem for symmetric (resp., asymmetric) co-Büchi AGCTAS is solveable in exponential time w.r.t. $n \cdot \log(n \cdot n')$ and double exponential in the number m of agents. Precisely, in time $2^{O((n \cdot \log(n \cdot n'))^m)}$.

4.2 From Path Formulas to Satellites

As mentioned before, an mATL* path formula is satisfied at a certain node of a path by taking into account both the future and the past. Although the past is unlimited, it only requires a finite representation. This is due to the fact that LTL formulas with past operators (pLTL) [Gab87, LPZ85] can be translated into automata on infinite words of bounded size [Var88], and that pLTL represents the temporal path core of mpATL* (as LTL is the corresponding one for ATL*). Here, we show how to build the satellite that represents the memory on the past in order to solve satisfiability and model-checking for mpATL*. To this aim, we introduce the following notation. A basic formula b in φ is a subformula of φ of the form $b = \langle\!\langle A_b \rangle\!\rangle \psi_b$. Observe that the present trace for b is irrelevant, so we directly write $\mathcal{G}, \rho \models b$. By $sub(\varphi)$ we denote the set of all basic subformulas of φ and by $dsub(\varphi) \subseteq sub(\varphi)$ the immediate subformulas of φ. Finally, we use the following abbreviations $\mathrm{AP}_\varphi = \mathrm{AP} \cup dsub(\varphi)$, $\mathrm{AP}^*_\varphi = \mathrm{AP} \cup sub(\varphi)$, $\mathrm{AP}^{pr}_\varphi = \mathrm{AP}_\varphi \cup \{present\}$, and $\mathrm{AP}^{*,pr}_\varphi = \mathrm{AP}^*_\varphi \cup \{present\}$.

Before showing the full satellite construction, we first show how to build it from a single basic formula $b = \langle\!\langle A_b \rangle\!\rangle \psi_b$. Let $\widehat{\psi}_b$ be the pLTL formula obtained by replacing in ψ_b all the occurrences of direct basic subformulas $b' \in dsub(b)$ by the label b' read as atomic proposition. By using a slight variation of the procedure developed in [Var88], we can translate $\widehat{\psi}_b$ into a universal co-Büchi word automaton[3] $\mathcal{U}_b = \langle 2^{\mathrm{AP}^{pr}_b}, \mathrm{Q}_b, \delta_b, \mathrm{Q}_{0b}, \mathrm{F}_b \rangle$, with a number of states at most exponential in $|\psi_b|$, that accepts all and only the infinite words on $2^{\mathrm{AP}^{pr}_b}$ that are models of $\widehat{\psi}_b$. By applying the classical subset construction to $\mathcal{U}_b$, we obtain the satellite $\mathcal{D}_b = \langle 2^{\mathrm{AP}^{pr}_b}, 2^{\mathrm{Q}_b}, \delta^d_b, \mathrm{Q}_{0b} \rangle$, where, for all sets $Q \subseteq \mathrm{Q}_b$ and labels $\sigma \subseteq \mathrm{AP}^{pr}_b$, it holds that $\delta^d_b(Q, \sigma) = \bigcup_{q \in Q} \delta_b(q, \sigma)$. To better understand the usefulness of the satellite $\mathcal{D}_b$, consider $\mathcal{U}_b$ after that a prefix w' of an infinite word $w \in (2^{\mathrm{AP}^{pr}_b})^\omega$ is read. Since $\mathcal{U}_b$ is universal, there exists a number of active states that are ready to continue with the evaluation of the remaining part of the word w. Consider now the satellite $\mathcal{D}_b$ after that the same prefix w' is read. Since $\mathcal{D}_b$ is deterministic, there is only one active state that, by construction, is the set of all the active states of $\mathcal{U}_b$. It is clear then that, using $\mathcal{D}_b$, we are able to maintain all possible computations of $\mathcal{U}_b$.

We now define two different satellites, which we use for satisfiability and model-checking. Regarding satisfiability, we have to maintain, at the same time, a memory for all path formulas ψ_b contained in the mpATL* formula φ that we want to check. To this aim, we build the product-satellite $\mathcal{D}_\varphi = \langle 2^{\mathrm{AP}^{*,pr}_\varphi}, \prod_{b \in sub(\varphi)} 2^{\mathrm{Q}_b}, \delta^d_\varphi, \prod_{b \in sub(\varphi)} \{\mathrm{Q}_{0b}\} \rangle$ over all the satellites $\mathcal{D}_b$, with $b \in sub(\varphi)$, where, for all $Q_b \subseteq \mathrm{Q}_b$ and $\sigma \subseteq \mathrm{AP}^{*,pr}_\varphi$, it is set $\delta^d_\varphi(\prod_{b \in sub(\varphi)} Q_b, \sigma) = \prod_{b \in sub(\varphi)} \{\delta^d_b(Q_b, \sigma \cap \mathrm{AP}^{pr}_b)\}$. Regarding model-checking, since we verify one basic formulas $b \in sub(\varphi)$ at a time, we build the product-satellite $\mathcal{D}^*_{b,P} = \langle 2^{\mathrm{AP}^{*,pr}_b}, \mathrm{Q}^*_b, \delta^*_b, P \rangle$ over all the satellites $\mathcal{D}_{b'}$, with $b' \in sub(b)$, where, for all $Q_{b'} \subseteq \mathrm{Q}_{b'}$ and $\sigma \subseteq \mathrm{AP}^{*,pr}_b$, it is set $\mathrm{Q}^*_b = \prod_{b' \in sub(b)} 2^{\mathrm{Q}_{b'}}$, $P \in \mathrm{Q}^*_b$, and $\delta^*_b(\prod_{b' \in sub(b)} Q_{b'}, \sigma) = \prod_{b' \in sub(b)} \{\delta^d_{b'}(Q_{b'}, \sigma \cap \mathrm{AP}^{pr}_{b'})\}$. Note that the size of the satellites $\mathcal{D}_\varphi$ and $\mathcal{D}^*_{b,P}$, i.e., the number of their states, is bounded by $2^{O(2^{|\varphi|})}$ and $2^{O(2^{|\psi_b|})}$, respectively.

[3] Word automata can be seen as tree automata in which the tree has just one path. Moreover, a universal word automaton accepts a word iff all its runs are accepting.

4.3 Satisfiability

The satisfiability procedure we now propose technically extends that used for ATL* in [Sch08] along with that for mCTL* in [KV06]. Such an extension is possible due to the fact that the memoryful quantification has no direct interaction with the strategic features of the logic. In particular as for ATL*, it is possible to show that every CGS model of an mpATL* formula φ can be transformed into an *explicit* CGT model of φ. Such a model includes a certificate for both the truth of each of its basic subformula b in the respective node of the tree and the strategy used by the agents A_b to achieve the goal described by the corresponding path formula ψ_b (for a formal definition see [Sch08]). The main difference of our definition of explicit models w.r.t. that given in [Sch08] is in the fact that the *witness* of a basic formula b does not start in the node from which the path formula ψ_b needs to be satisfied, but from the node in which the quantification is applied, i.e., the present node. This difference, which directly derives from the memoryful feature of mpATL*, is due to the request that ψ_b needs to be satisfied on a path that starts at the root of the model. The proof of an explicit model existence is exploited by constructing an AGCTAS that accepts all and only the explicit models of the specification. The proof follows that used in Theorem 4 of [Sch08] and changes w.r.t. the use of the satellite $\mathcal{D}_\varphi$ that helps the main automaton $\mathcal{A}$ whenever it needs to start with the verification of a given path formula ψ_b, with $b \in sub(\varphi)$. In particular, $\mathcal{A}$ needs to send to the successors of a node x labeled with b in the AAT given in input, all the states of the universal co-Büchi automaton $\mathcal{U}_b$ that are active after $\mathcal{U}_b$ has read the word derived by the trace starting in the root of the tree and ending in x. By extending an idea given in [KV06], this requirement is satisfied by $\mathcal{A}$ by defining the transition function, for the part of interest, as follows: $\delta(q_b,(\sigma,Q)) = ((\Box,\mathrm{Ag}),q_b) \wedge \bigwedge_{q\in Q_{|b}} \bigwedge_{q'\in\delta_b(q,\sigma\cap \mathrm{AP}_b\cup\{present\})}((\Box,\mathrm{Ag}),(q',new))$, where $b \in \sigma$ and $Q_{|b}$ is the state of $\mathcal{D}_b$ in the product-state set Q. Putting the above reasoning all together, the following result holds.

Theorem 4. *Given a* mpATL* *formula* φ*, we can build a symmetric co-Büchi* AGCTAS $\langle \mathcal{A}, \mathcal{D}_\varphi \rangle$*, where* $\mathcal{D}_\varphi$ *has* $2^{O(2^{|\varphi|})}$ *states and* $\mathcal{A}$ *has* $O(2^{|\varphi|})$ *states and contains all and only the agents used in* φ*, such that* $\mathcal{L}(\langle \mathcal{A}, \mathcal{D}_\varphi \rangle)$ *is exactly the set of all the tree models of* φ*.*

Using Theorems 3 and 4, we obtain that the check of the existence of a model for a given mpATL* specification φ can be done in time $2^{2^{O(|\varphi|^2)}}$, resulting in a 2EXPTIME algorithm in the size of φ. Since mpATL* subsumes mCTL*, which has a satisfiability problem 2EXPTIME-HARD [KV06], we then derive the following result.

Theorem 5. *The satisfiability problem for* mpATL* *is* 2EXPTIME-COMPLETE.

4.4 Model Checking

As for ATL*, for mpATL* we use a bottom-up model-checking algorithm. The procedure we propose extends that used for ATL* in [AHK02] by means of the satellite. Note that this procedure is different from that used for mCTL* in [KV06], which is top-down and uses a local model-checking method. I.e., it checks whether the initial state satisfies the formula. Contrarily, our procedure is a global model checking that returns all states satisfying the formula. We now give the main idea behind our procedure.

Consider a CGS $\mathcal{G}$ and an mpATL* formula φ. If one uses directly the procedure from [AHK02], each state s of $\mathcal{G}$ turns labeled by a basic subformula b of φ together with a possible initial trace ρ ending in s iff $\mathcal{G},\rho \models b$. Then, one can check whether $\mathcal{G},\rho \models b$ by building an AGCTA $\mathcal{A}$ such that $\mathcal{L}(\mathcal{A}) \neq \emptyset$ iff $\mathcal{G},\rho \models b$. Usually, $\mathcal{A}$ is the product of two different automata $\mathcal{A}_1$ and $\mathcal{A}_2$, where $\mathcal{A}_1$ is used to select, according to A_b agents' strategy, all subtrees coming from the unwinding of $\mathcal{G}$ starting at s and $\mathcal{A}_2$ is used to verify that such subtrees satisfy ψ_b with ρ being the present. Although this procedure seems reasonable, it cannot be used because of the fact that we have infinitely many possible initial traces, while the set of atomic proposition in a CGS, as well as the number of checks the procedure can perform, have to be finite. A solution we propose here is to substitute ρ with "finite information", which is supplied by a satellite. In particular observe that, to manage the memoryful quantification, we only need an amount of memory whose size just depends on the size of the formula. Indeed, suppose b is the innermost basic subformula of φ, it is possible to prove that, if we have two traces ρ_1 and ρ_2 with the same final state and such that the satellites $\mathcal{D}_b$ reading the two words related to ρ_1 and ρ_2 reach the same state, then $\mathcal{G},\rho_1 \models b$ iff $\mathcal{G},\rho_2 \models b$. Using this fact, we can substitute the trace of the above procedure, with the relative state of the satellite, thus partitioning the information carried by the traces into equivalence classes.

We now describe the complete model-checking procedure for mpATL*. We start with the innermost basic formulas b of φ and terminate with its direct basic subformulas. For the base case, we use an automaton similar to that used in the previous sketch. So, we can build an extended CGS $\mathcal{G}'$ such that each state s of $\mathcal{G}'$ is labeled by a pair (b,Q), with $Q \in 2^{\mathrm{Q}_b}$, iff $\mathcal{G},\rho \models b$, for all the traces ρ ending in s such that, when $\mathcal{D}_b$ reads the word related to ρ it reaches the state Q. For the iterative case, assume that there is an extended CGS $\mathcal{G}_b$ for which the satisfaction of all the basic subformulas of b has already been determined. Then, using $\mathcal{G}_b$, we can build an AGCTAS $\mathcal{A}_{P,Q}$, with $\mathcal{D}^*_{b,P}$ as satellite, where $P \in \mathrm{Q}^*_b$ and $Q \in 2^{\mathrm{Q}_b}$, such that $\mathcal{L}(\mathcal{A}_{P,Q}) \neq \emptyset$ iff $\mathcal{G},\rho \models b$, where the word related to ρ on the extended CGS $\mathcal{G}_b$ carries the satellite $\mathcal{D}^*_{b,P_0}$, with $P_0 = \prod_{b' \in sub(b)} 2^{\mathrm{Q}_{0b'}}$, to the state P and the satellite $\mathcal{D}_b$ to the state Q. As for the classical procedure, also the main automaton of $\mathcal{A}_{P,Q}$ is the product of two different automata. The first one selects, using the satellite and accordingly to A_b agents' strategy, all subtrees coming from the unwinding of $\mathcal{G}$ starting at s, which carry in their labeling also the atomic propositions related to the basic subformulas of b. The second one verifies that such subtrees satisfy ψ_b with P "being the present". The resulting automaton is then used to have an extended structure that also includes the satisfaction of the formula b. By applying the procedure recursively, we obtain an enriched model for each basic formula $b \in sub(\varphi)$. Hence, we can determine whether the input formula φ is satisfied by the original structure $\mathcal{G}$ or not (for more technical details, see the full version of the paper). By a simple calculation, it follows that the over all procedure takes time $|\mathcal{G}|^{2^{\mathcal{O}(|\varphi|^2)}}$, resulting in an algorithm that is in PTIME w.r.t. the size of $\mathcal{G}$ and in 2EXPTIME w.r.t. the size of φ. Since, by item 1 of Theorem 2, there is a linear translation from ATL* to mpATL* and ATL* has a model-checking problem that is PTIME-HARD w.r.t. $\mathcal{G}$ and 2EXPTIME-HARD w.r.t φ [AHK02], we then derive the following result.

Theorem 6. *The model checking problem for* mpATL* *is* PTIME-COMPLETE *w.r.t. the size of the model and* 2EXPTIME-COMPLETE *w.r.t. the size of the specification.*

References

[AHK02] Alur, R., Henzinger, T.A., Kupferman, O.: Alternating-Time Temporal Logic. JACM 49(5), 672–713 (2002)

[DTV00] Daniele, M., Traverso, P., Vardi, M.Y.: Strong Cyclic Planning Revisited. In: Biundo, S., Fox, M. (eds.) ECP 1999. LNCS, vol. 1809, pp. 35–48. Springer, Heidelberg (2000)

[EH86] Emerson, E.A., Halpern, J.Y.: "Sometimes" and "Not Never" Revisited: On Branching Versus Linear Time. JACM 33(1), 151–178 (1986)

[FHMV95] Fagin, R., Halpern, J.Y., Moses, Y., Vardi, M.Y.: Reasoning about Knowledge. MIT Press, Cambridge (1995)

[Gab87] Gabbay, D.M.: The Declarative Past and Imperative Future: Executable Temporal Logic for Interactive Systems. In: Banieqbal, B., Pnueli, A., Barringer, H. (eds.) Temporal Logic in Specification. LNCS, vol. 398, pp. 409–448. Springer, Heidelberg (1989)

[Jam04] Jamroga, W.: Strategic Planning Through Model Checking of ATL Formulae. In: Rutkowski, L., Siekmann, J.H., Tadeusiewicz, R., Zadeh, L.A. (eds.) ICAISC 2004. LNCS (LNAI), vol. 3070, pp. 879–884. Springer, Heidelberg (2004)

[JW95] Janin, D., Walukiewicz, I.: Automata for the Modal μ-Calculus and Related Results. In: Hájek, P., Wiedermann, J. (eds.) MFCS 1995. LNCS, vol. 969, pp. 552–562. Springer, Heidelberg (1995)

[KP95] Kupferman, O., Pnueli, A.: Once and For All. In: LICS 1995, pp. 25–35. IEEE Computer Society, Los Alamitos (1995)

[KV06] Kupferman, O., Vardi, M.Y.: Memoryful Branching-Time Logic. In: LICS 2006, pp. 265–274. IEEE Computer Society, Los Alamitos (2006)

[KVW00] Kupferman, O., Vardi, M.Y., Wolper, P.: An Automata Theoretic Approach to Branching-Time Model Checking. JACM 47(2), 312–360 (2000)

[LMS02] Laroussinie, F., Markey, N., Schnoebelen, P.: Temporal Logic with Forgettable Past. In: LICS 2002, pp. 383–392. IEEE Computer Society, Los Alamitos (2002)

[LPZ85] Lichtenstein, O., Pnueli, A., Zuck, L.D.: The Glory of the Past. In: LP 1985, pp. 196–218 (1985)

[MS87] Muller, D.E., Schupp, P.E.: Alternating Automata on Infinite Trees. TCS 54(2-3), 267–276 (1987)

[MS95] Muller, D.E., Schupp, P.E.: Simulating Alternating Tree Automata by Nondeterministic Automata: New Results and New Proofs of Theorems of Rabin, McNaughton and Safra. TCS 141, 69–107 (1995)

[NPP08] Niebert, P., Peled, D., Pnueli, A.: Discriminative Model Checking. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 504–516. Springer, Heidelberg (2008)

[OR94] Osborne, M.J., Rubinstein, A.: A course in game theory. MIT Press, Cambridge (1994)

[PV07] Pistore, M., Vardi, M.Y.: The Planning Spectrum - One, Two, Three, Infinity. In: JAIR 2007, vol. 30, pp. 101–132 (2007)

[Sch08] Schewe, S.: ATL* Satisfiability is 2ExpTime-Complete. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part II. LNCS, vol. 5126, pp. 373–385. Springer, Heidelberg (2008)

[SF06] Schewe, S., Finkbeiner, B.: Satisfiability and Finite Model Property for the Alternating-Time μ-Calculus. In: Ésik, Z. (ed.) CSL 2006. LNCS, vol. 4207, pp. 591–605. Springer, Heidelberg (2006)

[Var88] Vardi, M.Y.: A Temporal Fixpoint Calculus. In: POPL 1988, pp. 250–259 (1988)

[vdHWW02] van der Hoek, W., Wooldridge, M.: Tractable multiagent planning for epistemic goals. In: AAMAS 2002, pp. 1167–1174 (2002)

[VW86] Vardi, M.Y., Wolper, P.: Automata-Theoretic Techniques for Modal Logics of Programs. JCSS 32(2), 183–221 (1986)
[WW01] Woolridge, M.J.: Introduction to Multiagent Systems. John Wiley & Sons, Chichester (2001)

A Full Definition of mpATL* Syntax and Semantics

The syntax of mpATL* is formally defined as follows.

Definition 7. mpATL* state *(φ) and* path *(ψ)* formulas *are built inductively from the sets of atomic propositions* AP *and agents* Ag *using the following context-free grammar, where $p \in \mathrm{AP}$ and $A \subseteq \mathrm{Ag}$:*

1. $\varphi ::= \mathrm{present} \mid p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \langle\!\langle A \rangle\!\rangle \psi \mid [\![A]\!] \psi$;
2. $\psi ::= \varphi \mid \neg\psi \mid \psi \wedge \psi \mid \psi \vee \psi \mid \mathsf{X}\psi \mid \mathsf{Y}\psi \mid \widetilde{\mathsf{Y}}\psi \mid \psi \mathsf{U} \psi \mid \psi \mathsf{S} \psi \mid \psi \mathsf{R} \psi \mid \psi \mathsf{B} \psi$.

The class of mpATL* *formulas is the set of all the state formulas generated by the above grammar, in which the occurrences of the special atomic proposition* present *is in the scope of a strategy quantifier.*

The semantics of mpATL* is formally defined as follows.

Definition 8. *Given a* CGS $\mathcal{G} = \langle \mathrm{AP}, \mathrm{Ag}, \mathrm{Ac}, \mathrm{St}, \lambda, \tau, s_0 \rangle$ *and two initial traces* $\rho, \rho_p \in \mathrm{Trc}$*, where* $\rho = \rho' \cdot s$*,* $\rho' \in \mathrm{Trc} \cup \{\varepsilon\}$*, and* $s \in \mathrm{St}$*, it holds that:*

1. $\mathcal{G}, \rho, \rho_p \models \mathrm{present}$ *iff* $\rho = \rho_p$;
2. $\mathcal{G}, \rho, \rho_p \models p$*, for* $p \in \mathrm{AP}$*, iff* $p \in \lambda(s)$;
3. $\mathcal{G}, \rho, \rho_p \models \neg\varphi$ *iff not* $\mathcal{G}, \rho, \rho_p \models \varphi$*, that is* $\mathcal{G}, \rho, \rho_p \not\models \varphi$;
4. $\mathcal{G}, \rho, \rho_p \models \varphi_1 \wedge \varphi_2$ *iff* $\mathcal{G}, \rho, \rho_p \models \varphi_1$ *and* $\mathcal{G}, \rho, \rho_p \models \varphi_2$;
5. $\mathcal{G}, \rho, \rho_p \models \varphi_1 \vee \varphi_2$ *iff* $\mathcal{G}, \rho, \rho_p \models \varphi_1$ *or* $\mathcal{G}, \rho, \rho_p \models \varphi_2$;
6. $\mathcal{G}, \rho, \rho_p \models \langle\!\langle A \rangle\!\rangle \psi$ *iff there exists an s-defined strategy* f_A *of agents in A such that for all* f_A*-plays* π *it holds that* $\mathcal{G}, \rho' \cdot \pi, 0, \rho \models \psi$;
7. $\mathcal{G}, \rho, \rho_p \models [\![A]\!] \psi$ *iff for all the s-defined strategies* f_A *of agents in A there exists an* f_A*-play* π *such that* $\mathcal{G}, \rho' \cdot \pi, 0, \rho \models \psi$.

Moreover, for a path π*, and a number* $k \in \mathbb{N}$*, it holds that:*

8. $\mathcal{G}, \pi, k, \rho_p \models \varphi$ *iff* $\mathcal{G}, \pi_{\leq k}, \rho_p \models \varphi$;
9. $\mathcal{G}, \pi, k, \rho_p \models \neg\psi$ *iff not* $\mathcal{G}, \pi, k, \rho_p \models \psi$*, that is* $\mathcal{G}, \pi, k, \rho_p \not\models \psi$;
10. $\mathcal{G}, \pi, k, \rho_p \models \psi_1 \wedge \psi_2$ *iff* $\mathcal{G}, \pi, k, \rho_p \models \psi_1$ *and* $\mathcal{G}, \pi, k, \rho_p \models \psi_2$;
11. $\mathcal{G}, \pi, k, \rho_p \models \psi_1 \vee \psi_2$ *iff* $\mathcal{G}, \pi, k, \rho_p \models \psi_1$ *or* $\mathcal{G}, \pi, k, \rho_p \models \psi_2$;
12. $\mathcal{G}, \pi, k, \rho_p \models \mathsf{X}\psi$ *iff* $\mathcal{G}, \pi, k+1, \rho_p \models \psi$;
13. $\mathcal{G}, \pi, k, \rho_p \models \mathsf{Y}\psi$ *iff* $k > 0$ *and* $\mathcal{G}, \pi, k-1, \rho_p \models \psi$;
14. $\mathcal{G}, \pi, k, \rho_p \models \widetilde{\mathsf{Y}}\psi$ *iff* $k = 0$ *or* $\mathcal{G}, \pi, k-1, \rho_p \models \psi$;
15. $\mathcal{G}, \pi, k, \rho_p \models \psi_1 \mathsf{U} \psi_2$ *iff there is an index i, with* $k \leq i$*, such that* $\mathcal{G}, \pi, i, \rho_p \models \psi_2$ *and, for all indexes j, with* $k \leq j < i$*, it holds* $\mathcal{G}, \pi, j, \rho_p \models \psi_1$;
16. $\mathcal{G}, \pi, k, \rho_p \models \psi_1 \mathsf{S} \psi_2$ *iff there is an index i, with* $i \leq k$*, such that* $\mathcal{G}, \pi, i, \rho_p \models \psi_2$ *and, for all indexes j, with* $i < j \leq k$*, it holds* $\mathcal{G}, \pi, j, \rho_p \models \psi_1$;
17. $\mathcal{G}, \pi, k, \rho_p \models \psi_1 \mathsf{R} \psi_2$ *iff for all indexes i, with* $k \leq i$*, it holds that* $\mathcal{G}, \pi, i, \rho_p \models \psi_2$ *or there is an index j, with* $k \leq j < i$*, such that* $\mathcal{G}, \pi, j, \rho_p \models \psi_1$;
18. $\mathcal{G}, \pi, k, \rho_p \models \psi_1 \mathsf{B} \psi_2$ *iff for all indexes i, with* $i \leq k$*, it holds that* $\mathcal{G}, \pi, i, \rho_p \models \psi_2$ *or there is an index j, with* $i < j \leq k$*, such that* $\mathcal{G}, \pi, j, \rho_p \models \psi_1$;

Counting and Enumeration Problems with Bounded Treewidth*

Reinhard Pichler, Stefan Rümmele, and Stefan Woltran

Vienna University of Technology, Vienna, Austria
{pichler,ruemmele,woltran}@dbai.tuwien.ac.at

Abstract. By Courcelle's Theorem we know that any property of finite structures definable in monadic second-order logic (MSO) becomes tractable over structures with bounded treewidth. This result was extended to counting problems by Arnborg et al. and to enumeration problems by Flum et al. Despite the undisputed importance of these results for proving fixed-parameter tractability, they do not directly yield implementable algorithms. Recently, Gottlob et al. presented a new approach using monadic datalog to close the gap between theoretical tractability and practical computability for MSO-definable decision problems. In the current work we show how counting and enumeration problems can be tackled by an appropriate extension of the datalog approach.

1 Introduction

The most common problem type studied in algorithms and complexity theory is the class of *decision problems*, which usually ask whether a solution to a given problem instance exists, e.g., whether a given graph has a valid 3-coloring. On the other hand, *counting problems* ask how many solutions an instance possesses, e.g., how many different 3-colorings are possible in the given graph. Finally, *enumeration problems* require as an answer the output of all possible solutions, e.g., all possible 3-colorings of the graph. Unfortunately, many interesting decision problems are computationally intractable. Clearly, the corresponding counting and enumeration problems then are intractable as well. A promising approach for dealing with intractability comes from the area of *parameterized complexity theory* (see [9,12] for an overview). Thereby the complexity analysis is not only based on the input size but also on some additional (structural) property of the input, the *parameter*. Imagine that some problem admits an algorithm with running time $f(k) \cdot n^{\mathcal{O}(1)}$, where n is the input size, k is the parameter and f is some arbitrary (usually exponential) function. Problems which can be solved by such algorithms are called *fixed-parameter tractable* (FPT). The basic idea is that the running time of those algorithms is still feasible, as long as k remains sufficiently small. The *treewidth* of an input structure, which measures the degree of cyclicity (see Section 2 for a formal definition), is a commonly studied parameter.

Courcelle's Theorem [7] states that every decision problem definable in MSO is FPT (in fact, even linear in the input size) when parameterized by the treewidth of the input structure. This result was extended to counting problems by Arnborg et al. [2] as well

* Supported by the Austrian Science Fund (FWF), project P20704-N18.

E.M. Clarke and A. Voronkov (Eds.): LPAR-16, LNAI 6355, pp. 387–404, 2010.

as to enumeration problems by Flum et al. [11]. Moreover, Bagan [3] and Courcelle [8] showed that for enumeration problems, this result can be refined by stating that the delay between the output of two successive solutions is fixed-parameter linear. Those proofs are constructive in the sense that they transform the problem of evaluating an MSO formula into a tree language recognition problem, which is then solved via a finite tree automaton (FTA). Although those results are very useful for verifying that a given problem is FPT, they do not help in finding algorithms that are usable in practice, since even very simple MSO formulae quickly lead to a "state explosion" of the FTA [13]. Consequently, it was already stated in [15] that the algorithms derived via Courcelle's Theorem are "useless for practical applications". This creates the need for other tools that help the algorithm designer developing FPT algorithms for specific problems.

An alternative approach using monadic datalog was presented by Gottlob et al. [14]. They proved that for decision problems, the MSO evaluation problem can be transformed into a monadic datalog program, of which the evaluation is FPT (and even linear in the input size). Although the general transformation presented there does not lead to a better running time than the MSO to FTA transformation, it has been shown, that this datalog framework helps to find algorithms for specific problems that are indeed feasible in practice [14]. In [17], datalog was already used in an ad hoc manner to solve some MSO-definable counting problems (including #SAT – the problem of counting the number of satisfying truth assignments of a given propositional formula). However, it remained an open question, whether there exists an appropriate extension of monadic datalog, that is capable of solving *every* MSO-definable counting problem. Moreover, enumeration problems have been completely left out so far.

The goal of the current work is to systematically extend the datalog approach from [14] to counting and enumeration problems in order to give an affirmative answer to the question mentioned above. We identify a nontrivial extension of monadic datalog, which we call *quasi-guarded fragment of extended datalog* and show that this fragment allows us to solve every MSO-definable counting problem over structures with bounded treewidth. As a by-product, these extended datalog programs generate intermediate results which can be exploited by a post-processing algorithm to solve the corresponding enumeration problem.

As for the complexity, we prove a fixed-parameter linear time bound for our counting algorithms (assuming unit cost for arithmetic operations) and a delay between two successive solutions for the enumeration algorithms, that is fixed-parameter linear in the size of the input. Note that Bagan [3] and Courcelle [8] presented enumeration algorithms having a delay that is fixed-parameter linear in the size of the next solution. But since these approaches depend on an MSO to FTA transformation, their usefulness for solving concrete problems in practice is restricted for the same reason as stated for Courcelle's Theorem above.

Results. Our main contributions are:

- *Counting.* We identify an appropriate extension of datalog, capable of expressing every MSO-definable counting problem. Algorithms based on our MSO to datalog transformation solve these counting problems in fixed-parameter linear time, parameterized by the treewidth of the input structure (assuming unit cost for arithmetic operations).

– *Enumeration.* Building upon our algorithms for counting problems, we devise a post-processing method which solves the corresponding MSO-definable enumeration problems. We thus get an algorithm which outputs the results with fixed-parameter linear delay, parameterized by the treewidth of the input structure (assuming unit cost for arithmetic operations).

The paper is organized as follows. After recalling basic notations and results in Section 2, we prove the main result regarding MSO-definable counting problems in Section 3. In Section 4, we present a novel post-processing algorithm, which solves the corresponding enumeration problem. In Section 5, we illustrate our approach by the example of 3-COLORABILITY. A conclusion is given in Section 6.

2 Preliminaries

Finite Structures and Treewidth. A *(relational) signature* $\sigma = \{R_1, \ldots, R_n\}$ is a set of relation (or predicate) symbols. Each relation symbol $R \in \sigma$ has an associated arity $\mathrm{arity}(R) \geq 1$. A *finite structure* $\mathcal{A}$ over signature σ (or simply a σ-structure) consists of a finite *domain* $A = \mathrm{dom}(\mathcal{A})$ plus a relation $R^{\mathcal{A}} \subseteq A^{\mathrm{arity}(R)}$ for each $R \in \sigma$.

A *tree decomposition* $\mathcal{T}$ of a σ-structure $\mathcal{A}$ is a pair (T, χ), where T is a tree and χ maps each node n of T (we use $n \in T$ as a shorthand below) to a *bag* $\chi(n) \subseteq \mathrm{dom}(\mathcal{A}) = A$ with the following properties: (1) For each $a \in A$, there is an $n \in T$, s.t. $a \in \chi(n)$. (2) For each $R \in \sigma$ and each $(a_1, \ldots, a_\alpha) \in R^{\mathcal{A}}$, there is an $n \in T$, s.t. $\{a_1, \ldots, a_\alpha\} \subseteq \chi(n)$. (3) For each $n_1, n_2, n_3 \in T$, s.t. n_2 lies on the path from n_1 to n_3, $\chi(n_1) \cap \chi(n_3) \subseteq \chi(n_2)$ holds. The third condition is usually referred to as the *connectedness condition*. The *width* of a tree decomposition $\mathcal{T} = (T, \chi)$ is defined as $\max\{|\chi(n)| \mid n \in T\} - 1$. The *treewidth* of $\mathcal{A}$, denoted as $\mathrm{tw}(\mathcal{A})$, is the minimal width of all tree decompositions of $\mathcal{A}$. For given $w \geq 1$, deciding if a given structure has treewidth $\leq w$ and, if so, to compute a tree decomposition of width w, is FPT [4]. We often have to assume that the elements in a bag $\chi(n)$ are ordered (in an arbitrary way). In this case, $\chi(n)$ is a tuple of elements, denoted as $\boldsymbol{a}$. By slight abuse of notation, we shall nevertheless apply set operations to such tuples (e.g., $a \in \boldsymbol{a}$, $\boldsymbol{a} \cap \boldsymbol{b}$) with the obvious meaning.

A tree decomposition $\mathcal{T} = (T, \chi)$ is called *normalized* (or *nice*) [19] if: (1) Each $n \in T$ has at most two children. (2) For each $n \in T$ with two children n_1, n_2, $\chi(n) = \chi(n_1) = \chi(n_2)$. (3) For each $n \in T$ with one child n', $\chi(n)$ and $\chi(n')$ differ in at most one element (i.e., $|\chi(n) \Delta \chi(n')| \leq 1$). (4) Leaf nodes $n \in T$ have empty bags (i.e., $\chi(n) = \emptyset$). W.l.o.g., we assume that every tree decomposition is normalized, since this normalization can be obtained in linear time without increasing the width [19].

A σ-structure $\mathcal{A}$ can be extended in order to additionally represent a tree decomposition $\mathcal{T}$ of $\mathcal{A}$. To this end, we extend σ to $\sigma_{td} = \sigma \cup \{\mathit{root}, \mathit{leaf}, \mathit{child}_1, \mathit{child}_2, \mathit{bag}\}$ by unary relation symbols *root* and *leaf* with the obvious meaning and binary relation symbols child_1 and child_2. Thereby $\mathit{child}_1(n_1, n)$ denotes that n_1 is the first or the only child of n and $\mathit{child}_2(n_2, n)$ denotes that n_2 is the second child of n. Finally, *bag* has arity $w + 2$, where $\mathit{bag}(n, a_0, \ldots, a_w)$ expresses $\chi(n) = (a_0, \ldots, a_w)$. For bags of smaller size, we assume that the remaining positions in the *bag*-predicate are padded with dummy elements. We write $\mathcal{A}_{td}$ to denote the σ_{td}-structure representing both $\mathcal{A}$

and $\mathcal{T}$, i.e., the domain of $\mathcal{A}_{td}$ consists of $\mathrm{dom}(\mathcal{A})$ plus the nodes of $\mathcal{T}$. Moreover, the relations in $\mathcal{A}_{td}$ coincide with $\mathcal{A}$ for every $R \in \sigma$ and in addition, $\mathcal{A}_{td}$ contains a representation of $\mathcal{T}$ via appropriate relations for $R \in \{root, leaf, child_1, child_2, bag\}$.

The nodes of tree decompositions are either element introduction (EI), element removal (ER), permutation (P), branch (B), or leaf (L) nodes. Thereby "element introduction", "element removal", and "permutation" refers to the way in which the bag of some node is obtained from the bag of its child.

Monadic Second-Order Logic (MSO). MSO extends first-order logic by the use of *set variables* or *second-order variables* (denoted by upper case letters), which range over sets of domain elements. In contrast, the *individual variables* or *first-order variables* (denoted by lower case letters) range over single domain elements. The *quantifier depth* of an MSO-formula φ is defined as the maximum degree of nesting of quantifiers (both for individual and set variables) in φ.

Let $\varphi(\boldsymbol{x}, \boldsymbol{X})$ be an MSO-formula with free variables $\boldsymbol{x} = (x_1, \ldots, x_n)$ and $\boldsymbol{X} = (X_1, \ldots, X_m)$. Furthermore, let $\mathcal{A}$ be a σ-structure with $A = \mathrm{dom}(\mathcal{A})$, let $\boldsymbol{a} \in A^n$ and $\boldsymbol{A} \in \mathcal{P}(A)^m$, where $\mathcal{P}(A)$ denotes the powerset of A. We write $(\mathcal{A}, \boldsymbol{a}, \boldsymbol{A}) \models \varphi(\boldsymbol{x}, \boldsymbol{X})$ to denote that $\varphi(\boldsymbol{a}, \boldsymbol{A})$ evaluates to true in $\mathcal{A}$. Usually, we refer to $(\mathcal{A}, \boldsymbol{a}, \boldsymbol{A})$ simply as a "structure" rather than a "structure with distinguished elements and sets". We call $(\boldsymbol{a}, \boldsymbol{A})$ a model of φ over $\mathcal{A}$ and denote by $\varphi(\mathcal{A})$ the set of all models over $\mathcal{A}$.

We call structures $(\mathcal{A}, \boldsymbol{a}, \boldsymbol{A})$ and $(\mathcal{B}, \boldsymbol{b}, \boldsymbol{B})$ *k-equivalent* and write $(\mathcal{A}, \boldsymbol{a}, \boldsymbol{A}) \equiv_k^{MSO} (\mathcal{B}, \boldsymbol{b}, \boldsymbol{B})$, iff for every MSO-formula $\varphi(\boldsymbol{x}, \boldsymbol{X})$ of quantifier depth $\leq k$, the equivalence $(\mathcal{A}, \boldsymbol{a}, \boldsymbol{A}) \models \varphi(\boldsymbol{x}, \boldsymbol{X}) \Leftrightarrow (\mathcal{B}, \boldsymbol{b}, \boldsymbol{B}) \models \varphi(\boldsymbol{x}, \boldsymbol{X})$ holds. By definition, $\equiv_k^{MSO}$ is an equivalence relation possessing only finitely many equivalence classes for any k. These classes are referred to as *k-types* or simply as *types*. There is a nice characterization of k-equivalence by *Ehrenfeucht-Fraïssé games*: The k-round MSO-game on two structures $(\mathcal{A}, \boldsymbol{a}, \boldsymbol{A})$ and $(\mathcal{B}, \boldsymbol{b}, \boldsymbol{B})$ is played between two players – the spoiler and the duplicator. Thereby $(\mathcal{A}, \boldsymbol{a}, \boldsymbol{A})$ and $(\mathcal{B}, \boldsymbol{b}, \boldsymbol{B})$ are k-equivalent iff the duplicator has a winning strategy in the game on these structures. For details, see e.g. [21].

Datalog. We assume some familiarity with datalog, see e.g. [5]. Syntactically, a datalog program Π is a set of function-free, definite Horn clauses, i.e., each clause consists of a non-empty head and a possibly empty body. The (minimal-model) semantics can be defined as the least fixpoint (LFP) of applying the immediate consequence operator. A predicate is called *extensional* if it occurs only in the body of the rules in Π, whereas predicates also occurring in the heads are called *intensional*.

Let $\mathcal{A}$ be a σ-structure. In the context of datalog, it is convenient to think of the relations $R^{\mathcal{A}}$ with $R \in \sigma$ as sets of ground atoms. The set of all such ground atoms of a structure $\mathcal{A}$ is referred to as the extensional database (EDB) of $\mathcal{A}$, which we shall denote as $\mathcal{E}(\mathcal{A})$. We have $R(\boldsymbol{a}) \in \mathcal{E}(\mathcal{A})$ iff $\boldsymbol{a} \in R^{\mathcal{A}}$.

The fragment of *quasi-guarded datalog* has been recently introduced in [14]:

Definition 1. *Let σ_{td} be the extension of a signature σ and let Π be a datalog program over σ_{td}. Moreover, let r be a rule in Π and let x, y be variables in r. Then y is called* functionally dependent on x in one step*, if the body of r contains an atom of one of the following forms: $child_1(x, y)$, $child_1(y, x)$, $child_2(x, y)$, $child_2(y, x)$, or $bag(x, a_0, \ldots, a_k)$ with $y = a_i$ for some $i \in \{1, \ldots, k\}$.*

Consequently, y is called functionally dependent on x *if there exists some $n \geq 1$ and variables $z_0, \ldots, z_n$ in r with $z_0 = x, z_n = y$ and, for every $i \in \{1, \ldots, n\}$, z_i is functionally dependent on z_{i-1} in one step.*

Furthermore, Π is called quasi-guarded *if every rule $r \in \Pi$ contains an extensional atom B, s.t., every variable occurring in r either occurs in B or is functionally dependent on some variable in B.*

Evaluating a datalog program Π over a structure $\mathcal{A}$ is EXPTIME-complete [23] in general. It becomes NP-complete [10] if the arity of the intensional predicate symbols is bounded. Quasi-guarded datalog programs Π can be evaluated over σ_{td}-structures $\mathcal{A}_{td}$ in time $\mathcal{O}(|\Pi| * |\mathcal{A}_{td}|)$ [14].

We extend quasi-guarded datalog by adding counter variables. Such variables are integers which may be used as an additional argument for intensional predicates. An intensional predicate p having n arguments plus a counter is denoted by $p(t_1, \ldots, t_n, j)$. Thereby the value of j is required to be fully determined by the grounding of $t_1, \ldots, t_n$. If the predicate p occurs in the head of some rule r, then the counter j may have one of the following four forms: (1) j is initialized by a constant c (e.g., $p(t_1, \ldots, t_n, c)$). (2) j takes the value of some j' being already present in the body of r (e.g., $p(t_1, \ldots, t_n, j')$). (3) j is the product of two counters j_1, j_2 occurring in r's body (e.g., $p(t_1, \ldots, t_n, j_1 * j_2)$). Additionally, we allow rules without being quasi-guarded but having the following strict form:

$$p(t_1, \ldots, t_n, \mathrm{SUM}(j)) \leftarrow q(t_1, \ldots, t_m, j).$$

where p, q are intensional predicates and $m > n$. In this case the semantics is similar to the SUM-aggregate function in ordinary SQL, where one first applies a GROUP BY over the variables $t_1, \ldots, t_n$. Consider for example the predicates $instLecture(n, j)$ stating that there are j lectures held at institute n, as well as $persLecture(n, p, j)$ stating that there are j lectures held by lecturer p at institute n. A possible rule expressing the relationship between these two predicates would be

$$instLecture(n, \mathrm{SUM}(j)) \leftarrow persLecture(n, p, j).$$

whose meaning is also captured by the SQL query

```
SELECT n, SUM(j) FROM persLecture GROUP BY n.
```

We call the resulting fragment *quasi-guarded extended datalog*. For a formal definition of the semantics of SUM, see [18]. Aggregate functions are well studied, starting with the first formalizations of Klug [20] through to more recent work, e.g. [1, 6, 16].

3 Counting Problems

In this section, we consider MSO-definable counting problems. We start by giving a formal definition of a more general counting problem. Let C denote a class of structures and Φ a class of logical formulae. Then the counting variant of model checking (MC) is defined as follows:

#MC(C,Φ)
Instance: A structure $\mathcal{A} \in C$ and a formula $\varphi \in \Phi$.
Parameter: $\|\varphi\|$.
Problem: Compute $\|\varphi(\mathcal{A})\|$.

Thereby $\|\cdot\|$ denotes the size of a reasonable encoding. In this paper, we only consider the case $\Phi = \text{MSO}$, i.e., φ is an MSO-formula. Moreover, when we study the relationship between #MC(C,MSO) problems and datalog, C will be restricted to structures whose treewidth is bounded by some constant.

We illustrate the above concepts by expressing #3-COLORABILITY (i.e., the counting variant of 3-COLORABILITY) as a #MC(C,MSO)-problem. As the class C we have the set of finite, undirected graphs or, equivalently, the σ-structures where σ consists of a single, binary relation symbol $edge$. We can define an MSO-formula $\varphi(R, G, B)$ as follows, expressing that the sets R, G, B form a valid 3-coloring:

$$\varphi(R,G,B) \equiv \mathit{Partition}(R,G,B) \wedge \forall v_1 \forall v_2 [\mathit{edge}(v_1,v_2) \rightarrow (\neg R(v_1) \vee \neg R(v_2)) \wedge (\neg G(v_1) \vee \neg G(v_2)) \wedge (\neg B(v_1) \vee \neg B(v_2))],$$

where $\mathit{Partition}(R, G, B)$ is used as a short-hand for the following formula:

$$\mathit{Partition}(R,G,B) \equiv \forall v [(R(v) \vee G(v) \vee B(v)) \wedge (\neg R(v) \vee \neg G(v)) \wedge (\neg R(v) \vee \neg B(v)) \wedge (\neg G(v) \vee \neg B(v))].$$

Suppose that a graph (V, E) is given by an $\{edge\}$-structure $\mathcal{A}$ having domain $\text{dom}(\mathcal{A}) = V$ and relation $edge^{\mathcal{A}} = E$. The above formula $\varphi(R, G, B)$ is one possible way of expressing that the sets R, G, B of vertices form a valid 3-coloring, i.e., R, G, B form a partition of V and, for every pair of vertices v_1, v_2, if they are adjacent then they are not both in R or both in G or both in B. Then we obtain the number of valid 3-colorings of (V, E) as

$$\left|\{\boldsymbol{A} \in \mathcal{P}(V)^3 \mid (\mathcal{A}, \boldsymbol{A}) \models \varphi(R,G,B)\}\right|,$$

i.e., the number of possible assignments to the free set-variables R, G, B in φ to make φ true in $\mathcal{A}$.

It is convenient to define *MSO-definable counting problems* as #MC(C,MSO) problems for formulae φ without free first-order variables. Note that this means no loss of generality since any free individual variable x can be replaced by a set variable X plus an appropriate conjunct in φ which guarantees that X is a singleton.

The primary goal of this section is to show that every MSO-definable counting problem over structures with bounded treewidth can be expressed by a program Π in the quasi-guarded extended datalog fragment defined in Section 2. Actually, *expressing* a problem in datalog also means *solving* the problem since, in contrast to MSO, datalog also has an operational semantics in addition to its declarative semantics. We shall therefore also analyze the complexity of evaluating such programs. This will ultimately allow us to give an alternative proof of the fixed-parameter linear time upper bound (assuming unit cost for arithmetic operations) on this class of counting problems.

The generic construction of a program Π corresponding to an MSO-formula φ (which will be detailed in the proof of Theorem 1) crucially depends on traversing a tree decomposition $\mathcal{T}$ in a bottom-up manner and reasoning about the k-type of the structure induced by the subtree of $\mathcal{T}$ rooted at each node n. Lemma 1 below allows us to establish the connection between the k-type of the structure induced by the subtree rooted at n and the k-type of the structure(s) induced by the subtree(s) rooted at the only child (resp. the two children) of n. We first define some additional terminology, which will be helpful for the formulation of this lemma.

Definition 2. *Let T be a tree with a node $n \in T$. Then we denote the* subtree rooted at *n as T_n. Likewise, let $\mathcal{A}$ be a finite structure and let $\mathcal{T} = (T, \chi)$ be a tree decomposition of $\mathcal{A}$. Then we define $\mathcal{T}_n$ as the restriction of $\mathcal{T}$ to the nodes of T_n and we denote by $\mathcal{A}_n$ the substructure of $\mathcal{A}$ induced by the elements of the bags of $\mathcal{T}_n$.*

Definition 3. *Let $m \geq 1$ be an integer and let $\mathcal{A}$ and $\mathcal{B}$ be σ-structures. Moreover, let $\boldsymbol{a} = (a_0, \ldots, a_m)$ and $\boldsymbol{b} = (b_0, \ldots, b_m)$ be tuples with $a_i \in \mathrm{dom}(\mathcal{A})$ and $b_i \in \mathrm{dom}(\mathcal{B})$. We call $\boldsymbol{a}$ and $\boldsymbol{b}$* equivalent *and write $\boldsymbol{a} \equiv \boldsymbol{b}$, iff for all predicate symbols $R \in \sigma$ with $\alpha = \mathrm{arity}(R)$ and for all tuples $(i_1, \ldots, i_\alpha) \in \{0, \ldots, m\}^\alpha$, the equivalence $R^{\mathcal{A}}(a_{i_1}, \ldots, a_{i_\alpha}) \Leftrightarrow R^{\mathcal{B}}(b_{i_1}, \ldots, b_{i_\alpha})$ holds.*

Lemma 1. *Given σ-structures $\mathcal{A}$ and $\mathcal{B}$, let $\mathcal{S}$ (resp. $\mathcal{T}$) be a normalized tree decomposition of $\mathcal{A}$ (resp. $\mathcal{B}$) having width w and let n (resp. m) be an internal node in $\mathcal{S}$ (resp. $\mathcal{T}$). Let n' (resp. m') denote the only or left child of n (resp. m) and let n'' (resp. m'') denote the optional right child. Let $\boldsymbol{a}, \boldsymbol{a}', \boldsymbol{a}'', \boldsymbol{b}, \boldsymbol{b}'$, and $\boldsymbol{b}''$ denote the bags of nodes n, n', n'', m, m', and m'', respectively. Furthermore consider l-tuples of domain-subsets $\boldsymbol{X} \in \mathcal{P}(\mathrm{dom}(\mathcal{A}_n))^l$, $\boldsymbol{X}' \in \mathcal{P}(\mathrm{dom}(\mathcal{A}_{n'}))^l$, $\boldsymbol{X}'' \in \mathcal{P}(\mathrm{dom}(\mathcal{A}_{n''}))^l$, $\boldsymbol{Y} \in \mathcal{P}(\mathrm{dom}(\mathcal{B}_m))^l$, $\boldsymbol{Y}' \in \mathcal{P}(\mathrm{dom}(\mathcal{B}_{m'}))^l$ and $\boldsymbol{Y}'' \in \mathcal{P}(\mathrm{dom}(\mathcal{B}_{m''}))^l$.*

- **(P) nodes:** *Let n and m be of type (P). If $\boldsymbol{X} = \boldsymbol{X}'$, $\boldsymbol{Y} = \boldsymbol{Y}'$ and there exists a permutation π, s.t. $\boldsymbol{a} = \pi(\boldsymbol{a}')$ and $\boldsymbol{b} = \pi(\boldsymbol{b}')$, then $(\mathcal{A}_{n'}, \boldsymbol{a}', \boldsymbol{X}') \equiv_k^{MSO} (\mathcal{B}_{m'}, \boldsymbol{b}', \boldsymbol{Y}')$ implies $(\mathcal{A}_n, \boldsymbol{a}, \boldsymbol{X}) \equiv_k^{MSO} (\mathcal{B}_m, \boldsymbol{b}, \boldsymbol{Y})$.*
- **(ER) nodes:** *Let n and m be of type (ER), s.t. $\boldsymbol{a}' \setminus \boldsymbol{a} = \{a_j\}$ and $\boldsymbol{b}' \setminus \boldsymbol{b} = \{b_j\}$, i.e. the removed elements have the same index. If $\boldsymbol{X} = \boldsymbol{X}'$ and $\boldsymbol{Y} = \boldsymbol{Y}'$, then $(\mathcal{A}_{n'}, \boldsymbol{a}', \boldsymbol{X}') \equiv_k^{MSO} (\mathcal{B}_{m'}, \boldsymbol{b}', \boldsymbol{Y}')$ implies $(\mathcal{A}_n, \boldsymbol{a}, \boldsymbol{X}) \equiv_k^{MSO} (\mathcal{B}_m, \boldsymbol{b}, \boldsymbol{Y})$.*
- **(EI) nodes:** *Let n and m be of type (EI), s.t. $\boldsymbol{a} \setminus \boldsymbol{a}' = \{a_j\}$ and $\boldsymbol{b} \setminus \boldsymbol{b}' = \{b_j\}$. If $\boldsymbol{a} \equiv \boldsymbol{b}$ and there exists $(\varepsilon_1, \ldots, \varepsilon_l) \in \{0, 1\}^l$, s.t. $X_i = X_i'$ respectively $Y_i = Y_i'$ if $\varepsilon_i = 0$, and $X_i = X_i' \cup \{a_j\}$ respectively $Y_i = Y_i' \cup \{b_j\}$ if $\varepsilon_i = 1$, then $(\mathcal{A}_{n'}, \boldsymbol{a}', \boldsymbol{X}') \equiv_k^{MSO} (\mathcal{B}_{m'}, \boldsymbol{b}', \boldsymbol{Y}')$ implies $(\mathcal{A}_n, \boldsymbol{a}, \boldsymbol{X}) \equiv_k^{MSO} (\mathcal{B}_m, \boldsymbol{b}, \boldsymbol{Y})$.*
- **(B) nodes:** *Let n and m be of type (B). If $\boldsymbol{X} = (X_1' \cup X_1'', \ldots, X_n' \cup X_n'')$ and $\boldsymbol{Y} = (Y_1' \cup Y_1'', \ldots, Y_n' \cup Y_n'')$, then $(\mathcal{A}_{n'}, \boldsymbol{a}', \boldsymbol{X}') \equiv_k^{MSO} (\mathcal{B}_{m'}, \boldsymbol{b}', \boldsymbol{Y}')$ and $(\mathcal{A}_{n''}, \boldsymbol{a}'', \boldsymbol{X}'') \equiv_k^{MSO} (\mathcal{B}_{m''}, \boldsymbol{b}'', \boldsymbol{Y}'')$ imply $(\mathcal{A}_n, \boldsymbol{a}, \boldsymbol{X}) \equiv_k^{MSO} (\mathcal{B}_m, \boldsymbol{b}, \boldsymbol{Y})$.*

Proof Idea. The proof proceeds by a case distinction over the four possible types of internal nodes n and m in a normalized tree decomposition. All cases are shown by an easy argument using Ehrenfeucht-Fraïssé games (see [21]). By the k-equivalence $(\mathcal{A}_{n'}, \boldsymbol{a}', \boldsymbol{X}') \equiv_k^{MSO} (\mathcal{B}_{m'}, \boldsymbol{b}', \boldsymbol{Y}')$ and, optionally, $(\mathcal{A}_{n''}, \boldsymbol{a}'', \boldsymbol{X}'') \equiv_k^{MSO} (\mathcal{B}_{m''}, \boldsymbol{b}'', \boldsymbol{Y}'')$, the duplicator has a winning strategy in the k-round game played on the structures $(\mathcal{A}_{n'}, \boldsymbol{a}', \boldsymbol{X}')$ and $(\mathcal{B}_{m'}, \boldsymbol{b}', \boldsymbol{Y}')$ as well as on the structures $(\mathcal{A}_{n''}, \boldsymbol{a}'', \boldsymbol{X}'')$ and

$(\mathcal{B}_{m''}, \boldsymbol{b}'', \boldsymbol{Y}'')$. Then the winning strategy at the only child node of n and m (resp. at the two child nodes of n and m) can be extended (resp. combined) to a winning strategy in the game played on the structures $(\mathcal{A}_n, \boldsymbol{a}, \boldsymbol{X})$ and $(\mathcal{B}_m, \boldsymbol{b}, \boldsymbol{Y})$. Actually, in case of (P) and (ER) nodes, the winning strategy for $(\mathcal{A}_{n'}, \boldsymbol{a}', \boldsymbol{X}')$ and $(\mathcal{B}_{m'}, \boldsymbol{b}', \boldsymbol{Y}')$ may even be left unchanged. In order to extend the winning strategy in case of an (EI)-node and to combine the winning strategies in case of a (B)-node, the connectedness condition of tree decompositions is crucial. □

Lemma 1 gives us the intuition how to determine the k-type of the substructure induced by a subtree $\mathcal{T}_n$ via a bottom-up traversal of the tree decomposition $\mathcal{T}$. Essentially, the type of the structure induced by $\mathcal{T}_n$ is fully determined by three components: (i) the type of the structure induced by the subtree rooted at the child node(s) of n, (ii) the relations between elements in the bag at node n, and (iii) the intersection of the distinguished domain elements $\boldsymbol{a}$ and distinguished sets $\boldsymbol{X}$ with the elements in the bag at node n.

We now have a closer look at the distinguished sets and their effect on the type of a structure. Suppose that we have fixed some structure $\mathcal{A}$ together with distinguished elements $\boldsymbol{a}$. Then the question is if different choices $\boldsymbol{A}$ and $\boldsymbol{B}$ of distinguished sets necessarily lead to different types. In the following lemma we give a positive answer to this question for the case that $\boldsymbol{A}$ and $\boldsymbol{B}$ differ on an element in $\boldsymbol{a}$. This lemma will be very useful when, on our bottom-up traversal of the tree decomposition, we encounter an element introduction node: Let a denote the element that is new w.r.t. the bag at the child node and suppose that we are considering l distinguished sets. Then, by the lemma below, we know that each of the 2^l possible choices of either adding the element a to each of the l distinguished sets or not necessarily produces a different type. This property in turn is important for solving the #MC(C,MSO) problem since it guarantees that we do not count any solution twice. Similarly, in Section 4, it will keep us from outputting a solution twice.

Lemma 2. *Given a σ-structure $\mathcal{A}$ with $\boldsymbol{a} \in \mathrm{dom}(\mathcal{A})^m$ and $\boldsymbol{A}, \boldsymbol{B} \in \mathcal{P}(\mathrm{dom}(\mathcal{A}))^l$. If there exists an index $i \in \{1, \ldots, m\}$, s.t. $A_i \cap \boldsymbol{a} \neq B_i \cap \boldsymbol{a}$, then it follows that $(\mathcal{A}, \boldsymbol{a}, \boldsymbol{A}) \not\equiv_k^{MSO} (\mathcal{A}, \boldsymbol{a}, \boldsymbol{B})$.*

Proof. This follows directly from the definition of $\equiv_k^{MSO}$ through Ehrenfeucht-Fraïssé games. Indeed, suppose that some domain element $a \in \boldsymbol{a}$ is contained in some A_i but not in B_i (or vice versa). Then the spoiler can win the game on the structures $(\mathcal{A}, \boldsymbol{a}, \boldsymbol{A})$ and $(\mathcal{A}, \boldsymbol{a}, \boldsymbol{B})$ in a single move, simply by choosing a. □

We introduce one more auxiliary definition and then we will be ready to formulate and prove the main result of this section, namely Theorem 1.

Definition 4. *Let $\mathcal{A}$ be a σ-structure and let $\boldsymbol{a}$ be a tuple of elements of* $\mathrm{dom}(\mathcal{A})$. *Then $\mathcal{R}(\boldsymbol{a})$ denotes the set of all* ground atoms *with predicates in σ and arguments in $\boldsymbol{a}$, i.e. $\mathcal{R}(\boldsymbol{a}) = \{R(a_1, \ldots, a_\alpha) \mid R \in \sigma, \alpha = \mathrm{arity}(R), a_1, \ldots, a_\alpha \in \boldsymbol{a}\}$.*

Theorem 1. *Let signature σ and integer $w \geq 1$ be arbitrary but fixed. For the class C of σ-structures of treewidth at most w, the problem* #MC(C,MSO) *is definable in the quasi-guarded fragment of extended datalog over σ_{td}.*

Proof. Let $\varphi(\boldsymbol{X})$ be an arbitrary MSO-formula with free second-order variables $\boldsymbol{X} = X_1, \ldots, X_l$ and quantifier depth k. We will construct a quasi-guarded extended datalog program Π with a distinguished predicate *solution*, s.t. for any σ-structure $\mathcal{C} \in C$, *solution*(j) is in the LFP of $\Pi \cup \mathcal{C}_{td}$ iff $|\varphi(\mathcal{C})| = j$.

During the construction we maintain a set Θ of rank-k types ϑ of structures of the form $(\mathcal{A}, \boldsymbol{a}, \boldsymbol{A})$, where $\mathrm{tw}(\mathcal{A}) \leq w$, $\mathcal{T}$ is a tree decomposition of $\mathcal{A}$ with width $\mathrm{tw}(\mathcal{A})$, $n = \mathrm{root}(\mathcal{T})$, $\boldsymbol{a} = \mathrm{bag}(n)$ and $\boldsymbol{A} \in \mathcal{P}(\mathrm{dom}(\mathcal{A}))^l$. For each type $\vartheta \in \Theta$, we save a witness denoted by $W(\vartheta) = (\mathcal{A}, \boldsymbol{a}, \boldsymbol{A})$. The types $\vartheta \in \Theta$ will serve as names of binary predicates in our program $\mathcal{P}$ as well as names for constants. No confusion will arise from this "overloading" since the meaning will always be clear from the context.

It is important to notice that the construction of program Π only depends on $\varphi(\boldsymbol{X})$ and the upper bound w on the treewidth of the structures in C but not on a concrete structure $\mathcal{C} \in C$. But of course, Π will ultimately be used to compute $|\varphi(\mathcal{C})|$ for a concrete input structure $\mathcal{C}$. The intended meaning of the ϑ-predicate (which holds for any structure $\mathcal{C} \in C$) in our program construction is captured by Property A below.

Let $\vartheta \in \Theta$, let $\mathcal{C} \in C$ and let n be a node in a tree decomposition $\mathcal{T}$ of $\mathcal{C}$, s.t. the bag at n is $\boldsymbol{b}$. Then we define the set $\Gamma(n, \vartheta)$ as

$$\Gamma(n, \vartheta) = \{\boldsymbol{B} \in \mathcal{P}(\mathrm{dom}(\mathcal{C}_n))^l \mid (\mathcal{C}_n, \boldsymbol{b}, \boldsymbol{B}) \equiv_k^{MSO} W(\vartheta)\}.$$

(Recall from Definition 2 that $\mathcal{C}_n$ denotes the substructure of $\mathcal{C}$ induced by the elements in the bags of the subtree $\mathcal{T}_n$ rooted at n.) Then, we have

Property A. For every $j \geq 1$, there exists an atom $\vartheta(n, j)$ in the LFP of $\Pi \cup \mathcal{C}_{td}$ iff $|\Gamma(n, \vartheta)| = j$. Furthermore, there exists no atom $\vartheta(n, \_)$ in the LFP, iff $|\Gamma(n, \vartheta)| = 0$.

We shall show that Property A indeed holds at the end of the proof. First, we give the details of the construction of program Π. Initially, we set $\Theta = \Pi = \emptyset$. Then we construct Θ by structural induction over normalized tree decompositions. Since there are only finitely many MSO rank-k types for structures $(\mathcal{A}, \boldsymbol{a}, \boldsymbol{A})$ [21], the induction will eventually halt. For the base case of the induction, we consider a tree decomposition $\mathcal{T}$ consisting of a single node n with empty bag $\boldsymbol{a}$. To create structure $(\mathcal{A}, \boldsymbol{a}, \boldsymbol{A})$ having tree decomposition $\mathcal{T}$, let $\mathcal{A}$ be the σ-structure with $\mathrm{dom}(\mathcal{A}) = \emptyset$. Moreover $\boldsymbol{A} = (\emptyset, \ldots, \emptyset)$. Now we invent a new token ϑ, add it to Θ and save the witness $W(\vartheta) = (\mathcal{A}, \boldsymbol{a}, \boldsymbol{A})$. Additionally we add the following rule to Π:

$$\vartheta(n, 1) \leftarrow \mathit{leaf}(n).$$

In the induction step, we construct all possible structures $(\mathcal{A}, \boldsymbol{a}, \boldsymbol{A})$ that can be created by extending the tree decomposition of witness $W(\vartheta') = (\mathcal{A}', \boldsymbol{a}', \boldsymbol{A}')$ for any $\vartheta' \in \Theta$ in a "bottom-up" manner by introducing a new root node. Let $\boldsymbol{a}' = (a_0, \ldots, a_m)$ be the bag of the old root n'. The new root n can be of any of the following node types:

(P) node: Consider all possible permutations π of the indices $\{0, \ldots, m\}$ and set $\boldsymbol{a} = (a_{\pi(0)}, \ldots, a_{\pi(m)})$, $\mathcal{A} = \mathcal{A}'$, and $\boldsymbol{A} = \boldsymbol{A}'$. For each of these $(\mathcal{A}, \boldsymbol{a}, \boldsymbol{A})$, we check whether there exists $\vartheta \in \Theta$ with witness $W(\vartheta) = (\mathcal{B}, \boldsymbol{b}, \boldsymbol{B})$, s.t. $(\mathcal{A}, \boldsymbol{a}, \boldsymbol{A}) \equiv_k^{MSO} (\mathcal{B}, \boldsymbol{b}, \boldsymbol{B})$. If such a ϑ is found we take it, otherwise we invent a new token ϑ, add it to Θ and save the witness $W(\vartheta) = (\mathcal{A}, \boldsymbol{a}, \boldsymbol{A})$. In either case, we add the following rules to Π. Note that we do not write down the dummy elements for smaller bags.

$$\begin{aligned} \mathit{aux}_P(n, \vartheta, \vartheta', j) \leftarrow\ & \mathit{bag}(n, x_{\pi(0)}, \ldots, x_{\pi(m)}), \mathit{child}(n', n), \\ & \mathit{bag}(n', x_0, \ldots, x_m), \vartheta'(n', j). \\ \vartheta(n, j) \leftarrow\ & \mathit{aux}_P(n, \vartheta, \_, j). \end{aligned}$$

(ER) node: Set $\boldsymbol{a} = (a_1, \ldots, a_m)$, $\mathcal{A} = \mathcal{A}'$ and $\boldsymbol{A} = \boldsymbol{A}'$. For each of these $(\mathcal{A}, \boldsymbol{a}, \boldsymbol{A})$ we check whether there exists $\vartheta \in \Theta$ with $W(\vartheta) = (\mathcal{B}, \boldsymbol{b}, \boldsymbol{B})$, s.t. $(\mathcal{A}, \boldsymbol{a}, \boldsymbol{A}) \equiv_k^{MSO} (\mathcal{B}, \boldsymbol{b}, \boldsymbol{B})$. If no such ϑ is found, we invent a new token ϑ, add it to Θ and save $W(\vartheta) = (\mathcal{A}, \boldsymbol{a}, \boldsymbol{A})$. In either case, the following rule is added to Π:

$$\begin{aligned} \mathit{aux}_R(n, \vartheta, \vartheta', j) \leftarrow\ & \mathit{bag}(n, x_1, \ldots, x_m), \mathit{child}_1(n', n), \\ & \mathit{bag}(n', x_0, x_1, \ldots, x_m), \vartheta'(n', j). \end{aligned}$$

For all $\vartheta \in \Theta$ for which we created the rule above, we also add:

$$\vartheta(n, \mathrm{SUM}(j)) \leftarrow \mathit{aux}_R(n, \vartheta, \_, j).$$

(EI) node: Adding an (EI) node is only possible if $m < w$. We take a new element $a_{m+1} \notin \mathrm{dom}(\mathcal{A})$ and let $\boldsymbol{a} = (a_0, \ldots, a_m, a_{m+1})$. All possible structures $\mathcal{A}$ can be generated by setting $\mathrm{dom}(\mathcal{A}) = \mathrm{dom}(\mathcal{A}') \cup \{a_{m+1}\}$ and by extending the EDB $\mathcal{E}(\mathcal{A}')$ to $\mathcal{E}(\mathcal{A})$ in the following way. Let $\Delta = \mathcal{E}(\mathcal{A}) \setminus \mathcal{E}(\mathcal{A}')$ be an arbitrary set of tuples, s.t. $\Delta \subseteq \mathcal{R}(\boldsymbol{a})$ and a_{m+1} occurs as an argument of all tuples in Δ. Furthermore we consider all possible tuples $\boldsymbol{\varepsilon} = (\varepsilon_1, \ldots, \varepsilon_l) \in \{0, 1\}^l$ and extend $\boldsymbol{A}'$ to $\boldsymbol{A}$, s.t. $A_i = A_i'$ if $\varepsilon_i = 0$ and $A_i = A_i' \cup \{a_{m+1}\}$ if $\varepsilon_i = 1$. For every such structure, i.e. for each combination of $\mathcal{A}$ and $\boldsymbol{A}$, we check whether there exists $\vartheta \in \Theta$ with $W(\vartheta) = (\mathcal{B}, \boldsymbol{b}, \boldsymbol{B})$, s.t. $(\mathcal{A}, \boldsymbol{a}, \boldsymbol{A}) \equiv_k^{MSO} (\mathcal{B}, \boldsymbol{b}, \boldsymbol{B})$. If no such ϑ is found, we invent a new token ϑ, add it to Θ and save $W(\vartheta) = (\mathcal{A}, \boldsymbol{a}, \boldsymbol{A})$. In either case, the following rule is added to Π:

$$\begin{aligned} \mathit{aux}_I(n, \vartheta, \vartheta', \boldsymbol{\varepsilon}, j) \leftarrow\ & \mathit{bag}(n, x_0, \ldots, x_{m+1}), \mathit{child}_1(n', n), \\ & \mathit{bag}(n', x_0, \ldots, x_m), \vartheta'(n', j), \\ & \{R(x_{i_1}, \ldots, x_{i_r}) \mid R(a_{i_1}, \ldots, a_{i_r}) \in \mathcal{E}(\mathcal{A})\}, \\ & \{\neg R(x_{i_1}, \ldots, x_{i_r}) \mid R(a_{i_1}, \ldots, a_{i_r}) \notin \mathcal{E}(\mathcal{A})\}. \end{aligned}$$

For all $\vartheta \in \Theta$ for which we created the rule above, we also add:

$$\vartheta(n, \mathrm{SUM}(j)) \leftarrow \mathit{aux}_I(n, \vartheta, \_, \_, j).$$

(B) node: Let $\vartheta'' \in \Theta$ be a type with $W(\vartheta'') = (\mathcal{A}'', \boldsymbol{a}'', \boldsymbol{A}'')$, not necessarily distinct from ϑ'. W.l.o.g. we require $\boldsymbol{a}' = \boldsymbol{a}''$ and $\mathrm{dom}(\mathcal{A}') \cap \mathrm{dom}(\mathcal{A}'') = \boldsymbol{a}'$. Note that this can easily be achieved by renaming the elements of one of the two structures. Furthermore we check for inconsistency of the EDBs of the two structures $(\mathcal{A}', \boldsymbol{a}', \boldsymbol{A}')$ and $(\mathcal{A}'', \boldsymbol{a}', \boldsymbol{A}'')$, i.e. we check whether $\mathcal{E}(\mathcal{A}') \cap \mathcal{R}(\boldsymbol{a}') \neq \mathcal{E}(\mathcal{A}'') \cap \mathcal{R}(\boldsymbol{a}')$ or $\boldsymbol{A}' \cap \boldsymbol{a}' \neq \boldsymbol{A}'' \cap \boldsymbol{a}'$. If this is true, we ignore the pair, otherwise we create the new structure $(\mathcal{A}, \boldsymbol{a}, \boldsymbol{A})$ by setting $\mathrm{dom}(\mathcal{A}) = \mathrm{dom}(\mathcal{A}') \cup \mathrm{dom}(\mathcal{A}'')$, $\mathcal{E}(\mathcal{A}) = \mathcal{E}(\mathcal{A}') \cup \mathcal{E}(\mathcal{A}'')$, $\boldsymbol{a} = \boldsymbol{a}'$ and $A_i = A_i' \cup A_i''$. For every such structure, we check whether there exists $\vartheta \in \Theta$ with witness $W(\vartheta) = (\mathcal{B}, \boldsymbol{b}, \boldsymbol{B})$, s.t. $(\mathcal{A}, \boldsymbol{a}, \boldsymbol{A}) \equiv_k^{MSO} (\mathcal{B}, \boldsymbol{b}, \boldsymbol{B})$. If such a ϑ is

found we take it, otherwise we invent a new token ϑ, add it to Θ and save the witness $W(\vartheta) = (\mathcal{A}, \boldsymbol{a}, \boldsymbol{A})$. In either case, we add the following rule to Π:

$$\begin{aligned} \mathit{aux}_B(n, \vartheta, \vartheta_1, \vartheta_2, j_1 * j_2) \leftarrow\ & \mathit{child}_1(n_1, n), \mathit{bag}(n_1, x_0, \ldots, x_m), \\ & \mathit{child}_2(n_2, n), \mathit{bag}(n_2, x_0, \ldots, x_m), \\ & \mathit{bag}(n, x_0, \ldots, x_m), \vartheta_1(n_1, j_1), \vartheta_2(n_2, j_2). \end{aligned}$$

For all $\vartheta \in \Theta$ for which we created the rule above, we also add:

$$\vartheta(n, \mathrm{SUM}(j)) \leftarrow \mathit{aux}_B(n, \vartheta, \_, \_, j).$$

Now that we have constructed Θ, the only thing left to do is the actual counting of solutions. To this end, we check for every $\vartheta \in \Theta$ with $W(\vartheta) = (\mathcal{A}, \boldsymbol{a}, \boldsymbol{A})$, whether $\mathcal{A} \models \varphi(\boldsymbol{A})$ holds and, if the answer is affirmative, we add the following rule to $\mathcal{P}$:

$$\mathit{aux}_{\mathit{root}}(\vartheta, j) \leftarrow \mathit{root}(n), \vartheta(n, j).$$

Finally we also add:

$$\mathit{solution}(\mathrm{SUM}(j)) \leftarrow \mathit{aux}_{\mathit{root}}(\_, j).$$

The bottom-up construction of Θ guarantees that we construct all possible rank-k types of structures $(\mathcal{A}, \boldsymbol{a}, \boldsymbol{A})$, where $\mathrm{tw}(\mathcal{A}) \leq w$ and $\boldsymbol{a}$ is the bag of the root node of a tree decomposition of $\mathcal{A}$. This follows from Lemma 1 and an easy induction argument.

Now consider an arbitrary input structure $\mathcal{C} \in C$ with tree decomposition $\mathcal{T}$. We have to show that $\mathit{solution}(j)$ is in the LFP of $\Pi \cup \mathcal{C}_{td}$ iff $|\varphi(\mathcal{C})| = j$. By the above two rules with head predicates $\mathit{aux}_{\mathit{root}}$ and *solution*, it suffices to show that the ϑ-predicate has the desired Property A. We prove this by discussing the program rules added for each node type. Below, we restrict ourselves to the case that $|\Gamma(n, \vartheta)| \geq 1$. The case $|\Gamma(n, \vartheta)| = 0$ (i.e., no fact $\vartheta(n, \_)$ is in the LFP of $\Pi \cup \mathcal{C}_{td}$) is obvious.

(L) nodes: For an (L) node n with type ϑ, $\Gamma(n, \vartheta) = \{(\emptyset, \ldots, \emptyset)\}$. Since the only rule for deriving $\vartheta(n, j)$ sets $j = 1$, the statement holds.

(P) nodes: Rules for a (P) node n with type ϑ do not alter the counter j. But neither does a permutation of the elements in the root bag change the corresponding set $\Gamma(n, \vartheta)$.

(ER) nodes: For an (ER) node n with type ϑ, the rule deriving aux_R does not alter j. If the type ϑ' of child n' changes for solution $\boldsymbol{A}'$ stored in $W(\vartheta')$ to the type ϑ, then by Lemma 1, this is also true for all $\boldsymbol{B}' \in \Gamma(n', \vartheta')$. For the second rule note that, for two types ϑ' and ϑ'' both leading to ϑ, we have $\Gamma(n', \vartheta') \cap \Gamma(n', \vartheta'') = \emptyset$. Thus $|\Gamma(n, \vartheta)|$ can be computed by summation over all the counters appearing in a derived fact aux_R.

(EI) nodes: For an (EI) node n with type ϑ, the rule deriving aux_I does not alter j either. If the modification of solution $\boldsymbol{A}'$ stored in $W(\vartheta')$ according to the tuple $\boldsymbol{\varepsilon} = (\varepsilon_1, \ldots, \varepsilon_l)$ changes the type ϑ' of child n' to the type ϑ, then by Lemma 1, this is also true for all $\boldsymbol{B}' \in \Gamma(v', \vartheta')$ under the same modification $\boldsymbol{\varepsilon}$. Lemma 2 states that a different modification $\boldsymbol{\varepsilon}'$ leads to a type different from ϑ. Therefore and by the same argument as for (ER) nodes, $|\Gamma(n, \vartheta)|$ can be computed by summation over all the counters appearing in a derived fact aux_I.

(B) nodes: For a (B) node n, we know by Lemma 1, if the combination of solution $\boldsymbol{A_1}$ in $W(\vartheta_1)$ with solution $\boldsymbol{A_2}$ in $W(\vartheta_2)$ leads to ϑ, then any solution $\boldsymbol{B_1} \in \Gamma(n_1, \vartheta_1)$

combined with any solution $\boldsymbol{B_2} \in \Gamma(n_2, \vartheta_2)$ leads to ϑ. Since $\boldsymbol{B_1} \cap \boldsymbol{B_2} = \emptyset$ and from the connectedness condition of tree decompositions, it follows that there are indeed $|\Gamma(n_1, \vartheta_1)| * |\Gamma(n_2, \vartheta_2)| = j_1 * j_2$ possible different solutions that lead to ϑ through combination of ϑ_1 and ϑ_2. The second rule groups all solutions leading to ϑ together.

From the discussion above and from the rules for aux_{root} and *solution*, it follows that Π is the desired quasi-guarded extended datalog program. □

So far, we have shown hat every MSO-definable counting problem is indeed solvable by quasi-guarded extended datalog. The following result complements this expressibility result by showing that quasi-guarded extended datalog programs can be evaluated in linear time.

Theorem 2. *Let Π be a quasi-guarded extended datalog program and let $\mathcal{A}$ be a finite structure. Then Π can be evaluated over $\mathcal{A}$ in time $\mathcal{O}(\|\Pi\| * \|\mathcal{A}\|)$, assuming unit cost for arithmetic operations. Thereby $\|\Pi\|$ denotes the size of the extended datalog program and $\|\mathcal{A}\|$ denotes the size of the data.*

Proof. Gottlob et al. [14] showed that this theorem holds for quasi-guarded datalog without the extension by counter variables. By definition, the counter is fully determined from the other arguments of the predicate. Therefore, the number of possible groundings of a rule r does not increase and hence the argument in [14] is still valid for quasi-guarded extended datalog without the $\mathrm{SUM}(\cdot)$ operator. On the other hand, it is easy to see, that adding rules with this operator does not harm the linear time bound, since we could replace rule r by rule r', where $\mathrm{SUM}(j)$ is substituted by j. The facts for r can then be derived by summation of the counter of r'. Since we assume unit cost for arithmetic operations, this does not violate the linear time bound. □

Note that the construction and, therefore, the size of the datalog program Π in the proof of Theorem 1 only depends on the length of the formula φ and the treewidth w, but not on the structure $\mathcal{A}$. Hence, combining Theorem 1 and 2 immediately yields an alternative proof of the counting variant of Courcelle's Theorem as a corollary. Originally, this result was shown in [2] via the correspondence of MSO with FTAs.

Corollary 1. *Let C be the class of structures whose treewidth is bounded by some constant w. Then* #MC(C,MSO) *is solvable in fixed-parameter linear time, i.e. for each structure $\mathcal{A} \in C$ and formula $\varphi \in$ MSO it is solvable in time $\mathcal{O}(f(\|\varphi\|, w) \cdot \|\mathcal{A}\|)$.*

4 Enumeration Problems

We now show how the extended datalog programs from Theorem 1 can be used to solve also MSO-definable enumeration problems. Similarly as in Section 3, we start with a formal definition of the enumeration variant of model checking (MC).

#ENUM-MC(C,Φ)
Instance: A structure $\mathcal{A} \in C$ and a formula $\varphi \in \Phi$.
Parameter: $\|\varphi\|$.
Problem: Compute $\varphi(\mathcal{A})$.

As before, $\varphi(\mathcal{A})$ denotes the set of possible assignments to the free variables in φ that make φ true in $\mathcal{A}$. Again, we only consider the case that $\Phi = \text{MSO}$ and C is a class of structures with bounded treewidth. By *MSO-definable enumeration problems* we mean #ENUM-MC(C,MSO) problems for formulae φ without free first-order variables.

Let $\varphi(\boldsymbol{X})$ be an MSO-formula, Π be the extended datalog program from Theorem 1 to solve the counting problem defined by $\varphi(\boldsymbol{X})$, $\mathcal{A}$ be a σ-structure, and $\mathcal{A}_{td}$ denote the σ_{td}-structure representing both $\mathcal{A}$ and a tree decomposition $\mathcal{T}$ of $\mathcal{A}$. We cannot expect to use Π directly to compute the solutions of $\varphi(\boldsymbol{X})$. This would mean computing tuples of sets, which clearly surpasses the expressive power of datalog. However, we can use the facts in the LFP of $\Pi \cup \mathcal{A}_{td}$ (i.e., the intermediate results of our counting algorithm) and generate all solutions of $\varphi(\boldsymbol{X})$ in a postprocessing step. In Figure 1, we present the program enumerateSolutions, which is designed for precisely this purpose.

Our enumeration algorithm thus starts at the root node of $\mathcal{T}$ rather than at the leaves. Note that this is exactly what we have to do, since the root is the only node where types ϑ which correspond to some solutions of $\varphi(\boldsymbol{X})$ are identified. More precisely, our algorithm iterates over all types ϑ, s.t. the LFP of $\Pi \cup \mathcal{A}_{td}$ contains a fact $aux_{root}(\vartheta, j)$. By construction, these types correspond to the solutions of $\varphi(\boldsymbol{X})$. More formally, let $\boldsymbol{a}$ be the tuple of elements in the bag at the root of $\mathcal{T}$ and let $\boldsymbol{A}$ be a tuple of sets. Then the k-type of $(\mathcal{A}, \boldsymbol{a}, \boldsymbol{A})$ is some ϑ, s.t. $aux_{root}(\vartheta, \_)$ is in the LFP of $\Pi \cup \mathcal{A}_{td}$ iff $\mathcal{A} \models \varphi(\boldsymbol{A})$.

We can then exploit further facts from the LFP of $\Pi \cup \mathcal{A}_{td}$ to construct the actual solutions. Hereby, we collect for all nodes in $\mathcal{T}$ the types which *contributed* to the currently processed ϑ. We do so by constructing a new tree Υ for each such ϑ, which is stored in an internal data structure and described in detail below. Function getSol$(\cdot)$ in Figure 1 then traverses Υ several times until all solutions $\boldsymbol{A}$ of $\varphi(\boldsymbol{X})$ corresponding to ϑ are found. Below, we give a more detailed description of this procedure and all the auxiliary procedures used inside it.

Let us first describe the construction of the tree Υ for a type ϑ. This is done in the auxiliary procedure initTree(ϑ) of enumerateSolutions. In fact, Υ has as root node (n, ϑ), where n is the root node of tree decomposition $\mathcal{T}$. We then recursively add to a node (n', ϑ') in Υ a child node (n'', ϑ''), if $child_1(n'', n')$ and $aux_\lambda(n', \vartheta', \vartheta'', \ldots)$ (for $\lambda \in \{P, R, I\}$) are in the LFP of $\Pi \cup \mathcal{A}_{td}$. In case n' is a (B) node, we analogously add nodes using $aux_B(n', \vartheta', \vartheta_1, \vartheta_2, \_)$ but distinguish between left and right children. We store the children of a node as an ordered list of pointers (for (B) nodes, we have two such lists) together with a mark which sits on exactly one pointer in each such list. initTree(ϑ) initializes all marks to the first pointer in the respective lists. We will see below how marks are moved to obtain a new solution in each call of getSol$(\cdot)$ from enumerateSolutions.

We now describe function getSol$(\cdot)$ (ignoring the flag *isLast* and the function hasNextChild$(\cdot)$ which both are described below). Roughly speaking, getSol$(\cdot)$ traverses Υ following the marked pointers using getChildren(n, ϑ) which yields the result of the pointer marked in the list for the current node (for (B) nodes, this method accordingly returns two nodes, one for each of the two child nodes in the tree decomposition). This way, we go down the tree Υ by recursive calls of getSol$(\cdot)$ until the leaves are reached. In the leaves, we start with an empty solution and update it on the way back to the root. The only modifications are done when we are at an (EI) node or at a (B) node.

```
Program enumerateSolutions
begin
  take node n, s.t. root(n)
  for each ϑ with aux_root(ϑ, _) do
    initTree(ϑ)
    repeat
      (X, isLast) = getSol(n, ϑ, true)
      output X
    until isLast
  done
end

Function hasNextChild(n, ϑ)
input: node n, type ϑ
return: boolean isLast
begin
  l = number of children of (n, ϑ)
  if (n, ϑ).mark < l − 1 then
    (n, ϑ).mark++
    return true
  endif
  (n, ϑ).mark = 0
  return false
end

Function getSol(n, ϑ, isLast)
input: node n, type ϑ, boolean isLast
return: tuple of sets X, boolean isLast
begin
  if n is (L) node then
    X = (∅, ..., ∅)
  elseif n is (B) node then
    ((n', ϑ'), (n'', ϑ'')) = getChildren(n, ϑ)
    (X', isLast) = getSol(n', ϑ', isLast)
    (X'', isLast) = getSol(n'', ϑ'', isLast)
    X = X' ∪ X''
  elseif n is (EI) node then
    (n', ϑ') = getChildren(n, ϑ)
    (X', isLast) = getSol(n', ϑ', isLast)
    X = addElement(n, ϑ, X')
  else /* (ER) or (P) node */
    (n', ϑ') = getChildren(n, ϑ)
    (X, isLast) = getSol(n', ϑ', isLast)
  endif
  if isLast then
    isLast = hasNextChild(n, ϑ)
  endif
  return (X, isLast)
end
```

Fig. 1. Program enumerateSolutions

For (EI) nodes, addElement$(n, \vartheta, \boldsymbol{X}')$ alters $\boldsymbol{X}'$ according to ε in $aux_I(n, \vartheta, \vartheta', \varepsilon, j)$. For (B) nodes, $\boldsymbol{X}$ is simply the componentwise union of the respective results $\boldsymbol{X}'$, $\boldsymbol{X}''$.

Finally, we describe how the flag $isLast$ works and how the marks on pointers are moved when traversing the tree (this is done in function hasNextChild(n, ϑ)). Flag $isLast$ plays a kind of dual role. Being a parameter, it indicates whether we are currently allowed to move the mark, and this flag is passed down the recursive calls (note that the call from enumerateSolutions always sets this flag to 1, but this is not necessarily the case when going down the second subtree in a (B) node). Being a return value, flag $isLast$ indicates whether all possible positions of marks have been run through in the processed subtree. If this is the case, we may also move the mark of the current node; otherwise we do not touch it. Function hasNextChild(n, ϑ) takes care of the required manipulation of marks: it moves the mark to the next pointer, or in case the mark was already on the last pointer, it moves the mark back to the first pointer of the list; in both cases, the function returns the pointer which is now marked; however, in the former case, it returns as second value $false$, in the latter case (the mark is reset to the first pointer), it returns $true$. Thus, if $true$ is returned to enumerateSolutions we are done, since all possible positions of marks have been run through. Therefore, this algorithm guarantees that (i) each solution leading to the current ϑ in enumerateSolutions is found; (ii) no such solution is returned twice to enumerateSolutions.

Theorem 3. *Let σ and $w \geq 1$ be arbitrary but fixed. For the class C of σ-structures of treewidth w, the solutions of problem* #ENUM-MC(C,MSO) *for input $\mathcal{A} \in C$ and $\varphi \in$ MSO can be enumerated with delay $\mathcal{O}(f(\|\varphi\|, w) \cdot \|\mathcal{A}\|)$, where f is a function that only depends on w and $|\varphi|$.*

Proof. We call the bound $\mathcal{O}(f(\|\varphi\|, w) \cdot \|\mathcal{A}\|)$ fixed-parameter linear (FPL). Since the evaluation of the program Π over $\mathcal{A}$ in the proof of Theorem 1 is FPL, the number of atoms in the LFP of Π over $\mathcal{A}$ is also FPL. Therefore the creation and the size of Υ is FPL. Moreover, a traversal of Υ can also be done in FPL, and enumerateSolutions outputs the solutions with a delay in FPL. □

Program Π_{3-C} (#3-COLORABILITY)
/* (L) node */
$solve(n, \emptyset, \emptyset, \emptyset, 1) \leftarrow leaf(s)$.
/* (EI) node */
$forbidden(n, Y) \leftarrow bag(n, X), Y \subseteq X, edge(u, v), u \in Y, v \in Y$.
$allowed(n, Y) \leftarrow$ not $forbidden(n, Y)$.
$aux_I(n, R \uplus \{v\}, G, B, R, G, B, j) \leftarrow bag(n, X \uplus \{v\}), child_1(n_1, n), bag(n_1, X),$
$solve(n_1, R, G, B, j), allowed(n, R \uplus \{v\})$.
$aux_I(n, R, G \uplus \{v\}, B, R, G, B, j) \leftarrow bag(n, X \uplus \{v\}), child_1(n_1, n), bag(n_1, X),$
$solve(n_1, R, G, B, j), allowed(n, G \uplus \{v\})$.
$aux_I(n, R, G, B \uplus \{v\}, R, G, B, j) \leftarrow bag(n, X \uplus \{v\}), child_1(n_1, n), bag(n_1, X),$
$solve(n_1, R, G, B, j), allowed(n, B \uplus \{v\})$.
$solve(n, R, G, B, j) \leftarrow aux_I(n, R, G, B, \_, \_, \_, j)$.
/* (ER) node */
$aux_R(n, R, G, B, R \uplus \{v\}, G, B, j) \leftarrow bag(n, X), child_1(n_1, n), bag(n_1, X \uplus \{v\}),$
$solve(n_1, R \uplus \{v\}, G, B, j)$.
$aux_R(n, R, G, B, R, G \uplus \{v\}, B, j) \leftarrow bag(n, X), child_1(n_1, n), bag(n_1, X \uplus \{v\}),$
$solve(n_1, R, G \uplus \{v\}, B, j)$.
$aux_R(n, R, G, B, R, G, B \uplus \{v\}, j) \leftarrow bag(n, X), child_1(n_1, n), bag(n_1, X \uplus \{v\}),$
$solve(n_1, R, G, B \uplus \{v\}, j)$.
$solve(n, R, G, B, \mathrm{SUM}(j)) \leftarrow aux_R(n, R, G, B, \_, \_, \_, j)$.
/* (B) node */
$aux_B(n, R, G, B, R, G, B, j_1 * j_2) \leftarrow bag(n, X), child_1(n_1, n), child_2(n_2, n), bag(n_1, X),$
$bag(n_2, X), solve(n_1, R, G, B, j_1), solve(n_2, R, G, B, j_2)$.
$solve(n, R, G, B, j) \leftarrow aux_B(n, R, G, B, \_, \_, \_, j)$.
/* result (at the root node) */
$count(\mathrm{SUM}(j)) \leftarrow root(n), solve(n, \_, \_, \_, j)$.

Fig. 2. Counting 3-colorings

5 3-Colorability

In this section, we illustrate our approach with the aid of the counting and enumeration variant of 3-COLORABILITY, whose MSO-encoding $\varphi(R, G, B)$ was given in Section 3. A graph (V, E) with vertices V and edges E is represented as a σ-structure $\mathcal{A}$ with $\sigma = \{edge\}$. By $\mathcal{A}_{td}$ we denote the σ_{td}-structure which, in addition to the graph,

also represents a tree decomposition $\mathcal{T}$ of $\mathcal{A}$ of width $\leq w$ for some constant w. We write $\mathcal{C}(\mathcal{A})$ to denote the set of valid 3-colorings of $\mathcal{A}$, i.e., $\mathcal{C}(\mathcal{A}) = \{(\bar{R}, \bar{G}, \bar{B}) \mid \mathcal{A} \models \varphi(\bar{R}, \bar{G}, \bar{B})\}$. The program in Figure 2 takes a σ_{td}-structure $\mathcal{A}_{td}$ as input and calculates $|\mathcal{C}(\mathcal{A})|$.

Note that the set variables used in Figure 2 are not sets in the general sense, since their cardinality is restricted by the size $w+1$ of the bags, where w is a fixed constant. Hence, these "fixed-size" sets can be simply implemented by means of k-tuples over $\{0, 1\}$ with $k \leq (w + 1)$. For the sake of readability, we also use the non-datalog operator $\uplus$ (disjoint union), which could be easily replaced by a "proper" datalog expression. By considering the bags as sets, we no longer need the node type (P).

At the heart of this program is the intensional predicate $solve(n, R, G, B, j)$ with the following intended meaning: n denotes a node in $\mathcal{T}$, the sets R, G, B are the projections of some coloring on $\mathcal{A}_n$ onto $\mathrm{bag}(n)$ and j denotes the number of different colorings on $\mathcal{A}_n$ having this projection. More precisely, let $\Lambda(\mathcal{A}_n, R, G, B) = \{(\bar{R}, \bar{G}, \bar{B}) \in \mathcal{C}(\mathcal{A}_n)$ with projection (R, G, B) onto $\mathrm{bag}(n)\}$, then a fact $solve(n, R, G, B, j)$ shall be in the LFP of $\Pi_{3-C} \cup \mathcal{A}_{td}$, iff $|\Lambda(\mathcal{A}_n, R, G, B)| = j \geq 1$.

The main task of the program is the computation of all facts $solve(n, R, G, B, j)$ via a bottom-up traversal of the tree decomposition. The predicate *count* holds the final result. Π_{3-C} solves the #3-COLORABILITY problem in the following way:

Theorem 4. *Given an instance of* #3-COLORABILITY*, i.e., an* $\{edge\}$*-structure* $\mathcal{A}$*, together with a tree decomposition* $\mathcal{T}$ *of* $\mathcal{A}$ *having width* w*, then* $count(j)$ *with* $j \geq 1$ *is in the LFP of* $\Pi_{3-C} \cup \mathcal{A}_{td}$ *iff* $\mathcal{A}$ *has exactly* j *possible 3-colorings. Moreover, both the construction of* $\mathcal{A}_{td}$ *and the evaluation of* $\Pi_{3-C} \cup \mathcal{A}_{td}$ *can be computed in* $\mathcal{O}(3^{2w+2}) \cdot \|\mathcal{A}\|$*, assuming unit cost for arithmetic operations.*

Proof Idea. The *correctness* of the program Π_{3-C} follows immediately as soon as it has been shown that $solve(n, R, G, B, j)$ indeed has the intended meaning described above. This in turn can be easily shown by structural induction over the tree decomposition $\mathcal{T}$ via a case distinction for the possible node types (L), (EI), (ER), and (B) of n.

For the *complexity*, we notice that program Π_{3-C} is essentially a succinct representation of a quasi-guarded extended datalog program. For instance, in the atom $solve(n, R, G, B, j)$, the sets R, G, and B are subsets of size $\leq w$ of bag A_n at node n. Hence, each combination R, G, B could be represented by three sets $r, s, t \subseteq \{0, \ldots, w\}$ referring to indices of elements in A_n. Hence, $solve(n, R, G, B, j)$ is a succinct representation of constantly many predicates of the form $solve_{r,s,t}(n, j)$. Then $bag(n, X)$ is a quasi-guard in each rule. The *fixed-parameter linearity* follows from Theorem 2. A finer analysis of the program reveals, that for #3-COLORABILITY the function $f(\|\varphi\|, w)$ can be explicitly stated as 3^{2w+2}, since there are at most 3^{w+1} partitions R, G, B at each node n. Hence, the combination of two partitions as the argument of predicates (e.g., $aux_I(n, R, G, B, R', G', B', j)$) yield at most 3^{2w+2} groundings of these predicates. □

The enumeration variant of 3-COLORABILITY can be solved with linear delay by a slight modification of the program in Figure 1. Instead of nodes (n, ϑ) in Υ, we consider nodes of the form (n, R, G, B). Then we again use the $child(\cdot)$ and $aux_\lambda(\cdot)$ facts in the LFP of $\Pi_{3-C} \cup \mathcal{A}_{td}$ to establish a parent-child relation between nodes (n, R, G, B) and (n', R', G', B'). Analogously to the program in Figure 1, each solution $\bar{R}, \bar{G}, \bar{B}$ is

constructed in a bottom-up way, starting with an empty solution at the leaves. When we reach an (EI) node n, the set difference between R, G, B and R', G', B' of a fact $aux_I(n, R, G, B, R', G', B', j)$ determines how the sets $\bar{R}, \bar{G}, \bar{B}$ are extended.

Discussion. The generic construction of a program Π from some MSO-formula φ in the proof of Theorem 1 is "constructive" in theory but not feasible in practice. However, in contrast to the MSO-to-FTA approach, datalog allows us to construct tailor-made programs like program Π_{3-C}, which follow the intuition of the generic programs Π but incorporates several short-cuts that make it indeed feasible: Above all, as the intended meaning of the *solve*-predicate suggests, we only propagate those "types" (represented by the *solve*-facts) which can possibly be extended in bottom-up direction to a solution. Moreover, the *solve*-facts do not exactly correspond to the types in Theorem 1 but only describe the properties of each type which are crucial for the target formula $\varphi(R, G, B)$.

6 Conclusion

We have extended the monadic datalog approach [14] to MSO-definable counting and enumeration problems. For the latter, we have thus shown that they can be solved with linear delay in case of bounded treewidth. We have also illustrated the potential of our approach for constructing efficient algorithms by solving the counting and enumeration variant of 3-COLORABILITY. As future work, we plan to apply our approach to further MSO-definable problems. Finally, we also want to tackle extensions of MSO with our approach; in particular extensions by optimization (i.e., counting/enumerating the *minimal* or the *maximal* solutions only) [2] and by local cardinality constraints [22].

References

1. Afrati, F.N., Chirkova, R.: Selecting and using views to compute aggregate queries. In: Eiter, T., Libkin, L. (eds.) ICDT 2005. LNCS, vol. 3363, pp. 383–397. Springer, Heidelberg (2004)
2. Arnborg, S., Lagergren, J., Seese, D.: Easy problems for tree-decomposable graphs. Journal of Algorithms 12(2), 308–340 (1991)
3. Bagan, G.: MSO queries on tree decomposable structures are computable with linear delay. In: Ésik, Z. (ed.) CSL 2006. LNCS, vol. 4207, pp. 167–181. Springer, Heidelberg (2006)
4. Bodlaender, H.L.: A linear-time algorithm for finding tree-decompositions of small treewidth. SIAM J. Comput. 25(6), 1305–1317 (1996)
5. Ceri, S., Gottlob, G., Tanca, L.: Logic Programming and Databases. Springer, Heidelberg (1990)
6. Cohen, S., Nutt, W., Serebrenik, A.: Rewriting aggregate queries using views. In: Proc. PODS 1999, pp. 155–166. ACM, New York (1999)
7. Courcelle, B.: Graph rewriting: An algebraic and logic approach. In: Handbook of Theoretical Computer Science, vol. B, pp. 193–242. Elsevier Science Publishers, Amsterdam (1990)
8. Courcelle, B.: Linear delay enumeration and monadic second-order logic. Discrete Applied Mathematics 157(12), 2675–2700 (2009)
9. Downey, R.G., Fellows, M.R.: Parameterized Complexity. Springer, New York (1999)
10. Eiter, T., Faber, W., Fink, M., Woltran, S.: Complexity results for answer set programming with bounded predicate arities and implications. Annals of Mathematics and Artificial Intelligence 51(2-4), 123–165 (2007)

11. Flum, J., Frick, M., Grohe, M.: Query evaluation via tree-decompositions. Journal of the ACM 49(6), 716–752 (2002)
12. Flum, J., Grohe, M.: Parameterized Complexity Theory. Springer, Heidelberg (2006)
13. Frick, M., Grohe, M.: The complexity of first-order and monadic second-order logic revisited. In: Proc. LICS 2002, pp. 215–224 (2002)
14. Gottlob, G., Pichler, R., Wei, F.: Monadic datalog over finite structures with bounded treewidth. In: Proc. PODS 2007, pp. 165–174. ACM, New York (2007)
15. Grohe, M.: Descriptive and parameterized complexity. In: Flum, J., Rodríguez-Artalejo, M. (eds.) CSL 1999. LNCS, vol. 1683, pp. 14–31. Springer, Heidelberg (1999)
16. Grumbach, S., Rafanelli, M., Tininini, L.: On the equivalence and rewriting of aggregate queries. Acta Inf. 40(8), 529–584 (2004)
17. Jakl, M., Pichler, R., Rümmele, S., Woltran, S.: Fast counting with bounded treewidth. In: Cervesato, I., Veith, H., Voronkov, A. (eds.) LPAR 2008. LNCS (LNAI), vol. 5330, pp. 436–450. Springer, Heidelberg (2008)
18. Kemp, D.B., Stuckey, P.J.: Semantics of logic programs with aggregates. In: Proc. ISLP, pp. 387–401 (1991)
19. Kloks, T.: Treewidth, Computations and Approximations. LNCS, vol. 842. Springer, Heidelberg (1994)
20. Klug, A.C.: Equivalence of relational algebra and relational calculus query languages having aggregate functions. J. ACM 29(3), 699–717 (1982)
21. Libkin, L.: Elements of Finite Model Theory. Springer, Heidelberg (2004)
22. Szeider, S.: Monadic second order logic on graphs with local cardinality constraints. In: Ochmański, E., Tyszkiewicz, J. (eds.) MFCS 2008. LNCS, vol. 5162, pp. 601–612. Springer, Heidelberg (2008)
23. Vardi, M.Y.: The complexity of relational query languages (extended abstract). In: Proc. STOC 1982, pp. 137–146. ACM, New York (1982)

The Nullness Analyser of JULIA

Fausto Spoto

Dipartimento di Informatica, Università di Verona, Italy
fausto.spoto@univr.it

Abstract. This experimental paper describes the implementation and evaluation of a static nullness analyser for single-threaded Java and Java bytecode programs, built inside the JULIA tool. Nullness analysis determines, at compile-time, those program points where the `null` value might be dereferenced, leading to a run-time exception. In order to improve the quality of software, it is important to prove that such situation does not occur. Our analyser is based on a denotational abstract interpretation of Java bytecode through Boolean logical formulas, strengthened with a set of denotational and constraint-based supporting analyses for locally non-`null` fields and *full* arrays and collections. The complete integration of all such analyses results in a correct system of very high precision whose time of analysis remains in the order of minutes, as we show with some examples of analysis of large software.

1 Introduction

Software is everywhere nowadays. From computers, it has subsequently been embedded in phones, home appliances, industries, aircraft, nuclear power stations, with more applications coming every day. As a consequence, its complexity is increasing and bugs spread with the software itself. While bugs are harmless in some situations, in others they have economical, human or civil consequences. Therefore, software verification is increasingly recognised as an important aspect of software technology and consumes larger and larger portions of the budget of software development houses.

Software verification aims at proving software error-free. The notion of *error* is in general very large, spanning from actual run-time errors to bad programming practices to badly-devised visual interfaces. Techniques for software verification are also manifold. It is generally accepted that the programming language of choice affects the quality of software: languages with strong static (compile-time) checks, simple syntax and simple semantics reduce the risk of errors. Good programming practices are also a key element of software quality. The reuse of trusted libraries is another. Nevertheless, bugs keep being present in modern software.

Hence, the big step in software verification should be automatic software verification tools, able to find, automatically and in reasonable time, the bugs in a program. Of course, no tool can be both correct (finding *all* bugs) and complete (finding *only* bugs), because of the well-known undecidability property of

E.M. Clarke and A. Voronkov (Eds.): LPAR-16, LNAI 6355, pp. 405–424, 2010.

software. But a tool can well be useful if it approximates the set of bugs in a precise way, as a set of *warnings*, yet leaving to the programmer the burden of determining which warnings are real bugs.

In this paper we describe our implementation of a software verifier for `null`-pointer analysis, also called *nullness* analysis, embedded inside the JULIA tool, carried out in the last three years. We show the structure of the implementation, its precision and strengths. Its goal is to spot program points in Java and Java bytecode where a `null`-pointer exception might be raised at run-time, because the value `null` is dereferenced during the access to a field, a call to a method or during synchronisation. Although JULIA works over bytecode, it can be applied to Java source code by compiling it into bytecode and we only report examples, here, over Java source code, more easily understood by the reader.

It is important to stress that we do not claim the absolute superiority of our implementation *w.r.t.* other tools for automatic software verification of Java. We only want to highlight the specific features of our tool that make it relevant in the context of software verification, possibly in coexistence with other tools. Namely, a software verification tool can be judged *w.r.t.* many orthogonal aspects, including:

correctness/completeness. A correct verifier has the advantage of definitely excluding the presence of bugs outside the list of warnings provided to the user. Nevertheless, it is understandable that some tools have decided to sacrifice correctness (and completeness), since this lets them shrink the list of warnings to the most realistic ones: a long list of warnings can only scare the user, who will not use the tool anymore. Our choice is to stay correct and restrict the list of warnings as much as possible with the help of the most advanced techniques for static analysis;

annotations/no annotations. A verifier using annotations requires the programmer to decorate the source code with invariants that can be exploited but must also be checked by the tool. Examples are method pre- and post-conditions and loop invariants. This approach burdens the programmer with an extra task, but it obliges him to reason on what he writes and clean and document the code. Moreover, a tool exploiting annotations can be extremely precise and its analysis can be modular, that is, parts of the program can be changed without having to verify everything again. Our choice has been not to use annotations;

genericity. A generic verifier performs many kinds of software verifications and is consequently more useful. Nevertheless, an implementation centered around a given verification problem can be more focused on its specific target and more optimised. Our choice is to have a general analyser (JULIA), but our limited time has been used to build advanced instantiations for nullness verification and termination analysis [20] only;

real-time/off-line. A real-time verifier runs every time the source code is modified and provides immediate feedback to the programmer. It is typically modular*, i.e.,* it performs local reasonings on each method and requires source code annotations. Sometimes (but not always) some incorrect assumptions

are made, such as assuming that methods are always called with non-`null` arguments that do not share any data structure, which is not always true. Real-time verifiers are very effective during software development. Off-line verifiers require larger resources and are consequently used at fixed milestones during software development. But they can afford the most precise verification techniques and can provide a correct and thorough report on the analysed program. Our choice is that of an off-line analyser, although its run-times are kept in the order of minutes on a standard computer.

Many different tools provide some form of nullness analysis. The ECLIPSE IDE [1] provides a real-time, very limited, incorrect, largely imprecise nullness analysis, as a plug-in. ESC/JAVA2 [4] is a generic tool, evolved from ESC/JAVA [8], that uses theorem-proving to perform an off-line nullness analysis, using annotations provided by the programmer. The tool can also work without annotations, but then the number of warnings for `null`-pointer dereferences becomes very large [15]. It is possible to infer typical nullness annotations with a tool called HOUDINI [7], which calls ESC/JAVA to validate or refuse the annotations. It does not work with ESC/JAVA2 and the latest version is from 2001. Since then, Java has largely changed. We did not manage to build HOUDINI and ESC/JAVA on our Linux machine. FINDBUGS [10] is another generic tool that performs an only syntactical off-line nullness analysis, in general incorrect (see later for an example). Nevertheless, this tool is considered very effective at spotting typical erroneous programming patterns where `null`-pointer bugs occur. JLINT [2] performs a simple, only syntactical analysis of the code to spot some possible `null`-pointer dereferences of variables. It did not compile on our Linux machine, so we could not experiment with it. According to the `README` file, it is updated to Java version 1.4 only. DAIKON [6] is a tool that infers *likely* nullness annotations, but there is no guarantee of their correctness. A comparison of those tools, beyond the case of nullness analysis, is presented in [15]. It must be stated that they do not only verify `null`-pointer dereferences, but also other properties of Java programs. NIT [11], instead, is a tool explicitly targeted at nullness analysis. It performs a provably correct, fast off-line nullness analysis of Java bytecode programs. This tool is the most similar to JULIA, since both are based on semantical static analysis through abstract interpretation [5] of Java bytecode. It is faster than JULIA, but this comes at the price of precision, since its latest version $0.5d$ is almost as precise as an old version of JULIA (see the experiments in [18]) that used only the techniques up to Subsection 3.2 of this paper.

The contribution of this paper is the presentation of the structure of the nullness analysis implemented inside JULIA, together with a brief overview of the techniques that it exploits, partially based on logical formulas. As such, it is an example of implementation of logical systems for static analysis, with strong correctness guarantees. Our goal has been precision and correctness, which entails that JULIA will not be so fast as other tools, although it is already able to analyse around 5000 methods in a few minutes on standard hardware.

The rest of the paper is organised as follows. Section 2 yields an introductory example of nullness analysis with JULIA, FINDBUGS and NIT. Section 3

describes the structure of the nullness analysis of JULIA and how each component contributes to the precision of the overall result. Section 4 describes how JULIA creates an annotation file collecting nullness information, that can later be used by other tools. Section 5 concludes the paper. The theoretical results that form the basis of our implementation have already been published elsewhere. In particular, [17] reports a formal description and proofs of correctness for the techniques of Subsections 3.1 and 3.2; [18] includes the same for Subsection 3.3 also; [19] reports definitions and proofs for the rawness analysis of Section 4 and formalises the constraint-based techniques used in JULIA. The new analyses for arrays and collections in Subsections 3.4 and 3.5 have never been published before.

2 An Introductory Example

In order to show the kind of precision that can be expected from our nullness analysis, consider the Java program in Figure 1. Methods `equals()` and `hashCode()` are reachable since they are called, indirectly, by the calls at lines 30 and 32 to the library method `java.util.HashSet.add()`. The analysis of that program through JULIA does not signal any warning. According to the correctness guarantee of JULIA, this means that the program will never dereference the `null` value at run-time. The analysis of the same program with NIT yields 8 warnings:

```
line 14: unsafe call to bool equals(Object)
line 16: unsafe call to bool equals(Object)
line 19: unsafe call to int hashCode()
line 24: unsafe call to String replace(char,char)
line 26: unsafe call to String replace(char,char)
line 31: unsafe field read of C inner
line 34: unsafe call to void println(String)
line 35: unsafe call to void println(String)
```

They are *false alarms*, since:

- at lines 14 and 19, field `name` is always non-`null` in the objects of class `C` (look at lines 24 and 26 and consider that the return value of `String.replace()` is always non-`null`). Hence the call to `equals()` does not dereference `null`;
- at line 16, the non-nullness of field `inner` has been already checked at line 15 (Java implements a short-circuit semantics for `||`);
- at lines 24 and 26, `args[i]` is non-`null`, since the Java Virtual Machine passes to `main()` an array `args` containing non-`null` values only, and there is no other call to `main()` in the program;
- at line 31, variable `t` cannot hold `null` since it iterates over the elements of the array `ts` (line 29), which is *filled* with non-`null` values by the loop at lines 23 – 27;
- at lines 34 and 35, the static field `System.out` cannot hold `null` since it is initialised to a non-`null` value by the Java Virtual Machine and is not modified after by the program in Figure 1 (it never calls `System.setOut()` with a possibly `null` argument).

```
0:  import java.util.*;
1:  public class C {
2:    private String name;
3:    private C inner;
4:    public C(String name, C inner) {
5:      this.name = name;
6:      this.inner = inner;
7:    }
8:    public String toString() {
9:      if (inner != null) return "[" + inner + "]";
10:     else return "[]";
11:   }
12:   public boolean equals(Object other) {
13:     return other instanceof C &&
14:       ((C) other).name.equals(name) &&
15:       (((C) other).inner == null ||
16:          ((C) other).inner.equals(inner));
17:   }
18:   public int hashCode() {
19:     return name.hashCode();
20:   }
21:   public static void main(String[] args) {
22:     C[] ts = new C[args.length];
23:     for (int i = 0; i < ts.length; i++) {
24:       ts[i] = new C(args[i].replace('\\','/'), null);
25:       if (i < ts.length - 1)
26:         ts[++i] = new C(args[i].replace('\\','/'), ts[i - 1]);
27:     }
28:     Set<C> s1 = new HashSet<C>(), s2 = new HashSet<C>();
29:     for (C t: ts) {
30:       s1.add(t);
31:       if (t.inner != null)
32:         s2.add(t.inner);
33:     }
34:     for (C t: s1) System.out.println(t.toString());
35:     for (C t: s2) System.out.println(t.toString());
36:   }
37: }
```

Fig. 1. A Java program exposing interesting considerations about nullness

The analysis of the same program with FINDBUGS yields no warnings. However, this result is the consequence of some optimistic (and in general incorrect) hypotheses assumed by FINDBUGS on the behaviour of the program under analysis: it cannot be taken as a proof of the fact that the program in Figure 1 never dereferences `null`. Namely, assume to comment out lines 15 and 31 in Figure 1. The program contains two bugs now: at line 16, method `equals()` is called on a possibly `null` field `inner`; at line 35, method `toString()` is called on a possibly

`null` variable `t`, since the set `s2` might contain `null` now, introduced at line 32. JULIA correctly spots these situations and issues two warnings:

```
line 16: call with possibly-null receiver to C.equals(Object)
line 35: call with possibly-null receiver to C.toString()
```

However, FINDBUGS keeps signalling no warning at all and hence is not able to find those two bugs and is incorrect. NIT keeps signalling the same 8 warnings seen above. It somehow misses the bug at line 35, since, there, no warning is signalled for the incorrect call to `toString()` but only a false alarm about the call to `println()`. However, this must be just a bug in the implementation of NIT, that might be corrected in next releases, since the underlying theory has been certified in COQ to be correct.

3 Nullness Analysis in JULIA

We describe here the phases of the nullness analyser implemented inside the JULIA analysis tool and their contribution to the overall precision. We had to use many techniques in order to achieve a very high level of precision. JULIA currently performs semantical static analyses based on denotational abstract interpretation and on constraint-based abstract interpretation. Both come in different flavors here, for inferring properties of local variables, fields, arrays of references and collections. We test each technique with all the previous techniques turned on as well, so we will see progressively increasing times and precision.

Before the actual nullness analysis starts, JULIA must of course load and parse the `.class` or `.jar` files containing the analysed Java bytecode application. Moreover, it must extract the *control flow* of the program, linking method calls with the method implementations that they actually call. We perform this through a nowadays traditional *class analysis* [13]. Type inference for local variables and Java bytecode stack elements is performed as in the official documentation [12]. These aspects of JULIA are completely independent from the actual analysis which is later performed, nullness, termination or other.

3.1 Nullness Analysis of Local Variables

Given a program point p, the number of local variables accessible at p is finite, although arbitrarily large. In particular, it is possible to access all and only the local variables declared by the method before p, which include the formal parameters of the method. This entails that it is possible to build a finite constraint expressing the nullness behaviour of the variables at p *w.r.t.* the nullness behavior of the variables at a subsequent program point p'. In [17], this constraint is defined as a Boolean logical formula whose models include all possible nullness behaviours for the local variables.

Assume for instance that p is line 22 in Figure 1. The only variable in scope at p is `args`, since `ts` is not yet declared at p. Program p' is the logically subsequent statement in that program, *i.e.,* the assignment `int i = 0` at line 23. At p',

variables `args` and `ts` are declared. The Boolean formula built by JULIA to relate the nullness of the variables at p to that of the nullness at p' uses Boolean variables of two forms: $\check{v}$ stands for the value of variable v at p; $\hat{v}$ stands for its value just after p, that is, at p'. A special variable e is used to represent exceptional states. Namely, that formula is:

$$\neg\check{e} \wedge (\neg\hat{e} \rightarrow (\neg\hat{\texttt{args}} \wedge \neg\hat{\texttt{ts}})) \tag{1}$$

meaning that, if the assignment at line 22 is executed, then no exception must be raised immediately before p ($\neg\check{e}$); moreover, if no exception is raised by the assignment ($\neg\hat{e}$) then both `args` and `ts` are non-`null` at p' ($\neg\hat{\texttt{args}} \wedge \neg\hat{\texttt{ts}}$). In (1), program variables of reference type have been translated into Boolean variables of two kinds: *input* variables, such as $\check{\texttt{args}}$, stand for the nullness of the corresponding program variable at the beginning of p, while *output* variables, such as $\hat{\texttt{args}}$, stand for the nullness of the corresponding variable at the end of p, *i.e.,* at beginning of p'. The special variable e stands for an exceptional situation (in the sense of a raised Java exception, implicit or explicit).

Consider now the assignment `int i = 0` at line 23. For it JULIA builds the Boolean formula:

$$\neg\check{e} \wedge \neg\hat{e} \wedge (\check{\texttt{args}} \leftrightarrow \hat{\texttt{args}}) \wedge (\check{\texttt{ts}} \leftrightarrow \hat{\texttt{ts}}) \tag{2}$$

that is, if that assignment is executed then no exception must be raised just before it ($\neg\check{e}$); no exception is ever raised by that assignment ($\neg\hat{e}$); the nullness of `args` and `ts` is not affected by the assignment to `i` ($(\check{\texttt{args}} \leftrightarrow \hat{\texttt{args}}) \wedge (\check{\texttt{ts}} \leftrightarrow \hat{\texttt{ts}})$). Nothing is said about variable `i`, since it has primitive type so its value is abstracted away.

In order to analyse the sequential execution of more statements, JULIA performs the *abstract sequential execution* of Boolean formulas, each abstracting one of the statements. For instance, the abstract sequential execution of (1) and then (2) is computed by matching the output variables of (1) with the input variables of (2). That is, those variables are renamed into the same new, fresh variables, the resulting two formulas are conjuncted ($\wedge$) and the new fresh variables are projected away through an existential quantification. The result is

$$\neg\check{e} \wedge \neg\hat{e} \wedge \neg\hat{\texttt{args}} \wedge \neg\hat{\texttt{ts}} \tag{3}$$

The latter is an abstraction of the state at the beginning of the execution of the loop at line 23, after the initialisation of `i`. It clearly states that `ts` is non-`null` there.

These Boolean formulas can be built in a methodological way as a bottom-up abstraction of the code. Their construction is based on the abstract sequential composition of formulas but also on the logical disjunction of two formulas, to abstract conditionals and virtual methods calls with multiple target implementations. Method calls are abstracted by *plugging* the analysis (*denotation*) of their body in the point of call, so that the resulting analysis is inter-procedural and completely context-sensitive. The latter means that the approximation of the final state, after a method call, is not fixed, but depends on the approximation of

the input state before the same call. Loops and recursion are modelled through a fixpoint calculation. We have used no widening, since the abstract domain of Boolean formulas has finite height (at each program point, the number of local variables in scope is fixed and finite) and we have never experienced the need of accelerating the convergence of the fixpoint calculations.

The detailed formalisation of this analysis and its proof of correctness are done in [17] by using abstract interpretation. The analysis can be used to guarantee the absence of `null` dereferences in the code. For instance, Equation (3), together with a formula built for the body of the loop at lines $23-27$ and stating that the nullness of `ts` does not change inside the loop, is enough to conclude that the dereference `ts.length` at line 23 does not raise any `null`-pointer exception.

The implementation of this analysis is quite efficient since binary decision diagrams [3] are used to represent the Boolean formulas and no aliasing information must be computed before the analysis: the abstraction into Boolean formulas abstracts away the heap memory completely and only considers the activation records of the methods, where the local variables live. This means, for instance, that we do not have to bother that the call to the constructor of `C` at line 22 modifies the nullness of `args` since this is just impossible: in Java, the callee cannot modify the value of the local variables of the caller, but only, possibly, the fields and array elements reachable from those local variables and the static fields and everything reachable from them, which are not local variables. This simplicity comes at a price: the relative imprecision of the results. For instance, the analysis of the program in Figure 1 with JULIA tuned down to use this technique only yields the following set of warnings (false alarms):

```
line 14: call with possibly-null receiver to String.equals(Object)
line 16: call with possibly-null receiver to String.equals(Object)
line 19: call with possibly-null receiver to String.hashCode()
line 24: call with possibly-null receiver to String.replace(char,char)
line 26: call with possibly-null receiver to String.replace(char,char)
line 31: read with possibly-null receiver of field inner
line 34: call with possibly-null receiver to C.toString()
line 35: call with possibly-null receiver to C.toString()
```

We report below the precision of this analysis, as the amount *derefs* of dereferences that are proved *safe* in the analysed programs, *i.e.,* having a provably non-`null` receiver. The analysed programs do not contain unsafe dereferences, as we have verified by checking, manually, the warnings issued by the most precise analysis (Subsection 3.5). Then a very precise analyser could in principle find out that 100% of their dereferences are safe. The only exception is `EJE`, that contains three bugs, that is, three dereferences that can actually happen on `null` sometimes. For `EJE`, a very precise analyser could in principle find out that 99.89% of the dereferences are safe.

We also report below the number of safe dereferences restricted to some frequent examples, namely, those related to field accesses (*access*), field modifications (*update*) and method calls (*call*). In this and the following tables, we have analysed the programs by including the whole `java.*` hierarchy in the analysis,

which is reflected in the relatively high number of methods analysed. Fewer library methods might be included, but worst-case assumptions would then be made for them by the analyser, compromising the precision of the results. The time of the analysis, in seconds, includes that for parsing the class files of the application and of the libraries. `OurTunes` is an open source cross-platform file sharing client which allows users to connect to iTunes and share music files. `EJE` is a text editor. `JFlex` is a lexical analysers generator. `utilMDE` are Michael Ernst's supporting classes for DAIKON; they include test applications, that is what we analyse. The `Annotation File Utilities` (in the following just `AFU`) are tools that allow to apply or extract annotations into Java source code (see also Section 4). The experiments have been performed on a quad-core Intel Xeon computer running at 2.66Ghz, with 8 gigabytes of RAM and Linux 2.6.27.

program	methods	time	derefs	access	update	call	warnings
`OurTunes`	3036	24.49	86.48%	92.95%	99.64%	79.12%	425
`EJE`	3077	33.31	77.04%	99.43%	100.00%	57.37%	926
`JFlex`	3735	39.05	81.29%	76.79%	98.33%	81.71%	1254
`utilMDE`	3706	43.05	92.85%	93.68%	99.06%	88.42%	252
`Annotation File Utilities`	3625	39.48	91.09%	88.45%	99.66%	86.64%	523

For verification purposes, this first technique does not have a satisfying precision. Nevertheless, it is interesting for optimisation (removal of useless nullness tests), which is less fussy about precision. Moreover, it is correct for the analysis of multi-threaded applications, while the other techniques that we are going to describe give, in special situations, incorrect results on multi-threaded programs (their adaptation to multi-threaded applications is on its way).

3.2 Globally non-`null` Fields

The warning at line 14 in Subsection 3.1 is a consequence of the fact that method `equals()` is called on field `name` of variable `other` rather than on a variable. Hence the technique of that subsection cannot prove that that call does not raise any `null`-pointer exception. The same happens for the warning at line 19. To solve this problem, we need information on the nullness of the fields also, not just of the local variables of the program. Field definitions are finite in any Java program, but an unbound number of objects can be allocated by a program, thus allowing an unbound number of field instances. Hence, it is not possible to allocate a Boolean variable for each of those instances. Even considering field definitions only, and merging their instances in the same approximation, the number of field definitions is typically too high to be reflected in the same number of Boolean variables. In [17], we have solved this problem by labelling fields as *non-*`null` whenever they are always initialised by all the constructors of their defining class, are never accessed before that initialisation and the program only stores a non-`null` values inside them (in constructors or methods). If this is the case, we are sure that these non-`null` fields always hold a non-`null` value when they are accessed. Since their identification requires nullness information about the values that are written into them, this new nullness analysis is performed in an *oracle-based* way: it is first assumed that all fields that are always initialised

by all the constructors of their defining class, before being read, are non-`null`. That is, we use an initial *oracle* that contains all such fields. A first nullness analysis is computed as in Subsection 3.1, exploiting such (optimistic) hypothesis. Then some fields are discarded from the oracle as potentially `null` whenever the last nullness analysis cannot prove that only non-`null` values are written into them. Hence a new nullness analysis is performed and the oracle further shrunk. This process is repeated until the oracle does not shrink anymore. This oracle-based technique, proved correct in [17], will be exploited also in the subsequent subsections, since the extra analyses that we will introduce there, for extra precision, need nullness information themselves.

This technique identifies *globally* non-`null` fields, since they stay non-`null`, forever, after their initialisation. Not surprisingly, it is computationally more expensive than that in Subsection 3.1. This is not only a consequence of the repeated execution of the nullness analysis, which is tamed by using caches. The actual complication is that it needs some form of definite aliasing information in order to identify the non-`null` fields. Namely, in order to spot the fields of `this` that are initialised by each constructor, it is not sufficient to look for assignments to `this.field`, since many assignments may occur, indirectly, by writing inside `x.field`, where `x` is a definite alias of `this`. This is particularly the case in Java bytecode, where, typically, the stack contains aliases of local variables (such as `this`). Moreover, helper functions are frequently used to help constructors build the state of `this` and definite aliasing is needed to track the information flow from constructors to helper functions. To that purpose, one needs to prove that the call to the helper function happens on a definite alias of `this`.

In conclusion, the cost of this analysis is higher, as well as its precision, than that of the analysis in Subsection 3.1 alone. For instance, for the program in Figure 1, JULIA reports now only 6 of the 8 warnings reported in Subsection 3.1 (we write inside square brackets the warnings that have been removed):

```
[line 14: call with possibly-null receiver to String.equals(Object)]
line 16: call with possibly-null receiver to String.equals(Object)
[line 19: call with possibly-null receiver to String.hashCode()]
line 24: call with possibly-null receiver to String.replace(char,char)
line 26: call with possibly-null receiver to String.replace(char,char)
line 31: read with possibly-null receiver of field inner
line 34: call with possibly-null receiver to C.toString()
line 35: call with possibly-null receiver to C.toString()
```

The following table shows time and precision for the analysis of the same applications analysed in Subsection 3.1 (the number of analysed methods does not change *w.r.t.* what is reported in that subsection). It reports, for comparison, inside brackets, some numbers as they were in that subsection:

program	time	derefs	access	update	call	warnings
OurTunes	31.94 (was 24.49)	95.09% (was 86.48%)	97.65%	100.00%	90.90%	173 (was 425)
EJE	43.34 (was 33.31)	98.25% (was 77.04%)	99.85%	100.00%	96.78%	74 (was 926)
JFlex	56.89 (was 39.05)	90.35% (was 81.29%)	85.23%	98.95%	93.71%	750 (was 1254)
utilMDE	64.17 (was 43.05)	95.11% (was 92.85%)	94.03%	100.00%	92.20%	195 (was 252)
AFU	52.37 (was 39.48)	96.04% (was 91.09%)	96.97%	100.00%	94.19%	231 (was 523)

This technique yields almost the same results as NIT, as the experiments in [18] show. NIT is in general faster but only slightly less precise than this technique, possibly because the analysis in Subsection 3.1 is completely context-sensitive, which is not the case for NIT.

In general, this technique, as well as those of the next subsections, is not correct for multi-threaded programs. This is because, although it assumes a field of an object o to be non-`null` only when it is always initialised by all constructors of the class of o before being read and only assigned non-`null` values in the program, it is possible, according to the Java memory model, that its initialisation is not immediately visible to other threads than that creating o. Hence, those threads might find `null` in the field [9]. NIT incurs in the same problem, since its proof of correctness considers a simplified memory model, which is not that of multi-threaded Java.

3.3 Locally non-`null` Fields

Some fields are not globally non-`null`. Namely, there are fields that are not initialised by all the constructors of their defining class, or that are accessed before being initialised, or that are assigned `null` somewhere in the program. For them, programmers often test their non-nullness before actually accessing them, with programming patterns such as that at lines 15 and 16 in Figure 1, where method `equals()` is called on field `inner` of `other` only if that field is found to contain a non-`null` value. In [18], a static analysis is coupled to that described in Subsection 3.2, which computes a set of definitely non-`null` fields *at a given program point*. This *local* non-nullness information is performed through a denotational, bottom-up analysis of the program, which is proved correct in [18]. In this analysis, the abstraction of a piece of code contains a set of definitely non-`null` fields for each variable in scope. These sets are implemented as bitmaps, for better efficiency and for keeping down the memory consumption.

Differently from Subsection 3.1, method calls might modify the approximation of the local variables of the caller here, which are sets of fields definitely non-`null` and hence possibly reset to `null` by the callee, since they are not globally non-`null` as in Subsection 3.2. This is a major complication, which requires a preliminary *sharing analysis* to infer which local variables might be affected by each method call. We perform that analysis with a denotational abstract interpretation, defined and proved correct in [16], implemented with Boolean formulas. For better precision, we couple it with a constraint-based *creation points* analysis, which provides the set of object creation statements in the program where the values bound to a given variable at a given program point or to a given field might have been created. A *constraint* is a graph whose nodes stand for the approximation of each variable at each program point and of each field and whose arcs bind those approximations, reflecting the program's information flow. A constraint-based analysis builds a large constraint for the whole program and lets information flow inside it. In our case, creation points flow along the arcs. Sets are, again, implemented as bitmaps. We use the creation points analysis at each method call to compare the creation points of each field

update instruction reachable from the code of the callee with the creation points of each local variable of the callee: if they do not intersect, the execution of the callee cannot affect the approximation for that variable. Note that sharing and creation points analysis are complementary: the former is context-sensitive (in our implementation), which is not the case for the latter that, however, lets us reason on the receiver of each single field update operations that might be performed during the execution of the callee. Moreover, constraint-based analyses allow a precise approximation of properties of the fields, which is not easy with denotational static analyses. For a formal definition of a constraint-based analysis and proof of correctness, see the case of rawness analysis in [19].

This technique increases the precision of the nullness analysis, but also its computational cost, mainly because of sharing and creation points analysis. For instance, JULIA, using this technique, reports only 5 of the 6 warnings reported in Subsection 3.2 for the program in Figure 1:

```
[line 16: call with possibly-null receiver to String.equals(Object)]
line 24: call with possibly-null receiver to String.replace(char,char)
line 26: call with possibly-null receiver to String.replace(char,char)
line 31: read with possibly-null receiver of field inner
line 34: call with possibly-null receiver to C.toString()
line 35: call with possibly-null receiver to C.toString()
```

The subsequent table shows times and precision of larger analyses with this technique:

program	time	derefs	access	update	call	warnings
OurTunes	102.91 (was 31.94)	98.47% (was 95.09%)	97.94%	100.00%	97.91%	54 (was 173)
EJE	128.66 (was 43.34)	98.66% (was 98.25%)	99.85%	100.00%	97.55%	56 (was 74)
JFlex	151.60 (was 56.89)	94.75% (was 90.35%)	86.44%	100.00%	97.49%	460 (was 750)
utilMDE	235.76 (was 64.17)	97.40% (was 95.11%)	94.03%	100.00%	96.34%	117 (was 195)
AFU	182.16 (was 52.37)	98.66% (was 96.04%)	97.19%	100.00%	98.27%	126 (was 231)

3.4 Full Arrays

The analyses described so far always assume that the elements of an array of references are potentially `null`. Hence, for instance, in Subsection 3.3 we still get the three warnings at lines 24, 26 and 31 of the program in Figure 1. For better precision, we need a static analysis that spots those arrays of references that only contain non-`null` elements. We call such arrays *full*. Examples of full arrays are the `args` parameter passed by the Java Virtual Machine to method `main()` or explicitly initialised arrays such as `Object[] arr = { a, b, c }`, provided it is possible to prove that `a`, `b` and `c` hold non-`null` values at run-time. Other examples are arrays iteratively initialised with loops such as that at lines $23-27$ of the program in Figure 1. Those loops must provably initialise *all* elements of the array with definitely non-`null` values. Note that this property is relatively complex to prove, since the initialisation of the array elements can proceed in many ways and orders. For instance, in Figure 1, each iteration initialises two elements of `ts` at a time. Moreover, the initialising loop might end at the final element of the array (or, downwards, at the first) and this might be expressed

through a variable rather than the `.length` notation as in Figure 1. Furthermore, the loop is often a `for` loop, but it might also be a `while` loop or a `do...while` loop (since we analyse Java bytecode, there is no real difference between those loops). In general, it is hence unrealistic to consider all possible initialisation strategies, but some analysis is needed, that captures the most frequent scenarios. Moreover, a *semantical* analysis is preferable, rather than a weak syntactical pattern matching over the analysed code. In our implementation, definite aliasing information is used to prove that `ts.length` (in Figure 1) is the size of an array that is definitely initialised inside the body of the loop at lines $23 - 27$. If this is the case, a denotational analysis based on regular expressions builds the *shape* of the body of this loop: if this shape (*i.e.,* regular expression) looks as an initialisation of some variable `i` (the same compared to `ts.length`) to 0, followed by an alternation of array stores at `i` of non-`null` values and unitary increments of `i`, then the array is assumed to hold non-`null` values at the natural exit point of the loop, but not at exceptional exit points*, i.e.,* those that end the loop if it terminates abnormally because of some exception. We are currently working at improving this analysis, by considering more iteration strategies in the initialising loop. We also plan to consider the case when the array is held in a field rather than in a local variable.

Full arrays can be copied, stored into fields or passed as parameters to methods. Hence, we use a constraint-based static analysis that tracks the flow of the arrays in the program. During this analysis, we consider that full arrays may lose their property of being full as soon as an array store operation is executed, which writes a possibly `null` value. For better precision, we exploit the available static type information and use the creation points analysis, the same of Subsection 3.3, to determine the variables, holding full arrays, that might be affected by each array store operation.

It must be said that JULIA is able to reason on single array elements as well, if they are first tested for non-nullness and then dereferenced. For instance, it knows that `args[i]` does not hold `null` immediately after line 24 in Figure 1, even if it were not able to prove that `args` is full. This is because `args[i]` has been dereferenced there, so it cannot hold `null` immediately after (or otherwise an exception would interrupt the method). Of course, this requires JULIA to prove that the constructor of `C` and method `replace()` do not write `null` into `args`, by using sharing and creation points analysis. We have embedded this local array non-nullness analysis inside the technique of Subsection 3.3, since local non-nullness of fields and of array elements can be considered in a uniform way.

The improvements induced by all these approximations of the nullness of the array elements increase the precision of the nullness analysis. For instance, the analysis with JULIA of the program in Figure 1, using these extra techniques, yields only 2 of the 5 warnings reported in Subsection 3.3 now:

```
[line 24: call with possibly-null receiver to String.replace(char,char)]
[line 26: call with possibly-null receiver to String.replace(char,char)]
[line 31: read with possibly-null receiver of field inner]
line 34: call with possibly-null receiver to C.toString()
line 35: call with possibly-null receiver to C.toString()
```

These techniques increase precision and time of the nullness analysis of larger applications as well:

program	time	derefs	access	update	call	warnings
OurTunes	160.63 (was 102.91)	98.47% (was 98.47%)	97.94%	100.00%	97.91%	54 (was 54)
EJE	164.58 (was 128.66)	98.83% (was 98.66%)	100.00%	100.00%	97.81%	47 (was 56)
JFlex	173.02 (was 151.60)	95.27% (was 94.75%)	86.44%	100.00%	97.59%	409 (was 460)
utilMDE	298.88 (was 235.76)	97.90% (was 97.40%)	94.03%	100.00%	97.24%	81 (was 117)
AFU	222.24 (was 182.16)	98.86% (was 98.66%)	97.19%	100.00%	98.63%	107 (was 126)

3.5 Collection Classes

The Java programming language comes with an extensive library. Some prominent library classes are the *collection classes*, such as `Vector`, `HashSet` and `HashMap`. They are largely used in Java programs, but complicate the nullness analysis, since most of their instances are allowed to contain `null` elements, keys or values. As a consequence, there is no syntactical guarantee, for instance, that the iterations at lines 34 and 35 in Figure 1 happen over non-`null` elements only. This is why JULIA signals two warnings there, up to the technique of Subsection 3.4. The techniques described up to now do not help here since, for instance, the elements of a hashset are stored inside a backing hashmap, which contains an array of key/value *entries*. The technique of Subsection 3.2 might be able to prove that the field holding the key or that holding the value in all entries are globally non-`null`, but this is not the case: in Figure 1 hashmaps are not only used inside the two hashsets `s1` and `s2`, but also internally by the Java libraries themselves. In some of those uses, not apparent from Figure 1, `null` is stored (or seems to JULIA to be stored) as a value or key in a hashmap. In any case, the technique of Subsection 3.2 would be very weak here because it *flattens* all collections into the same, global abstraction for the nullness of the fields of the entries inside a hashmap: a program may use more hashsets or hashmaps (as is the case in Figure 1) and it is important, for better precision, to distinguish the collections possibly having `null` among their elements from those that only contain non-`null` elements: in Section 2 we have seen that commenting out line 31 introduces a warning at line 35 but not at line 34. The technique of Subsection 3.3 does not help either. The method `get()` of a hashmap or the method `next()` of an iterator over the keys or values of a hashmap does not check for the non-nullness of the field of the hashmap entry holding the returned value (as method `equals()` in Figure 1 does for field `inner`) nor assigns it to a definitely non-`null` value before returning its value.

To prove that the two calls to `toString()` in Figure 1 happen on a non-`null` variable `t`, we use a new technique. Namely, we let JULIA prove that `s1` and `s2` are sets of non-`null` elements. To that purpose, we have developed a constraint-based analysis that approximates each local variable at each given program point and each field with a flag, stating if the value of the variable or field is an instance of a collection class that does not contain `null`. Whenever a method is called, such as `HashSet.add()`, which might modify the flag of some variable, the analysis checks if it can prove that the call adds a non-`null` value to the collection. If this is not the case, the affected variables and fields lose the flag

stating that they do not contain `null` elements. In order to over-approximate the variables and fields affected, we use sharing and creation points analysis, as in Subsection 3.3.

The property of containing non-`null` elements only is propagated by the constraint-based analysis, following variable assignments, method calls and return. Also, if an iterator is built from a collection that does not contain `null` elements, then we flag that iterator as iterating over non-`null` elements only. If a possibly `null` element is added, later, to that iterator, it will lose its flag, and this will happen also to the variables holding the backing collection.

Our implementation of this analysis currently considers around 20 collection classes but we plan to consider more in the future. In order to simplify the addition of new classes, the analysis consults some Java annotations that we have written for the methods of the collection classes. Adding more classes is hence a matter of writing new annotations. In particular, one does not need to modify the analysis itself.

By using this technique, the nullness analysis with JULIA of the program in Figure 1 issues no warning, instead of the 2 of Subsection 3.4. The following table shows that this technique improves the precision of the analysis of larger applications as well:

program	time	derefs	access	update	call	warnings
`OurTunes`	162.82 (was 160.63)	99.01% (was 98.47%)	99.11%	100.00%	98.58%	38 (was 54)
`EJE`	157.61 (was 164.58)	99.07% (was 98.83%)	100.00%	100.00%	98.26%	36 (was 36)
`JFlex`	176.61 (was 173.02)	98.66% (was 95.27%)	99.14%	100.00%	97.80%	118 (was 409)
`utilMDE`	257.01 (was 298.88)	98.76% (was 97.90%)	100.00%	100.00%	97.78%	58 (was 81)
`AFU`	231.42 (was 222.24)	99.05% (was 98.86%)	98.31%	100.00%	98.72%	91 (was 107)

The time for analysing `EJE` and `utilMDE` has actually decreased *w.r.t.* Subsection 3.4, since the extra precision has accelerated the convergence of the oracle-based nullness analysis. Three of the warnings issued for `EJE` are actual `null`-pointer bugs of that program. The others are false alarms.

This analysis and that of Subsection 3.4 are strictly intertwined. The analysis in Subsection 3.4 must first determine the program points that initialise a full array. Then, the property of being full is propagated across the program, together with the same property for collection classes. The *fullness* for arrays and collections actually interact: if a collection is full, then its `toArray()` methods return a full array.

4 Construction of the Annotation File

In Section 3, we have seen the different phases of the nullness analysis of JULIA and how they provide different levels of precision for software verification, *i.e.,* different numbers of warnings. But nullness analysis can also be used to build an *annotation* of the program under analysis, reporting which fields, parameters or return values of methods might hold `null` at run-time. This is interesting to the programmer, is useful for documentation and can also be seen as a *standard description* of the nullness behaviour of the program. As such, it can be imported and exported between different tools for software analysis.

We did not devise our own annotation language for nullness, but used one that has been developed for the *Checker Framework* for Java [14]. The latter is a generic tool for software verification, based on type checking. Types can be specified and written into Java source code as Java annotations. The system type-checks those types and reports inconsistencies. The checker framework contains a type-system for nullness and is bundled with a tool (*file annotation utilities*) that imports a succinct description of the nullness behaviour of the program (a *jaif file*, in their terminology) into Java source code. This is perfect for JULIA: since our tool analyses Java bytecode, it has no direct view of the source code, but it can generate a jaif file which is then importable into source code, by using the file annotation utilities.

```
class C:
  field inner: @Nullable
  method <init>(Ljava.lang.String;LC;)V:
    parameter #1: @Nullable
  method equals(Ljava.lang.Object;)Z:
    parameter #0: @Nullable
```

Fig. 2. Jaif file generated by JULIA for the program in Figure 1

Figure 2 reports the jaif file generated by JULIA for the program in Figure 1. The default hypothesis is that everything is non-`null`, so that only possible nullness must be explicitly reported in the file. Hence Figure 2 says, implicitly, that field `name` in Figure 1 is non-`null` and that the constructor of class `C` (method `<init>`) always receives a non-`null` value as its first parameter (numbered as 0). Jaif files report also the nullness of the elements of an array of references or of an object of a collection class. Hence, since nothing is explicitly reported in Figure 2 about method `main()`, that figure tells us that `main()` in Figure 1 is always called with a parameter which is a non-`null` array of non-`null` strings.

For another example, consider the program in Figure 3 and the corresponding jaif file generated by JULIA, shown in Figure 4. This jaif file tells us that JULIA has been able to conclude that every call to methods `main()` and `first()` happens with a non-`null` argument of non-`null` strings, as well as every call to the constructor of `Test`. The same is not stated for method `second()`, since in some cases `null` is passed to `second()` for `x`. Moreover, the calls to method `inLoop()` are reported to happen with a non-`null` argument `x`, but whose elements might be `null` (see the annotation `inner-type 0: @Nullable`). This is true, since `inLoop()` is called with an array `s` as actual parameter that is not always full at the point of call. Field `g` is reported as possibly-`null` and there are actual cases when it contains `null` at run-time. Field `map` is reported as non-`null` but its `inner-type 1` is possibly-`null`: this is the value component of the map. Instead, its key component is definitely non-`null`, since there is no annotation for `map` about `inner-type 0` and the default is non-`null`.

The Checker Framework benefits from information about which values are *raw*, that is, are objects whose non-`null` fields might not have been assigned yet. This is important since, when a value is raw, the type-checker of the Framework correctly assumes its non-`null` fields to be potentially `null`. This is the case of `this` at the beginning of the constructor of `Test`, since its fields `map` and `f` (reported to be non-`null` in the jaif file) have not been yet initialised there. Hence also `this` inside `inLoop()` and `first()` is raw. But `this` inside `second()` is *not* raw since all its non-`null` fields have been already initialised when `second()` is called. In the jaif file in Figure 4, rawness information is reported with the `@Raw` annotation, applied to the receiver `this`, although, in more complex examples, we might find it reported for fields, parameters and return types as well. JULIA never reports it for the receiver of a constructor (such as that of `Test`), since it is the default there.

JULIA computes rawness information with a constraint-based *rawness analysis*, performed after the nullness analysis of Section 3. Rawness analysis is implemented as a constraint-based static analysis, where each variable at each given program point and each field is approximated with the set of its non-`null` fields that have been definitely initialised there. Those sets flow along the constraint,

```
import java.util.*;
public class Test {
  private Map<String, Object> map;
  private String f, g;
  public static void main(String[] args) {
    new Test(args);
  }
  private Test(String[] args) {
    map = new LinkedHashMap<String, Object>();
    String[] s = new String[args.length / 2];
    for (int i = 0; i < s.length; i++) {
      map.put(s[i] = args[i * 2], null);
      inLoop(s);
    }
    System.out.println(first(s));
    f = "name";
    if (args.length > 5)
      g = "surname";
    second(g);
  }
  private void inLoop(String[] x) {}
  private String first(String[] s) {
    return s.length > 0 ? s[0] : null;
  }
  private void second(String x) {}
}
```

Fig. 3. A simple Java program

```
class Test:
  field g: @Nullable
  field map:
    inner-type 1: @Nullable
  method second(Ljava.lang.String;)V:
    parameter #0: @Nullable
  method first([Ljava.lang.String;)Ljava.lang.String;:
    return: @Nullable
    receiver: @Raw
  method inLoop([Ljava.lang.String;)V:
    parameter #0:
      inner-type 0: @Nullable
    receiver: @Raw
```

Fig. 4. The jaif file generated by JULIA for the program in Figure 3

reflecting assignments, parameter passing and method returns, and are enlarged at field assignments. A formal definition and a proof of correctness of this analysis are contained in [19].

Although the jaif files built by JULIA are correct, this does not mean that their application to the source code type-checks *w.r.t.* the type-checker of the Checker Framework. This is because the techniques of Section 3 are data-flow, much different from those applied by that type-checker. Nevertheless, we have been working at making JULIA and the Checker Framework closer.

The DAIKON tool is also able to generate jaif files, but they are only likely correct. NIT generates jaif files also and computes some rawness information during its nullness analysis. Nevertheless, it does not dump the rawness information into the jaif file. Since NIT is less precise than the full-featured nullness analysis of JULIA, the generated jaif files are less precise too.

5 Conclusion

We have described and experimented with the nullness analysis implemented inside JULIA. It is correct and very precise, with a cost in time which is still acceptable for off-line analyses (a few minutes). Note that the only other correct nullness analysis for Java is that in [11], whose precision is similar to that of Subsection 3.2, as experimentally validated in [17].

Our nullness analysis is the composition of many static analyses, denotational and constraint-based. In general, denotational analyses provide context-sensitivity, while constraint-based analyses provide better support for the analysis of fields. In terms of software engineering, JULIA contains two implementations of those classes of analysis, from which all concrete static analyses are derived by subclassing. This has simplified the development of new analyses and allowed the optimisation of the shared components and their debugging.

Our work is not finished yet. First of all, we aim at making JULIA more scalable, up to including the whole `javax.*` hierarchy in the analysis. This will

benefit the precision of the results, particularly for the analysis of those applications that make use of the Swing graphical library. We are also working at making the results correct for multi-threaded applications. In this direction, we are developing a static analysis that identifies those fields that are only accessed by the thread that has assigned them. If this is the case, all techniques from Section 3 are correct for them. For the other fields (hopefully not many) a worst-case assumption will be made. The web interface of JULIA must also be improved in order to provide better graphical effects and more feedback to the user. It is available at the address `http://julia.scienze.univr.it`, where the reader can test the tool without any local installation.

References

1. Eclipse.org Home, http://www.eclipse.org
2. Jlint, http://artho.com/jlint
3. Bryant, R.E.: Graph-Based Algorithms for Boolean Function Manipulation. IEEE Transactions on Computers 35(8), 677–691 (1986)
4. Cok, D.R., Kiniry, J.: Esc/Java2: Uniting ESC/Java and JML. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 108–128. Springer, Heidelberg (2005)
5. Cousot, P., Cousot, R.: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: Proc. of the 4th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1977), Los Angeles, California, USA, pp. 238–252. ACM, New York (1977)
6. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The Daikon System for Dynamic Detection of Likely Invariants. Science of Computer Programming 69(1-3), 35–45 (2007)
7. Flanagan, C., Leino, K.R.M.: Houdini, an Annotation Assistant for ESC/Java. In: Oliveira, J.N., Zave, P. (eds.) FME 2001. LNCS, vol. 2021, pp. 500–517. Springer, Heidelberg (2001)
8. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended Static Checking for Java. In: Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2002), Berlin, Germany. SIGPLAN Notices, vol. 37(5), pp. 234–245. ACM, New York (May 2002)
9. Goetz, B., Peierls, T., Bloch, J., Bowbeer, J., Holmes, D., Lea, D.: Java Concurrency in Practice. Addison-Wesley, Reading (2006)
10. Hovemeyer, D., Pugh, W.: Finding More Null Pointer Bugs, but Not Too Many. In: Das, M., Grossman, D. (eds.) Proc. of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 2007), San Diego, California, USA, pp. 9–14. ACM, New York (June 2007)
11. Hubert, L., Jensen, T., Pichardie, D.: Semantic Foundations and Inference of non-null Annotations. In: Barthe, G., de Boer, F.S. (eds.) FMOODS 2008. LNCS, vol. 5051, pp. 132–149. Springer, Heidelberg (2008)
12. Lindholm, T., Yellin, F.: The Java™ Virtual Machine Specification, 2nd edn. Addison-Wesley, Reading (1999)
13. Palsberg, J., Schwartzbach, M.I.: Object-oriented Type Inference. In: Proc. of the 6th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 1991), Phoenix, Arizona, USA. ACM SIGPLAN Notices, vol. 26(11), pp. 146–161. ACM, New York (November 1991)

14. Papi, M.M., Ali, M., Correa, T.L., Perkins, J.H., Ernst, M.D.: Practical Pluggable Types for Java. In: Ryder, B.G., Zeller, A. (eds.) Proc. of the ACM/SIGSOFT 2008 International Symposium on Software Testing and Analysis (ISSTA 2008), Seattle, Washington, USA, pp. 201–212. ACM, New York (July 2008)
15. Rutar, N., Almazan, C.B., Foster, J.S.: A Comparison of Bug Finding Tools for Java. In: Proc. of the 15th International Symposium on Software Reliability Engineering (ISSRE 2004), Saint-Malo, France, pp. 245–256. IEEE Computer Society, Los Alamitos (November 2004)
16. Secci, S., Spoto, F.: Pair-Sharing Analysis of Object-Oriented Programs. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 320–335. Springer, Heidelberg (2005)
17. Spoto, F.: Nullness Analysis in Boolean Form. In: Proc. of the 6th IEEE International Conference on Software Engineering and Formal Methods (SEFM 2008), Cape Town, South Africa, pp. 21–30. IEEE Computer Society, Los Alamitos (November 2008)
18. Spoto, F.: Precise null-Pointer Analysis. Software and Systems Modeling (to appear, 2010)
19. Spoto, F., Ernst, M.: Inference of Field Initialization. Technical Report UW-CSE-10-02-01, University of Washington, Department of Computer Science & Engineering (February 2010)
20. Spoto, F., Mesnard, F., Payet, E.: A Termination Analyser for Java Bytecode Based on Path-Length. ACM Transactions on Programming Languages and Systems (TOPLAS) 32(3) (March 2010)

Qex: Symbolic SQL Query Explorer

Margus Veanes, Nikolai Tillmann, and Jonathan de Halleux

Microsoft Research, Redmond, WA, USA
{margus,nikolait,jhalleux}@microsoft.com

Abstract. We describe a technique and a tool called Qex for generating input tables and parameter values for a given parameterized SQL query. The evaluation semantics of an SQL query is translated into a specific background theory for a satisfiability modulo theories (SMT) solver as a set of equational axioms. Symbolic evaluation of a goal formula together with the background theory yields a model from which concrete tables and values are extracted. We use the SMT solver Z3 in the concrete implementation of Qex and provide an evaluation of its performance.

1 Introduction

The original motivation behind Qex comes from *unit testing* of relational databases, where a key challenge is the automatic generation of input tables and parameters for a given query and a given test condition, where a typical test condition is that the result of the query is a nonempty table. An early prototype of Qex as a proof-of-concept and an integration of Qex into the Visual Studio Database edition is discussed in [23,28]. Here we present a new approach for encoding queries that uses algebraic data types and equational axioms, taking advantage of recent advances in SMT technology. The encoding is much simpler than the one described in [28], and boosted the performance of Qex by several orders of magnitude. In [28] algebraic data types were not available and queries were encoded into an intermediate background theory $\mathcal{T}^{\Sigma}$ using bags and a summation operator. The resulting formula was eagerly expanded, for a given size of the database, into a quantifier free formula that was then asserted to the SMT solver. The expansion often caused an exponential blowup in the size of the expanded formula, even when some parts of the expansion were irrelevant with respect to the test condition. The new approach not only avoids the eager expansion but avoids also the need for nonlinear constraints that arise when dealing with multiplicities of rows in bags and aggregates over bags. Moreover, the axiomatic approach makes it possible to encode frequently occurring like-patterns through an automata based technique, and other string constraints. To this end, Qex now encodes strings faithfully as character sequences, whereas in [28] strings were abstracted to integers with no support for general string operations. Furthermore, algebraic data types provide a straightforward encoding for value types that allow null. In addition, Qex now also handles table constraints and uses symmetry breaking formulas for search space reduction.

E.M. Clarke and A. Voronkov (Eds.): LPAR-16, LNAI 6355, pp. 425–446, 2010.

The core idea is as follows. A given SQL query q is translated into a term $[\![q]\!]$ over a rich background theory that comes with a collection of built-in (predefined) functions. Tables are represented by lists of tuples, where lists are built-in algebraic data types. In addition to built-in functions (such as arithmetical operations) the term $[\![q]\!]$ may also use functions whose meaning is governed by a set of additional axioms referred to as $Th(q)$. These custom axioms describe the evaluation rules of SQL queries and are in most cases defined as recursive list axioms that resemble functional programs. Table 1 provides a rough overview of the SQL constructs supported in Qex and the corresponding theories used for mapping a given construct into a formula for Z3 [30,10] that is used as the underlying SMT solver in the implementation of Qex. As indicated in the table, in all of the cases there is also an additional set of custom axioms that are used in addition to the built-in ones.

Table 1. Overview of features in Qex and related use of SMT theories

	Built-in theories						**Custom**
Features	Arithmetic	Bitvectors	Sets	Arrays	Algebraic d.t.	Tuples	**theories**
Table constraints	√		√		√	√	√
SELECT clauses	√				√		√
Aggregates	√		√	√	√	√	√
LIKE patterns		√			√		√
Null					√		√

For input tables and other parameters, the term $[\![q]\!]$ uses uninterpreted constants. Given a condition φ over the result of $[\![q]\!]$, e.g., $[\![q]\!] \neq nil$ ($[\![q]\!]$ is nonempty), φ is asserted to the SMT solver as a goal formula and $Th(q)$ is asserted to the SMT solver as an additional set of axioms, sometimes called a *soft theory*. Next, a satisfiability check is performed together with *model generation*. If φ is satisfiable then the generated model is used to extract concrete values (interpretations) for the input table constants and other additional parameter constants.

The rest of the paper is structured as follows. Section 2 introduces some basic notions that are used throughout the paper. Section 3 defines a custom theory of axioms over lists that are used in Section 4 to translate queries into formulas. Section 5 discusses the implementation of Qex with a focus on its interaction with Z3. Section 6 provides some experimental evaluation of Qex. Section 7 is about related work, and Section 8 provides some final remarks.

2 Preliminaries

We assume that the reader is familiar with elementary concepts in logic and model theory, our terminology is consistent with [15] in this regard.

We are working in a fixed multi-sorted universe $\mathcal{U}$ of values. For each sort σ, $\mathcal{U}^\sigma$ is a separate subuniverse of $\mathcal{U}$. The basic sorts needed in this paper are the

Boolean sort $\mathbb{B}$, ($\mathcal{U}^{\mathbb{B}} = \{true, false\}$), the integer sort $\mathbb{Z}$, and the n-tuple sort $\mathbb{T}\langle\sigma_0, \ldots, \sigma_{n-1}\rangle$ for $n \geq 1$ of some given basic sorts σ_i for $i < n$. We also use other sorts but they are introduced at the point when they are first needed.

There is a collection of functions with a fixed meaning associated with the universe, e.g., arithmetical operations over $\mathcal{U}^{\mathbb{Z}}$. These functions and the corresponding function symbols are called *built-in*. Each function symbol f of arity $n \geq 0$ has a fixed domain sort $\sigma_0 \times \cdots \times \sigma_{n-1}$ and a fixed range sort σ, $f : \sigma_0 \times \cdots \times \sigma_{n-1} \to \sigma$. For example, there is a built-in *relation* or *predicate* (Boolean function) symbol $< : \mathbb{Z} \times \mathbb{Z} \to \mathbb{B}$ that denotes the standard order on integers. One can also declare *fresh* (new) *uninterpreted* function symbols f of arity $n \geq 0$, for a given domain sort and a given range sort. Using model theoretic terminology, these new symbols *expand* the signature.

Terms and *formulas* (or Boolean terms) are defined by induction as usual and are assumed to be well-sorted. We write $FV(t)$ for the set of free variables in a term (or formula) t. A term or formula without free variables is *closed*.

A *model* is a mapping from function symbols to their interpretations (values). The built-in function symbols have the same interpretation in all models that we are considering, keeping that in mind, we may omit mentioning them in a model. A model M *satisfies* a closed formula φ, $M \models \varphi$, if it provides an interpretation for all the uninterpreted function symbols in φ that makes φ true. For example, let $f : \mathbb{Z} \to \mathbb{Z}$ be an uninterpreted function symbol and $c : \mathbb{Z}$ be an uninterpreted constant. Let M be a model where c^M (the interpretation of c in M) is 0 and f^M is a function that maps all values to 1. Then $M \models 0 < f(c)$ but $M \not\models 0 < c$.

A closed formula φ is *satisfiable* if it has a model. A formula φ with $FV(\varphi) = \bar{x}$ is *satisfiable* if its existential closure $\exists\bar{x}\varphi$ is satisfiable. We write $\models_{\mathcal{U}} \varphi$, or $\models \varphi$, if φ is *valid* (true in all models). Some examples: $0 < 1 \wedge 2 < 10$ is valid; $4 < x \wedge x < 5$, where $x : \mathbb{Z}$ is a free variable, is unsatisfiable because there exists no integer between 4 and 5; $0 < x \wedge x < 3$, where $x : \mathbb{Z}$ is a free variable, is satisfiable.

3 Equational Axioms over Lists

The representation of a table in Qex is a *list* of rows, where a row is a tuple. While bags of rows rather than lists would model the semantics of SQL more directly (order of rows is irrelevant, but multiple occurrences of the same row are relevant), the inductive structure of a list provides a way to define the evaluation semantics of queries by recursion. The mapping of queries to axioms, discussed in Section 4, uses a collection of axioms over lists that are defined next. Intuitively, the axioms correspond to definitions of standard (higher order) functionals that are typical in functional programming. The definitions of the axioms below, although more concise, correspond precisely to their actual implementation in Qex using the Z3 API. Before describing the actual axioms, we explain the intuition behind a particular kind of axioms, that we call *equational*, when used in an SMT solver.

3.1 Equational Axioms and *E*-Matching in SMT Solvers

During proof search in an SMT solver, axioms are triggered by matching subexpressions in the goal. Qex uses particular kinds of axioms, all of which are equations of the form

$$\forall \bar{x}(t_{\text{lhs}} = t_{\text{rhs}}) \tag{1}$$

where $FV(t_{\text{lhs}}) = \bar{x}$ and $FV(t_{\text{rhs}}) \subseteq \bar{x}$. The left-hand-side t_{lhs} of (1) is called the *pattern* of (1).

> *While SMT solvers support various kinds of patterns in general, in this paper we use the convention that the pattern of an equational axiom is always its left-hand-side.*

The high-level idea behind E-matching is as follows. The axiom (1) is *triggered* by the current goal ψ of the solver, if ψ contains a subterm u and there exists a substitution θ such that $u =_E t_{\text{lhs}}\theta$, i.e., *u matches the pattern* of the axiom (modulo the built-in theories E). If (1) is triggered, then the current goal is replaced by the *logically equivalent* formula where u has been replaced by $t_{\text{rhs}}\theta$.

Thus, the axioms that are used in Qex can be viewed as "rewrite rules", and each application of an axiom preserves the logical equivalence to the original goal. As long as there exists an axiom in the current goal that can be triggered, then triggering is guaranteed. Thus, termination is in general not guaranteed in the presence of (mutually) recursive axioms. Note that, unlike in term rewrite systems, there is no notion of term orderings or well-defined customizable strategies (at least not in the current version of Z3) that could be used to guide the triggering process of the axioms.

3.2 Axioms over Lists

For each sort σ there is a built-in *list sort* $\mathbb{L}\langle\sigma\rangle$ and a corresponding subuniverse $\mathcal{U}^{\mathbb{L}\langle\sigma\rangle}$. (In Z3, lists are provided as built-in algebraic data types and are associated with standard constructors and accessors.) For a given element sort σ there is an empty list *nil* (of sort $\mathbb{L}\langle\sigma\rangle$) and if e is an element of sort σ and l is a list of sort $\mathbb{L}\langle\sigma\rangle$ then $cons(e, l)$ is a list of sort $\mathbb{L}\langle\sigma\rangle$. The accessors are, as usual, *hd* (head) and *tl* (tail). In the following consider a fixed element sort σ. Observe that one can define a well-ordering such that, in all of the recursive cases of the axioms, the right-hand-side decreases with respect to that ordering, which guarantees that triggering terminates and implies that the axioms are well-defined. In all of the cases, the use of the list constructors in the patterns is fundamental. In most cases one can provide more compact and logically equivalent definitions of the axioms where the right-hand-sides are combined in a disjunction, but where the pattern is too general and may cause nontermination of axiom triggering in an SMT solver.

Filter. Let φ be a formula with a single free variable $x_0 : \sigma$. Declare the function symbol $\mathit{Filter}[\varphi] : \mathbb{L}\langle\sigma\rangle \to \mathbb{L}\langle\sigma\rangle$ and define the following axioms:

$$\begin{aligned} \mathit{Filter}[\varphi](\mathit{nil}) &= \mathit{nil} \\ \forall x_0\, x_1\, (\mathit{Filter}[\varphi](\mathit{cons}(x_0, x_1)) &= \mathit{Ite}(\varphi, \mathit{cons}(x_0, \mathit{Filter}[\varphi](x_1)), \mathit{Filter}[\varphi](x_1))) \end{aligned}$$

The *Ite*-term $\mathit{Ite}(\phi, t_1, t_2)$ equals t_1, if ϕ is true; it equals t_2, otherwise. *Ite* is a built-in function.

Map. Let $t : \rho$ be a term with a single free variable $x_0 : \sigma$. Declare the function symbol $\mathit{Map}[t] : \mathbb{L}\langle\sigma\rangle \to \mathbb{L}\langle\rho\rangle$ and define:

$$\begin{aligned} \mathit{Map}[t](\mathit{nil}) &= \mathit{nil} \\ \forall x_0\, x_1\, (\mathit{Map}[t](\mathit{cons}(x_0, x_1)) &= \mathit{cons}(t, \mathit{Map}[t](x_1))) \end{aligned}$$

Reduce. Let $t : \rho$ be a term with two free variables $x_0 : \sigma$ and $x_1 : \rho$. Declare the function symbol $\mathit{Reduce}[t] : \mathbb{L}\langle\sigma\rangle \times \rho \to \rho$ and define:

$$\begin{aligned} \forall x\, (\mathit{Reduce}[t](\mathit{nil}, x) &= x) \\ \forall x_0\, x_1\, x_2\, (\mathit{Reduce}[t](\mathit{cons}(x_0, x_2), x_1) &= \mathit{Reduce}[t](x_2, t)) \end{aligned}$$

For example, if $l : \mathbb{L}\langle\mathbb{Z}\rangle$ is a list of integers, then $\mathit{Reduce}[x_0 + x_1](l, 0)$ is equal to the sum of the integers in l, or 0 if l is empty (in any model that satisfies the corresponding *Reduce*[]-axioms).

Cross product. Declare the function symbols $\mathit{Cross} : \mathbb{L}\langle\sigma\rangle \times \mathbb{L}\langle\rho\rangle \to \mathbb{L}\langle\mathbb{T}\langle\sigma, \rho\rangle\rangle$ and $\mathit{Cr} : \sigma \times \mathbb{L}\langle\sigma\rangle \times \mathbb{L}\langle\rho\rangle \times \mathbb{L}\langle\rho\rangle \to \mathbb{L}\langle\mathbb{T}\langle\sigma, \rho\rangle\rangle$, and define

$$\begin{aligned} \forall x\, (\mathit{Cross}(\mathit{nil}, x) &= \mathit{nil}) \\ \forall x\, (\mathit{Cross}(x, \mathit{nil}) &= \mathit{nil}) \\ \forall \bar{x}\, (\mathit{Cross}(\mathit{cons}(x_0, x_1), \mathit{cons}(x_2, x_3)) &= \mathit{Cr}(x_0, x_1, \mathit{cons}(x_2, x_3), \mathit{cons}(x_2, x_3))) \\ \forall \bar{x}\, (\mathit{Cr}(x_0, x_1, \mathit{nil}, x_2) &= \mathit{Cross}(x_1, x_2)) \\ \forall \bar{x}\, (\mathit{Cr}(x_0, x_1, \mathit{cons}(x_2, x_3), x_4) &= \mathit{cons}(T(x_0, x_2), \mathit{Cr}(x_0, x_1, x_3, x_4))) \end{aligned}$$

where $T : \sigma \times \rho \to \mathbb{T}\langle\sigma, \rho\rangle$ is the built-in tuple constructor (for the given sorts). For example, the term $\mathit{Cross}(\mathit{cons}(1, \mathit{cons}(2, \mathit{nil})), \mathit{cons}(3, \mathit{cons}(4, \mathit{nil})))$ is equal to the term

$$\mathit{cons}(T(1,3), \mathit{cons}(T(1,4), \mathit{cons}(T(2,3), \mathit{cons}(T(2,4), \mathit{nil})))).$$

Remove duplicates. The function *RemoveDuplicates* is used to remove duplicates from a list. The definition makes use of built-in sets and set operations; the set sort of element sort σ is denoted $\mathbb{S}\langle\sigma\rangle$.

Declare: $\mathit{RemoveDuplicates} : \mathbb{L}\langle\sigma\rangle \to \mathbb{L}\langle\sigma\rangle$, $\mathit{Rd} : \mathbb{L}\langle\sigma\rangle \times \mathbb{S}\langle\sigma\rangle \to \mathbb{L}\langle\sigma\rangle$. Define:

$$\begin{aligned} \forall x\,(\mathit{RemoveDuplicates}(x) &= \mathit{Rd}(x, \emptyset)) \\ \forall x\,(\mathit{Rd}(\mathit{nil}, x) &= \mathit{nil}) \\ \forall \bar{x}\,(\mathit{Rd}(\mathit{cons}(x_0, x_1), x_2) &= \mathit{Ite}(x_0 \in x_2, \mathit{Rd}(x_1, x_2), \\ &\qquad \mathit{cons}(x_0, \mathit{Rd}(x_1, \{x_0\} \cup x_2)))) \end{aligned}$$

Select with grouping and aggregates. Select clauses with aggregates and grouping are translated into formulas using the following axioms. Each aggregate function α (either *MIN*, *MAX*, or *SUM*) for a sort σ is defined as a binary operation over the lifted sort $?\langle\sigma\rangle$, i.e., $\alpha : ?\langle\sigma\rangle \times ?\langle\sigma\rangle \to ?\langle\sigma\rangle$. The data type $?\langle\sigma\rangle$ is associated with the constructors $\mathit{NotNull} : \sigma \to ?\langle\sigma\rangle$, $\mathit{Null} : ?\langle\sigma\rangle$, the accessor $\mathit{Value} : ?\langle\sigma\rangle \to \sigma$ (that maps any value $\mathit{NotNull}(a)$ to a), and the testers $\mathit{IsNotNull} : ?\langle\sigma\rangle \to \mathbb{B}$, $\mathit{IsNull} : ?\langle\sigma\rangle \to \mathbb{B}$. Regarding implementation, such data types are directly supported in the underlying solver Z3. (For *COUNT* the range sort is $?\langle\mathbb{Z}\rangle$.) In SQL, aggregation over an empty collection yields null and null elements in the collection are discarded, e.g., sum aggregation over an empty collection yields null. The definition of *MAX* (similarly for *MIN*) is:

$$\begin{aligned} \mathit{MAX}(x_0, x_1) \stackrel{\text{def}}{=} \mathit{Ite}(&\mathit{IsNull}(x_0), x_1, \mathit{Ite}(\mathit{IsNull}(x_1), x_0, \\ &\mathit{Ite}(\mathit{Value}(x_0) > \mathit{Value}(x_1), x_0, x_1))) \end{aligned}$$

The definition of *SUM* is:

$$\begin{aligned} \mathit{SUM}(x_0, x_1) \stackrel{\text{def}}{=} \mathit{Ite}(&\mathit{IsNull}(x_0), x_1, \mathit{Ite}(\mathit{IsNull}(x_1), x_0, \\ &\mathit{NotNull}(\mathit{Value}(x_0) + \mathit{Value}(x_1)))) \end{aligned}$$

Let $t : \rho$ be a term with a single free variable $x_0 : \sigma$. Let $a : \zeta$ be a term with a single free variable $x_0 : \sigma$. Intuitively, σ is a tuple sort, both t and a are projections, and a corresponds to an aggregate parameter. For example (see the schema in Example 1 below) x_0 is a row in the `Scores` table, t corresponds to the projection `Scores.StudentID`, and a corresponds to the projection `Scores.Points` in `MAX(Scores.points)`.

We declare the function symbol $\mathit{Select}^{\alpha}[t, a] : \mathbb{L}\langle\sigma\rangle \to \mathbb{L}\langle\mathbb{T}\langle\rho, \zeta\rangle\rangle$ and define a set of recursive axioms for it that for each element in the list collect the aggregated value with respect to a and then create a list of pairs that for each projection t provides that aggregated value. In order to define these axioms, arrays (mathematical maps) are used.

Given domain sort σ_1 and range sort σ_2, $\mathbb{A}\langle\sigma_1, \sigma_2\rangle$ is the corresponding *array sort*. (In particular, the set sort $\mathbb{S}\langle\sigma_1\rangle$ is synonymous with $\mathbb{A}\langle\sigma_1, \mathbb{B}\rangle$.) Declare

$$\begin{aligned} \mathit{Select}^{\alpha}[t, a] &: \mathbb{L}\langle\sigma\rangle \to \mathbb{L}\langle\mathbb{T}\langle\rho, \zeta\rangle\rangle, \\ \mathit{Collect} &: \mathbb{L}\langle\sigma\rangle \times \mathbb{A}\langle\rho, \zeta\rangle \times \mathbb{L}\langle\sigma\rangle \to \mathbb{L}\langle\mathbb{T}\langle\rho, \zeta\rangle\rangle, \\ \mathit{List} &: \mathbb{L}\langle\sigma\rangle \times \mathbb{A}\langle\rho, \zeta\rangle \to \mathbb{L}\langle\mathbb{T}\langle\rho, \zeta\rangle\rangle \end{aligned}$$

and define the following axioms, where $Read : \mathbb{A}\langle\rho,\zeta\rangle \times \rho \to \zeta$ and $Store : \mathbb{A}\langle\rho,\zeta\rangle \times \rho \times \zeta \to \mathbb{A}\langle\rho,\zeta\rangle$ are the standard built-in functions of the array theory. The empty array ϵ maps all elements of the domain sort to the default value of the range sort. For lifted sorts the default value is null.

$$\forall x\,(Select^{\alpha}[t,a](x) = Collect(x,\epsilon,x))$$

$$\forall \bar{x}\,(Collect(cons(x_0,x_1),x_2,x_3) = Collect(x_1, Store(x_2,t,\alpha(a,Read(x_2,t))),x_3))$$

$$\forall \bar{x}\,(Collect(nil,x_0,x_1) = List(x_1,x_0))$$

$$\forall \bar{x}\,(List(cons(x_0,x_1),x_2) = cons(T(t,Read(x_2,t)),List(x_1,x_2)))$$

$$\forall x\,(List(nil,x) = nil)$$

In the current implementation, the above axioms are specialized to the case when the aggregate argument is required to be non null (for performance reasons), and the sort of a is not lifted. Although lifted sorts are avoided, this limitation requires special treatment of the cases when the collection is empty and implies that aggregates do not work with nullable column types.

4 From SQL to Formulas

In this section we show how we translate an SQL query q into a set of axioms $Th(q)$ that is suitable as an input soft theory to an SMT solver. The translation makes use of the list axioms discussed in Section 3. Although functional encodings of queries through comprehensions and combinators have been used earlier for compiler construction and query optimization (e.g. [14]), we are not aware of such encodings having been used for symbolic analysis or SMT solving. We illustrate the encodings here in order to make the paper self-contained. The concrete implementation with Z3 terms is very similar.

We omit full details of the translation and illustrate it through examples and templates, which should be adequate for understanding how the general case works. The focus is on the purely relational subset of SQL (without side-effects). We start by describing how tables are represented.

4.1 Tables and Table Constraints

Tables are represented by lists of rows where each row is a tuple. The sorts of the elements in the tuple are derived from the types of the corresponding columns that are given in the database schema. The currently supported column types in Qex are: `BigInt`, `Int`, `SmallInt`, `TinyInt`, `Bit`, and `Char`. The first four types are mapped to $\mathbb{Z}$ (and associated with a corresponding range constraint, e.g., between 0 and 255 for `TinyInt`). `Bit` is mapped to $\mathbb{B}$. `Char` (that in SQL stands for a sequence of characters) is mapped to the *string sort* (or *word sort*) $\mathbb{W} = \mathbb{L}\langle\mathbb{C}\rangle$, where $\mathbb{C}$ is the built-in sort of n-bitvectors for some fixed n that depends on the character range: UTF-16 ($n = 16$), basic ASCII ($n = 7$), extended ASCII ($n = 8$).

The order of rows in a table is irrelevant regarding the evaluation semantics of queries. The number of times the same row occurs in a table is the *multiplicity* of the row. In general, duplicate rows are allowed in tables so the multiplicity may be more than one. However, in most cases input tables have primary keys that disallow duplicates. Tables may also be associated with other constraints such as foreign key constraints and restrictions on the values in the columns. In Qex, these constraints are translated into corresponding formulas on the list elements. The following example illustrates that aspect of Qex.

Example 1. Consider the following schema for a school database.

```
CREATE TABLE [School].[Scores]
(StudentID tinyint not null FOREIGN KEY REFERENCES Students(StudentNr),
 CourseID tinyint not null CHECK(1 <= CourseID and CourseID <= 100),
 Points tinyint not null CHECK(Points <= 10),
 PRIMARY KEY (StudentID, CourseID),
 CHECK(NOT(1 <= CourseID and CourseID <= 10) or Points < 6));

CREATE TABLE [School].[Students]
(StudentNr tinyint not null PRIMARY KEY,
 StudentName char(100) not null);
```

The (primary) key of the `Scores` table is the pair containing a student id and a course id and each row provides the number of points the student has received for the given course. The additional constraints are that the course ids go from 1 to 100, no course gives more than 10 points and courses 1 through 10 give a maximum of 5 points.

Qex declares the variables $Scores : \mathbb{L}\langle\mathbb{T}\langle\mathbb{Z},\mathbb{Z},\mathbb{Z}\rangle\rangle$ and $Students : \mathbb{L}\langle\mathbb{T}\langle\mathbb{Z},\mathbb{W}\rangle\rangle$ for tables. There is a given bound k on the number of rows in each table. (In general there is a separate bound per table and the bounds are increased during model generation discussed in Section 5.) The following equalities are generated:

$$Scores = cons(Scores_0, \ldots, cons(Scores_{k-1}, nil))$$
$$Students = cons(Students_0, \ldots, cons(Students_{k-1}, nil))$$

where $Scores_i : \mathbb{T}\langle\mathbb{Z},\mathbb{Z},\mathbb{Z}\rangle$ and $Students_i : \mathbb{T}\langle\mathbb{Z},\mathbb{W}\rangle$ for $i < k$. For the primary key constraints, the following formulas are generated. The distinctness predicate and the projections functions π_i on tuples are built-in. *We use t.i to abbreviate the term* $\pi_i(t)$.

$$Distinct(T(Scores_0.0, Scores_0.1), \ldots, T(Scores_{k-1}.0, Scores_{k-1}.1))$$
$$Distinct(Students_0.0, \ldots, Students_{k-1}.0)$$

For expressing the foreign key constraint, Qex uses the built-in sets and the subset predicate:

$$\{Scores_i.0\}_{i<k} \subseteq \{Students_i.0\}_{i<k}$$

Currently, foreign key constraints are not supported over nullable types. The remaining constraints are conjunctions of check-constraints on individual rows, e.g.,

$$\bigwedge_{i<k} (\neg(1 \leq Scores_i.1 \wedge Scores_i.1 \leq 10) \vee Scores_i.2 < 6)$$

asserts that courses 1 through 10 give a maximum of 5 points.

4.2 Nullable Values

If a column in a table is optional, it may contain "null" as a placeholder. Any column in SQL (other than a primary key column) is optional unless a `not null` type constraint is associated with the column type. Algebraic data types provide a convenient mechanism to represent optional values through *lifted sorts* as defined in Section 3.2.

When an SQL expression E is encoded as a term $[\![E]\!]$, it is assumed that E is *well-formed*: in the current implementation, operations using optional values are assumed to occur in a context where the value is known to be not null. SQL includes particular predicates `IS NULL` and `IS NOT NULL` for this purpose. In the translation the corresponding testers are used and the *Value* accessor is applied to cast the optional value to its underlying sort.

For example, assuming the column `Points` of the `Scores` table is declared `NOT NULL` (as in Example 1), the expression $E =$ `Points > 3` is translated to $Scores.2 > 3$, but if `Points` is nullable, the expression E would have to occur in a context that is guarded by `Points IS NOT NULL`, e.g., $E =$`Points IS NOT NULL AND Points > 3`, in which case $[\![E]\!]$ is $IsNotNull(Scores.2) \wedge Value(Scores.2) > 3$.

Currently, well-formedness is not automatically detected and automatic support for such transformations is on the to-do-list. Regarding aggregates, the current implementation of Qex does not support aggregation over nullable types and a proper support for nullable values in combination of aggregates requires and adaptation of the corresponding axioms, which is yet another item on the to-do-list.

4.3 Formulas for Queries

As the concrete input of queries, Qex uses a subset of the abstract syntax of the TSQL [1] grammar and the parser `TSql100Parser` that is available in the VSTS'08 database edition. The currently supported constructs, some of which are also illustrated in the examples, are

- Selection, projection, group-by with having clause, inner join, nested queries.
- Aggregates: MIN, MAX, COUNT, SUM.
- Check constraints (composite), foreign and primary key constraints (composite).
- Arithmetic operations including negative numbers.
- Like-patterns and string length constraints.
- Restricted form of null support in table schemas.

We refer to the supported fragment by SQL^-. For dealing with null, the current translation does not fully support key constraints where values may be null or aggregates over columns where null is possible. Some of the corner cases require careful special handling in order to stay faithful to the semantics of SQL. Similarly, variable length string types and various character encodings are currently not supported. Although the underlying solver is capable of supporting full Unicode, the current experiments assume ASCII character encoding. The following constructs are currently not supported.

- Nested queries in from clauses. Correlated nested queries. Other join operations besides inner join.
- Set-type operands in where clauses, exists-expressions and in-expressions. Set operations such as union, intersection and difference.
- Order by.
- Store procedures.

Regarding the first two items, there is a plan to support most common cases. Order by clauses are viewed as postprocessing of the result and are currently not planned to be supported as part of model generation. Store procedures fall outside the scope of this paper, although there are future plans to look into symbolic execution of store procedures.

It is not feasible to fit the details of the translation from queries to formulas into the paper, instead, we look at a collection of representative samples that illustrate the core ideas behind the translation. In the samples, we reuse the schema from Example 1. We denote the term resulting from an SQL^- expression E by $[\![E]\!]$. The overall goal of the translation is summarized by the following proposition. Given a list l let $\{\!\{l\}\!\}$ denote the corresponding multiset where the order of list elements is removed.

Proposition 1. *Let q be an SQL^- query using input table references $X_i, i < n$, let ψ be a formula expressing the input table constraints, and let φ be a condition over the result Y of q. If $Th(q) \wedge \psi \wedge \varphi \wedge Y = [\![q]\!]$ has a model M then $\{\!\{X_i^M\}\!\}, i < n$, is a set of input tables satisfying ψ and the evaluation of q with respect to the input tables produces the result $\{\!\{Y^M\}\!\}$ satisfying φ.*

Proof (Proof (sketch).). The complete proof uses induction over the structure of SQL^- expressions and is contingent upon a complete definition of SQL^- as well as a formal mapping of SQL types to the corresponding background sorts. For example, the select clause has the following abstract syntax in simplified form:

select_clause `::= SELECT [DISTINCT]` *select_list*
`FROM` *table_src* `[WHERE` *condition*`]` `[`*group_by_having*`]`

The translation of a select clause depends on whether grouping is used and whether aggregates occur in the select list. Suppose q is a simple select clause `SELECT` L `FROM` T `WHERE` C without aggregates. The expression `FROM` T `WHERE` C is translated to $t = Filter[\![C]\!]([\![T]\!])$ that filters out all elements in the list $[\![T]\!]$ that do not satisfy the condition $[\![C]\!]$. Note that this translation preserves the multiplicities of the elements in $[\![T]\!]$ and is consistent with the multiset semantics. The

translation $[\![q]\!]$ of q is $Map[[\![L]\!]](t)$ where the elements of t are projected according to L. This translation also preserves the multiset semantics even if the projection $[\![L]\!]$ is not injective, i.e., several occurrences of the same element may arise as a result of the map operation. Other SQL^- expressions are treated similarly.

SELECT clauses. The main component of a query is a *select clause*. A select clause refers to a particular selection of columns from a given table by using a *select list*. The table is often a derived table, as the result of a join operation. Consider the query q:

```
SELECT StudentName, Points
FROM Students JOIN Scores ON Scores.StudentID = Students.StudentNr
WHERE Scores.CourseID = 10 AND Scores.Points > 0
```

The formula $[\![q]\!]$ is:

$$Map[T(x.0.1, x.1.2)](Filter[x.1.1 = 10 \wedge x.1.2 > 0](\\ Filter[x.0.0 = x.1.0](Cross(Students, Scores))))$$

where $x : \mathbb{T}\langle\mathbb{T}\langle\mathbb{Z}, \mathbb{W}\rangle, \mathbb{T}\langle\mathbb{Z}, \mathbb{Z}, \mathbb{Z}\rangle\rangle$. Such formulas get unreadable very quickly. During the process of creating $[\![q]\!]$, usually several list axioms are created. This set of axioms is referred to as $Th(q)$. In particular, in this case $Th(q)$ includes the axioms for the map, filter, and cross product function symbols that occur in $[\![q]\!]$.

Aggregates. Aggregates are used to combine values from a group of rows in a table. The most common aggregates are `MIN`, `MAX`, `SUM`, and `COUNT`. For example, the following query q_1 selects the maximum points from the Scores table.

```
SELECT MAX(Points) from Scores
```

Depending on the use of q_1, the translation $[\![q_1]\!]$ is either the singleton list:

$$cons(T(Reduce[Ite(x_0.2 \geq x_1, x_0.2, x_1)](Scores, MinValue(\mathbb{Z}))), nil)$$

or just the $Reduce[]$-term:

$$Reduce[Ite(x_0.2 \geq x_1, x_0.2, x_1)](Scores, MinValue(\mathbb{Z}))$$

The first case applies if q_1 is used as a top-level query, the second case applies if q_1 is used as a *subquery expression*. The second case applies in the following query q_2 that also uses the `MAX` aggregate in the top level select list in combination with `GROUP BY` that eliminates duplicates from the resulting table:

```
SELECT StudentID, MAX(Points) FROM Scores GROUP BY StudentID
HAVING MAX(Points) = (SELECT MAX(Points) from Scores)
```

The query q_2 selects all students that have the most points at some course. The translation of $[\![q_2]\!]$ is as follows where the $Filter[]$ application corresponds to the HAVING clause that is applied to the result of the grouping.

$$Filter[x.1 = [\![q_1]\!]](RemoveDuplicates(Select^{MAX}[x_0.0, x_0.3](Scores)))$$

LIKE-patterns. Like-patterns are particular regular expressions that can be used as constraints on strings. A like-pattern r is converted into a *symbolic finite automaton* [27] (SFA) A_r that is similar to a classical finite automaton except that moves are labeled by formulas denoting *sets* of characters rather than single characters. The full expressiveness of patterns r that is currently supported by the conversion A_r is that of .NET regexes (except for anchors `\G`, `\b`, `\B`, named groups, lookahead, lookbehind, as-few-times-as-possible quantifiers, backreferences, conditional alternation, and substitution).

The automaton A_r is translated into a theory $Th(A_r)$. The theory describes the acceptance condition for words in $L(A_r)$. In particular, $Th(A_r)$ defines a predicate

$$Acc^{A_r} : \mathbb{W} \times \mathbb{N} \to \mathbb{B},$$

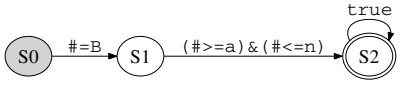
where $\mathbb{N}$ is an algebraic datatype for *unary natural numbers* with the constructors $\overline{0} : \mathbb{N}$ and $\mathbf{s} : \mathbb{N} \to \mathbb{N}$. We write $\overline{k+1}$ for $\mathbf{s}(\overline{k})$. Intuitively, $Acc^{A_r}(t, \overline{k})$ expresses that t is a word of at most k characters that matches the pattern r. We use the following property of the theory of A_r [27, Theorem 1]:

Proposition 2. *Let t be a closed term of sort $\mathbb{W}$, k a nonnegative integer, and M a model of $Th(A_r)$. Then $M \models Acc^{A_r}(t, \overline{k})$ iff $t^M \in L(A_r)$ and $|t^M| \leq k$.*

In column type declarations of SQL database schemas, a maximum string length is associated with the `char` type (default being 1), e.g., the type `char(100)` of a column allows strings containing at most 100 characters. In the formula $Acc^{A_r}(t, \overline{k})$, where t refers to a column whose values are strings, k is the maximum length of the strings in that column.

Example 2. Consider the query q that selects students whose name starts with the letter `B` followed by any letter between `a` and `n` followed by 0 or more additional characters:

```
SELECT StudentName FROM Students WHERE StudentName like "B[a-n]%"
```

The SFA A for `"B[a-n]%"` is 

where # is a free variable of sort $\mathbb{C}$ and each symbolic move $(i, \varphi[\#], j)$ denotes the set of transitions $\{(i, a, j) \mid a \in \mathcal{U}^{\mathbb{C}}, \models \varphi[a]\}$. For each state *S0*,*S1*,*S2* of A there are two axioms in $Th(A)$, one for length bound $= 0$ and the other one for length bound > 0:

$$\begin{aligned}
&S0 : \forall x\,(Acc(x, \overline{0}) \Leftrightarrow \mathit{false})\\
&\quad \forall x\,y\,(Acc(x, \mathbf{s}(y)) \Leftrightarrow x \neq \mathit{nil} \wedge hd(x) = \mathtt{B} \wedge Acc_1(tl(x), y))\\
&S1 : \forall x\,(Acc_1(x, \overline{0}) \Leftrightarrow \mathit{false})\\
&\quad \forall x\,y\,(Acc_1(x, \mathbf{s}(y)) \Leftrightarrow x \neq \mathit{nil} \wedge hd(x) \geq \mathtt{a} \wedge hd(x) \leq \mathtt{n} \wedge Acc_2(tl(x), y))\\
&S2 : \forall x\,(Acc_2(x, \overline{0}) \Leftrightarrow x = \mathit{nil})\\
&\quad \forall x\,y\,(Acc_2(x, \mathbf{s}(y)) \Leftrightarrow x = \mathit{nil} \vee (x \neq \mathit{nil} \wedge Acc_2(tl(x), y)))
\end{aligned}$$

The term $[\![q]\!]$ is $Map[T(x_0.1)](Filter[Acc(x_0.1, \overline{100})](Students))$. Note that $Th(A)$ is a subset of $Th(q)$.

The automata based approach opens up several transformation techniques that can be performed in the process of encoding queries and theories of queries that involve like-patterns. These upfront transformations can greatly simplify the formulas. We illustrate this with an example involving the use of *product* of SFAs. The following proposition follows directly from the product definition (see [27]).

Proposition 3. *Let A and B be SFAs, then $L(A \otimes B) = L(A) \cap L(B)$.*

Example 3. Consider the following query q_{LIKE} with $n+1$ occurrences of "_" in the first like-pattern and n occurrences of "_" in the second like-pattern:

```
SELECT StudentName FROM Students
WHERE StudentName like "%a_____" AND StudentName like "%b____"
```

The first like-pattern corresponds to the regex r_1=`.*a.{n+1}` and the second like-pattern corresponds to the regex r_2=`.*b.{n}`. The query is essentially an intersection constraint of r_1 and r_2. In a direct encoding of q_{LIKE}, $Th(q_{\text{LIKE}})$ includes both the axioms for A_{r_1} as well as A_{r_2}. Rather than using A_{r_1} and A_{r_2} separately, the product $A_{r_1} \otimes A_{r_2}$ of A_{r_1} and A_{r_2} can be used together with the theory $Th(A_{r_1} \otimes A_{r_2})$ instead of $Th(A_{r_1}) \cup Th(A_{r_2})$. Thus, with *product encoding*,

$$[\![q_{\text{LIKE}}]\!] = Map[T(x_0.1)](Filter[Acc^{A_{r_1} \otimes A_{r_2}}(x_0.1, \overline{100})](Students))$$

and with *direct encoding*,

$$[\![q_{\text{LIKE}}]\!] = Map[T(x_0.1)](Filter[Acc^{A_{r_1}}(x_0.1, \overline{100}) \wedge Acc^{A_{r_2}}(x_0.1, \overline{100})](Students))$$

The gain in performance is discussed in Section 6.

Note that correctness of the transformation illustrated in Example 3 follows from Propositions 2 and 3.

5 Implementation

Qex uses the SMT solver Z3 [30,10]. Interaction with Z3 is implemented through its programmatic API rather than using a textual format, such as the smt-lib format [24]. The main reasons for working with the API are: access to built-in data types; model generation; working within a given context. The first point is fundamental, since algebraic data types are central to the whole approach and are not part of the smt-lib standard.

Besides allowing to check satisfiability, perhaps the most important feature exposed by some SMT solvers (including Z3) for the purposes of test input generation is *generating a model* as a witness of the satisfiability check, i.e., a mapping of the uninterpreted function symbols to their interpretations. Z3 has a separate method for satisfiability checking with model generation. This code snippet illustrates the use of that functionality:

```
Model m;
z3.AssertCnstr(f);
LBool sat = z3.CheckAndGetModel(out m);
Term v = m.Eval(s); ...
```

A *context* includes declarations for a set of symbols, and assertions for a set of formulas. A context is essentially a layering mechanism for signature expansions with related constraints. There is a *current context* and a backtrack stack of previous contexts. Contexts can be saved through *pushing* and restored through *popping*. When a satisfiability check is performed in a given context, the context may become inconsistent. Qex uses contexts during table generation and in SFA algorithms during theory generation for like-patterns.

5.1 Incremental Table Generation

Let q be a fixed query and assume that $X_1, \ldots, X_n$ are the input table variables. Assume $[\![q]\!] : \sigma$ and let $\varphi[Y]$ be a formula with the free variable $Y : \sigma$. Intuitively, φ is a *test condition* on the result of the query, e.g. $Y \neq nil$. The following *basic table generation procedure* describes the input table generation for q and φ.

1. Assert $Th(q)$, i.e. add the axioms of q to the current context.
2. Let $\boldsymbol{k} = (k_1, \ldots, k_n) = (1, \ldots, 1)$ be the initial sizes of the input tables. Repeat the following until a model M is found or a timeout occurs.
   (a) Push the current context, i.e., create a backtrack point.
   (b) Create constraints for $X_1, \ldots, X_n$ using $\boldsymbol{k}$ to fix the table sizes.
   (c) Assert $\varphi[[\![q]\!]]$
   (d) Check and get the model M. If the check fails, increase $\boldsymbol{k}$ systematically (e.g., by using a variation of Cantor's enumeration of rationals), and pop the context.
3. Get the values of $X_1, \ldots, X_n$ in M.

There are several possible variations of the basic procedure. The table constraints can be updated incrementally when the table sizes are increased. The table constraints can also be created for *upper* bounds rather than exact bounds on the table sizes. One way to do so is as follows:

$$table = cons(row1, rest1) \wedge (rest1 = nil \vee (rest1 = cons(row2, rest2) \wedge \cdots))$$

The size of the overall resulting formula is always polynomial in the size of the original query and $\boldsymbol{k}$. In practice, Qex uses bounds on $\boldsymbol{k}$ and overall time constraints to guarantee termination, as deciding the satisfiability of queries is undecidable in general [9].

5.2 Symmetry Breaking Formulas

The translation of a query q into a formula $[\![q]\!]$ together with $Th(q)$ and the additional table constraints looks very much like a "functional program with

constraints". This intuition is correct as far as the logical meaning of the translation is concerned. There are, however, no mechanisms to control the evaluation order of patterns (such as, "outermost first") and no notion of term orderings. The search space for $[\![q]\!]$ is typically vast.

Recall that although Qex uses lists to encode tables, the order of rows is not relevant according to the SQL semantics. We can therefore assert predicates that constrain the input tables to be *ordered* (thus eliminating all symmetrical models where the ordering does not hold). Consider a table

$$cons(row_0, cons(row_1, \cdots cons(row_n, nil)))$$

of sort $\mathbb{L}\langle\sigma\rangle$. Define a lexicographic order predicate $\preceq\ : \sigma \times \sigma \to \mathbb{B}$. The definition of $\preceq$ on integers is just the built-in order $\leq$, similarly for bitvectors. For tuples, it is the standard lexicographic order defined in terms of the orders of the respective element sorts. For strings (lists of bitvectors) the order predicate can be defined using recursion over lists. Assert the *symmetry breaking formula*

$$\bigwedge_{i<n-1} row_i \preceq row_{i+1}$$

In some situations the symmetry breaking formula can be strengthened. For example, when the table has a primary key then the formula can be strengthened by using the strict order $\prec$ instead of $\preceq$. Moreover, if all of the columns are part of the primary key then the primary key constraint itself becomes redundant.

6 Experiments

We provide some performance evaluation results of Qex on a collection of sample queries.[1] In the first set of experiments we look at the performance of the basic table generation procedure. In these experiments we use the same bound k for both tables. The test condition used here is that the result is nonempty. Table 2 summarizes the overall time t (in ms) for each query q, which includes the parsing time, the generation time of $Th(q)$, and the model generation time. (Note that query #3 is a valid SQL query without a group-by clause.) The column k shows the number of rows generated for the input tables. Some of the queries include parameters, indicated with `@`, the values of parameters are also generated. (The actual data that was generated is not shown here.) We reuse the schema introduced in Example 1. The last query uses an additional table called `Courses` with the schema:

```
CREATE TABLE [School].Courses
 (CourseNr tinyint not null PRIMARY KEY, CourseName char(15) not null);
```

Using symmetry breaking over lists did not improve the performance for these examples. In some cases it had the opposite effect, e.g., for query #3 the time

[1] The experiments were run on a Lenovo T61 laptop with Intel dual core T7500 2.2GHz processor.

Table 2. Sample queries

#	Query	t[ms]	k

1	`DECLARE @x as tinyint;`
	`SELECT Scores.StudentID, SUM(Scores.Points)`
	`FROM Scores`
	`WHERE Scores.Points > 2`
	`GROUP BY Scores.StudentID`
	`HAVING SUM(Scores.Points) >= @x AND @x > 5`
		20	1

2	`SELECT Scores.StudentID, MAX(Scores.Points)`
	`FROM Scores`
	`GROUP BY Scores.StudentID`
	`HAVING MAX(Scores.Points) = (SELECT MAX(Scores.Points) FROM Scores)`
		20	1

3	`DECLARE @x as tinyint;`
	`SELECT COUNT(S.StudentName)`
	`FROM Students as S`
	`WHERE S.StudentName LIKE "%Mar[gkc]us%" AND S.StudentNr > @x`
	`HAVING COUNT(S.StudentName) > @x AND @x > 2`
		1300	4

4	`DECLARE @x as tinyint;`
	`SELECT Students.StudentName, SUM(Points)`
	`FROM Scores JOIN Students ON Scores.StudentID = Students.StudentNr`
	`WHERE Scores.Points > 2 AND Students.StudentName LIKE "John%"`
	`GROUP BY Students.StudentName`
	`HAVING SUM(Points) >= @x AND @x > 15`
		200	2

5	`SELECT Students.StudentName, Scores.Points`
	`FROM Students JOIN Scores ON Scores.StudentID = Students.StudentNr`
	`WHERE Scores.CourseID = 10 AND Scores.Points > 0`
		30	1

6	`SELECT Students.StudentName, Courses.CourseName, Scores.Points`
	`FROM Scores JOIN Students ON Scores.StudentID = Students.StudentNr`
	`JOIN Courses ON Courses.CourseNr = Scores.CourseID`
	`WHERE Scores.Points > 2 AND Students.StudentName LIKE "bob%"`
	`AND Courses.CourseName LIKE "AI"`
		80	1

increase is 30%. Although symmetry breaking was crucial for the examples in [28], here the benefits are unclear. If we consider the test condition that the result has 4 rows, and also that the input tables all have 9 rows then, for query #6 the total time to generate the three input tables is 75s without symmetry breaking and 45s with symmetry breaking. However, in general it seems that the ordering constraints on strings are expensive. At this point we do not have enough experience to draw clear conclusions when it pays off to use them.

The total size of the query seems to have very little effect on the time t. The key factor is the use of *aggregates* and the constraints they introduce that cause the input tables to grow, thus, causing backtracking during model generation, that is clearly seen for query #3. Consider the following experiment. Take query #3 and replace the constant 2 in it with the constant n for $n = 1, \ldots, 15$. Figure 1 shows the time t in seconds as a function of n; k is always $n + 2$.

Given a query q, several optimizations or transformations can be performed on the term $[\![q]\!]$ as well as the set of axioms $Th(q)$ prior to asserting them to the solver. Figure 2 shows a drastic decrease in model generation time for q_{LIKE} from Example 3 in Qex when the product construction is used. By performing localized SMT solver queries during product construction of SFAs, the size of the resulting automata can often dramatically decrease. We have experimented with a few special cases of this nature, but have not systematically applied such

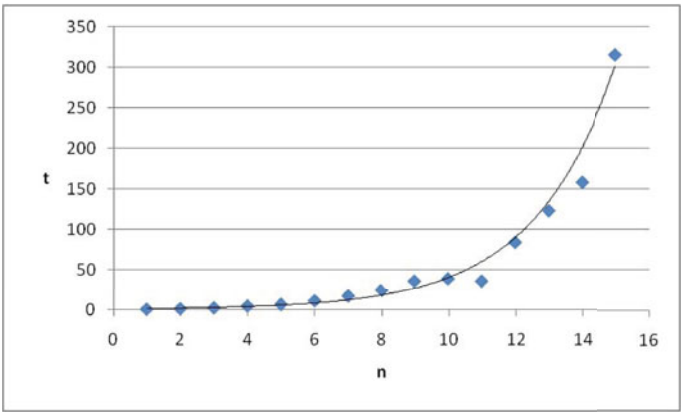


Fig. 1. Exploration times (sec) for query #3 in Table 2 when the constant 2 is replaced with n for $n = 1, \ldots, 15$

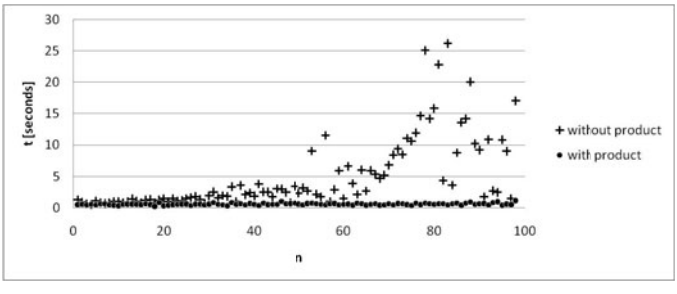


Fig. 2. Exploration times (sec) for query q_{LIKE} without (scattered crosses) and with (solid line of dots at the bottom) product construction for $n = 1, \ldots, 98$

transformations or other transformations such as combining several consecutive filters as a single filter.

We also reevaluated the performance of Qex on the benchmarks reported in [28, Table 1] that use a different sample database schema (where strings do not occur). In all of the cases the performance improvement was between 10x and 100x. As we suspected, the eager expansion time reported as t_{exp} in [28], that was by an order of magnitude larger than the model generation time t_{z3}, is avoided completely in the new approach. The initial cost of creating $[\![q]\!]$ is negligible, since the size of $[\![q]\!]$ is polynomial in the size of q in theory, and close to linear in practice. The added overhead during model generation due to the use of axioms only marginally increased the model generation time t_{z3}.

The final example illustrates an application of the tool on normal form analysis of schemas.

Example 4. Consider the following additional schema of a table `Learning`.

```
CREATE TABLE [School].Learning
 (Student TINYINT NOT NULL, Course TINYINT NOT NULL,
  Teacher TINYINT NOT NULL,
  PRIMARY KEY (Student, Course),
  UNIQUE (Student, Teacher));
```

It is easy to see that the table satisfies 3NF (3rd normal form) since all attributes are prime (belong to a candidate key of `Learning`). Suppose that there is a functional dependency `Teacher` $\rightarrow$ `Course`. One can show that the table does not satisfy BCNF (Boyce-Codd normal form, it is a slightly stronger version of 3NF), where for each functional dependency $X \rightarrow Y$, X must be a superkey (i.e. a candidate key or a superset of a candidate key). One can show that `Teacher` is not a superkey of the `Learning` table by showing that the following query can yield a nonempty answer:

```
SELECT X.Teacher
FROM Learning AS X JOIN Learning AS Y ON (X.Teacher = Y.Teacher)
WHERE X.Course < Y.Course;
```
(2)

The basic table generation procedure for (2) provides the following solution for `Learning`:

Student	Course	Teacher
0	69	133
1	70	133

The following query can be used to serve the same purpose:

```
SELECT Teacher, COUNT(Course)
FROM Learning
GROUP BY Teacher HAVING COUNT(Course) > 1;
```
(3)

For both queries (2) and (3) the total execution time is around 20 milliseconds: we repeated the experiment for (2) 100 times with total execution time of 2 seconds, and we repeated the experiment with (3) 100 times as well, with total execution time of 2 seconds also. The actual model generation time for a single run was around 10 milliseconds for both queries, the rest of the time was due to the overhead of the startup, file handling and parsing. However, by changing query (3) slightly, by replacing `COUNT(Course) > 1` by `COUNT(Course) >` n, for $n > 1$, one can detect exponential increase in model generation time: for $n = 4; 5; 6$ the experiment took $0.1; 0.2; 0.4$ seconds.

7 Related Work

The first prototype of Qex was introduced in [28]. The current paper presents a continuation of the Qex project [23], and a redesign of the encoding of queries into formulas based on a lazy axiomatic approach that was briefly mentioned

in [28] but required support for algebraic data types in the underlying solver. Moreover, Qex now also supports a substantially larger fragment of SQL (such as subquery expressions) and like-patterns on strings, as discussed above.

Deciding satisfiability of SQL queries requires a formal semantics. While we give meaning to SQL queries by an embedding into the theory of an SMT solver, there are other approaches, e.g., defining the semantics in the Extended Three Valued Predicate Calculus [19], or using bags as a foundation [7]. Satisfiability of queries is also related to logic-based approaches to semantic query optimization [5]. The general problem of satisfiability of SQL queries is undecidable and computationally hard for very restricted fragments, e.g., deciding if a query has a nonempty answer is NEXP-hard for nonrecursive range-restricted queries [9].

Several research efforts have considered formal analysis and verification of aspects of database systems, usually employing a possibly interactive theorem prover. For example, one system [25] checks whether a transaction is guaranteed to maintain integrity constraints in a relational database; the system is based on Boyer and Moore-style theorem proving [4].

There are many existing approaches to generate database tables as test inputs. Most approaches create data in an ad-hoc fashion. Only few consider a target query. Tsai et.al. present an approach for test input generation for relational algebra queries [26]. They do not use lists to represent tables. They propose a translation of queries to a set of systems of linear inequalities, for which they implemented an ad-hoc solving framework which compares favorably to random guessing of solutions. A practical system for testing database transactions is AGENDA [12]. It generates test inputs satisfying a database schema by combining user-provided data, and it supports checking of complex integrity constraints by breaking them into simpler constraints that can be enforced by the database. While this system does not employ a constraint solver, it has been recently refined with the TGQG [6] algorithm: Based on given SQL statements, it generates test generation queries; execution of these queries against a user-provided set of data groups yields test inputs which cover desired properties of the given SQL statements.

Some recent approaches to test input generation for databases employ automated reasoning. The relational logic solver Alloy [16,17] has been used by Khalek et.al. [18] to generate input data for database queries. Their implementation supports a subset of SQL with a simplified syntax. In queries, they can reason about relational operations on integers, equality operations on strings, and logical operations, but not about nullable values, or grouping with aggregates such as `SUM`; they also do not reason about duplicates in the query results. QAGen [3] is another approach to query-solving. It first processes a query in an adhoc-way, which requires numerous user-provided "knob" settings as additional inputs. From the query, a propositional logic formula is generated, which is then decided by the Cogent [8] solver to generate the test inputs. In [2] a model-checking based approach, called Reverse Query Processing, is introduced that, given a query and a result table as input, returns a possible database instance that could have produced that result for that query, the approach uses

reverse relational algebra. In [29] an intentional approach is presented in which the database states required for testing are specified as constrained queries using a domain specific language. Recently, test input generation of queries has been combined with test input generation of programs that contain embedded queries in the program text [13], using ad-hoc heuristic solvers for some of the arising constraints from the program and the queries.

Generating sample input data for databases is related to generating sample data for dataflow programs, the work in [20] discusses input data generation for Pig Latin [21], developed at Yahoo! Research, that is a query language in between SQL and the mapreduce [11] programming model. The approach in [20] focuses on certain core aspects of Pig Latin that can also handle aggregation through GROUP and TRANSFORM constructs of the language. The algorithm in [20] does not use off-the-shelf tools or symbolic analysis techniques but is a stand-alone multi-pass dataflow analysis algorithm. It is unclear as to how the approach can be combined with additional constraints, for example arithmetical constraints or string constraints in form of regular patterns.

8 Conclusion and Future Work

The current implementation of the Qex project is still in its early stages, but we were highly encouraged by the performance improvements when switching to the lazy approach and reducing the need for nonlinear constraints through a different representation of tables. There are many more possible optimizations that can be performed as a preprocessing step on formulas generated by Qex, before asserting them to the SMT solver. One such optimization, using automata theory, was illustrated in Example 3 and Figure 2 when multiple like-patterns occur in a query. Systematic preprocessing can also often reveal that a query is trivially false, independent of the size of input tables, e.g., if an '_' is missed in the first like-pattern in Example 3 then the product automaton would be empty.

For practical usage in an industrial context, where SQL queries are usually embedded in other programs or in store procedures, we are looking at integrating Qex in Pex [22]. For efficient support for regex constraints in Pex, integration of Rex [27] is a first step in that integration.

It is also possible to apply a translation similar to the one described in the paper to LINQ queries, although, unlike in SQL, the semantics of LINQ queries depends on the order of the rows in the tables. This fits well with the list representation of tables but imposes some limitations on the use of certain optimizations (such as the use of symmetry breaking formulas).

A practical limitation of Qex is if queries use multiple joins and aggregates and the input tables need to contain a high number of rows in order to satisfy the test condition. Another limitation is the use nonlinear constraints over unbounded integers, in particular multiplication, that has currently only limited support in Z3. We consider using bitvectors instead. Despite these limitations, the mere size of queries does not seem to be a concern, neither the size of $Th(q)$ for a given query q. The size of $Th(q)$ may easily be in hundreds, in particular when several

like-patterns are used, where the number of axioms is proportional to the size of the finite automaton accepting the pattern.

Acknowledgements

We thank Nikolaj Bjørner for the continuous support with Z3 and for suggesting the idea of symmetry breaking formulas. We also thank the reviewers for many useful comments and for pointing out some related work that we were not aware of. One reviewer provided very detailed and constructive comments, up to an extent that could under different circumstances warrant coauthorship, in particular, suggested to formulate Proposition 1, provided a more accurate handling of null, corrected (essentially rewrote) the axioms in Section 3.2 (the original description was in an appendix, and besides having several typos, could not handle null), and provided the schema normalization Example 4 as an intuitive and different application of the techniques presented in the paper.

References

1. SELECT (T-SQL), http://msdn.microsoft.com/en-us/library/ms189499.aspx
2. Binnig, C., Kossmann, D., Lo, E.: Reverse query processing. In: Proceedings of the 23rd International Conference on Data Engineering (ICDE 2007), pp. 506–515. IEEE, Los Alamitos (2007)
3. Binnig, C., Kossmann, D., Lo, E., Özsu, M.T.: Qagen: generating query-aware test databases. In: SIGMOD 2007: Proceedings of the 2007 ACM SIGMOD international conference on Management of data, pp. 341–352. ACM, New York (2007)
4. Boyer, R.S., Moore, J.S.: A computational logic handbook. Academic Press Professional, Inc., San Diego (1988)
5. Chakravarthy, U.S., Grant, J., Minker, J.: Logic-based approach to semantic query optimization. ACM Trans. Database Syst. 15(2), 162–207 (1990)
6. Chays, D., Shahid, J., Frankl, P.G.: Query-based test generation for database applications. In: Proceedings of the 1st International Workshop on Testing Database Systems (DBTest 2008), pp. 1–6. ACM, New York (2008)
7. Chinaei, H.R.: An ordered bag semantics of SQL. Master's thesis, University of Waterloo, Waterloo, Ontario, Canada (2007)
8. Cook, B., Kroening, D., Sharygina, N.: Cogent: Accurate theorem proving for program verification. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 296–300. Springer, Heidelberg (2005)
9. Dantsin, E., Voronkov, A.: Complexity of query answering in logic databases with complex values. In: Adian, S., Nerode, A. (eds.) LFCS 1997. LNCS, vol. 1234, pp. 56–66. Springer, Heidelberg (1997)
10. 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)
11. Dean, J., Ghemawat, S.: MapReduce: a flexible data processing tool. Commun. ACM 53(1), 72–77 (2010)
12. Deng, Y., Frankl, P., Chays, D.: Testing database transactions with AGENDA. In: ICSE 2005, pp. 78–87. ACM, New York (2005)

13. Emmi, M., Majumdar, R., Sen, K.: Dynamic test input generation for database applications. In: Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA 2007), pp. 151–162. ACM, New York (2007)
14. Grust, T., Scholl, M.H.: How to comprehend queries functionally. Journal of Intelligent Information Systems 12(2-3), 191–218 (1999)
15. Hodges, W.: Model theory. Cambridge Univ. Press, Cambridge (1995)
16. Jackson, D.: Automating first-order relational logic. SIGSOFT Softw. Eng. Notes 25(6), 130–139 (2000)
17. Jackson, D.: Software Abstractions. MIT Press, Cambridge (2006)
18. Khalek, S.A., Elkarablieh, B., Laleye, Y.O., Khurshid, S.: Query-aware test generation using a relational constraint solver. In: ASE, pp. 238–247 (2008)
19. Negri, M., Pelagatti, G., Sbattella, L.: Formal semantics of SQL queries. ACM Transactions on Database Systems 17(3), 513–534 (1991)
20. Olston, C., Chopra, S., Srivastava, U.: Generating example data for dataflow programs. In: SIGMOD, pp. 245–256. ACM, New York (2009)
21. Olston, C., Reed, B., Srivastava, U., Kumar, R., Tomkins, A.: Pig latin: a not-so-foreign language for data processing. In: SIGMOD, pp. 1099–1110. ACM, New York (2008)
22. Pex, http://research.microsoft.com/projects/pex
23. Qex, http://research.microsoft.com/projects/qex
24. Ranise, S., Tinelli, C.: The SMT-LIB Standard: Version 1.2. Technical report, Department of Computer Science, The University of Iowa (2006), www.SMT-LIB.org
25. Sheard, T., Stemple, D.: Automatic verification of database transaction safety. ACM Trans. Database Syst. 14(3), 322–368 (1989)
26. Tsai, W.T., Volovik, D., Keefe, T.F.: Automated test case generation for programs specified by relational algebra queries. IEEE Trans. Softw. Eng. 16(3), 316–324 (1990)
27. Veanes, M., de Halleux, P., Tillmann, N.: Rex: Symbolic regular expression explorer. In: Cavalli, A., Ghosh, S. (eds.) Third International Conference on Software Testing, Verification and Validation (ICST 2010), Paris, France, IEEE, Los Alamitos (April 2010)
28. Veanes, M., Grigorenko, P., de Halleux, P., Tillmann, N.: Symbolic query exploration. In: Breitman, K., Cavalcanti, A. (eds.) ICFEM 2009. LNCS, vol. 5885, pp. 49–68. Springer, Heidelberg (2009)
29. Willmor, D., Embury, S.M.: An intensional approach to the specification of test cases for database applications. In: 28th International Conference on Software Engineering, pp. 102–111 (2006)
30. Z3, http://research.microsoft.com/projects/z3

Automated Proof Compression by Invention of New Definitions

Jiří Vyskočil[1,⋆], David Stanovský[2,⋆⋆], and Josef Urban[3,⋆⋆⋆]

[1] Czech Technical University in Prague
[2] Charles University in Prague
[3] Radboud University, Nijmegen

Abstract. State-of-the-art automated theorem provers (ATPs) are today able to solve relatively complicated mathematical problems. But as ATPs become stronger and more used by mathematicians, the length and human unreadability of the automatically found proofs become a serious problem for the ATP users. One remedy is automated proof compression by invention of new definitions.

We propose a new algorithm for automated compression of arbitrary sets of terms (like mathematical proofs) by invention of new definitions, using a heuristics based on substitution trees. The algorithm has been implemented and tested on a number of automatically found proofs. The results of the tests are included.

1 Introduction, Motivation, and Related Work

State-of-the-art automated theorem provers (ATPs) are today able to solve relatively complicated mathematical problems [McC97], [PS08], and are becoming a standard part of interactive theorem provers and verification tools [MP08], [Urb08]. But as ATPs become stronger and more used by mathematicians, understanding and refactoring the automatically found proofs becomes more and more important.

There is a number of examples, and significant amount of more or less successful relevant work in the field of formal proof refactoring. The most well-known example is the proof of the Robbins conjecture found automatically by EQP. This proof has been semi-automatically simplified by the ILF system [Dahn98], and later also rewritten as a Mizar formalization [Gra01]. Other examples include the refactoring of the proof of the Four Color Theorem by Gonthier [Gon07], the hint strategy used regularly to simplify the proofs found automatically by the Prover9 system [Ver01], translation of resolution proofs into assertion level natural deduction proofs [Hua96], various utilities for formal proof refactoring in the Mizar

⋆ Supported by the institutional grant MSM 6840770038.
⋆⋆ Supported by the institutional grant MSM 0021620839 and by the GAČR grant 201/08/P056.
⋆⋆⋆ Supported by the NWO project "MathWiki a Web-based Collaborative Authoring Environment for Formal Proofs".

E.M. Clarke and A. Voronkov (Eds.): LPAR-16, LNAI 6355, pp. 447–462, 2010.

system, and visualization of proofs and their compactification based on various interestingness criteria in the IDV and AGiNT systems [TPS07], [PGS06]. Introduction of definitions is a common part of state-of-the-art first-order ATPs, used to compute efficient clause normal forms [NW01]. Introduction of definitions is also an important part of unfold-definition-fold transformation in logic programming[1], the main purpose there is usually speed-up of the logic programs (reducing number of computation steps).

The work presented here tries to help understanding of formal proofs by automated finding of repeated patterns in the proofs, suggesting new definitions that capture the patterns and shorten the proofs, and help to develop a structured theory. We believe that this approach might not only help mathematicians to better understand the long automatically found proofs, but also that following the recent experiments with meta-systems for automated reasoning in large structured theories [USPV08] this approach could provide another way to attack hard problems automatically by enriching the theory first with new concepts, and smart heuristic abstracting away ("forgetting about") some of the concepts' properties irrelevant for the particular proof. The last mentioned is probably not the only machine-oriented application of proof compactification: compact proofs are likely to be more easy to verify, and also to combine and transform automatically in various ways.

The structure and content of this paper is as follows. Section 2 formally describes the approach used for proof compression by invention of new definitions. In Section 3 an efficient heuristic algorithm for finding best definitions based on substitution trees is suggested and its first implementation is described. In Section 4 the implementation is evaluated on ca. 8000 proofs from the TPTP library, and on several algebraic proofs. In Section 5 several examples demonstrating the work of the algorithm are shown and discussed. Section 6 discusses the possible extensions, improvements, testing, and uses of this approach, and concludes.

2 Problem Statement

As mentioned above, the problem of proof improvement and refactoring is quite wide, and it can be attacked by different methods, and by employing different criteria.

The motivation for the approach taken here is that given the original proof, it can contain a large number of "similar" complex terms (terms with a large weight). Mathematicians would typically quickly simplify such proofs by introducing suitable new concepts and notation. While it is nontrivial to tell what exactly makes a new definition mathematically plausible and worth introducing in a certain proof or theory, there is at least one well-defined and easy-to-use measure of the "plausibility" of a new definition, namely the degree in which it reduces weight of the particular proof. The problem then is to find the definitions

[1] It was firstly well defined in [TS84] then extended in [PP95] and also automated in [VV07].

that suitably generalize the largest number of similar terms, or more precisely, to find definitions that have the best value in terms of decreasing the overall weight of the proof after replacing terms with the newly defined symbol.

The precise definition of the problem is as follows.

2.1 Problem of Proof Compression by New Definitions

Proof: A formal mathematical *proof* is understood generally, as a sequence (list, or DAG, tree, etc.) of formulae (or sequents, or just arbitrary Prolog terms) connected by inference rules. The inference rules are not relevant for the initial approach, only the formulae matter. For the purpose of this work, it suffices to treat proofs as (a set of) arbitrary Prolog terms over some initial signature of symbols (e.g., predicate and function symbols used in the proof, and logical connectives). Particular instance of this approach are the first-order proofs written in the TPTP language[2], which are just sequences of first-order TPTP formulae and clauses (written as Prolog terms) annotated with their inference information. The input data for our algorithm are then just the formulae and clauses (set of Prolog terms), without the inference data.[3]

Weight: A *weight assignment* w is a function from the proof signature to integers, together with an integer value for variables. The *weight* of a proof signature symbol or variable is equal to the value of the weight assignment function on it. The weight of a term, formula, or proof is a sum of the weights of the symbols and variables contained in it. The weights of symbols, terms, formulae, and proofs under a particular weight assignment w are denoted $w(s)$, $w(t)$, $w(F)$, $w(P)$. Unless specified otherwise, the simple "symbol-counting" weight assignment giving value 1 to all symbols and variables will be used as default in what follows.

New definition: Given a proof (Prolog term) P, a *new definition* D *wrt* P is a binary Prolog clause D of the form

$$s(X_1, ..., X_n) :- T(X_1, ..., X_n) \qquad (D)$$

where s is a symbol not appearing in P, $T(X_1, ..., X_n)$ is a Prolog term over the signature of P, and $0 \leq n$. Note that this approach does not allow recursive definitions, and does not allow new variables in the body of the definition (otherwise

[2] http://www.cs.miami.edu/~tptp/TPTP/SyntaxBNF.html

[3] Note that the bound variables in TPTP first-order formulae are represented by Prolog variables in the TPTP syntax, however, these variables are not really "free" in the Prolog (and also first-order) sense. A proper treatment for our algorithm would be to e.g., rename such bound variables to de Bruijn indices, however the first version of our algorithm does not do this. This treatment is suboptimal, in the sense that a definition with a redundant variable can be introduced, however this has no impact on the correctness of the algorithm.

the *most compressing definition* problem below becomes Turing-complete). Unless specified otherwise, we will also require that with a given weight assignment w, the definition D satisfies the strict monotonicity condition

$$w(s(X_1, ..., X_n)) < w(T(X_1, ..., X_n))$$

Definition application at a position: When S is a term matching the body of the definition D (ie., there is a substitution σ such that $T(X_1, ..., X_n)\sigma = S$), then $D(S)$ will denote the replacing of S by the appropriately instantiated head of the definition (ie., $s(X_1, ...X_n)\sigma$, where σ is as above). Similarly, $D(P|_p)$ will denote the (unique) replacement of the subterm at position p in a term P.

Exhaustive definition application on the whole term: Now consider the following definition

$$s(X) :\!- f(f(X)) \tag{D_1}$$

and the term

$$f(f(f(a))) \tag{P_1}$$

Then D_1 can be applied either at the topmost position, yielding $s(f(a))$, or at the first subterm, yielding $f(s(a))$. However simultaneous application at both positions is not possible. In both cases, the default weight of the original term decreased from 4 to 3. Then consider the term

$$f(f(f(f(a)))) \tag{P_2}$$

The first application of D_1 can now be done at three different positions, yielding $s(f(f(a)))$, $f(s(f(a)))$, and $f(f(s(a)))$. For the first and third result, D_1 can be applied again, yielding $s(s(a))$ with weight 3 in both cases, while the second result with weight 4 cannot be further reduced using D_1. Hence the order of application of the definitions matters. The notation $D^*(P)$ will therefore denote any of the (possibly many and different) exhaustive applications of definition D to term P, i.e., $D^*(P)$ is a term where D can no longer be applied at any position. $D^*_{min_w}(P)$ (or just $D^*_{min}(P)$ when the weight assignment is fixed) will denote those exhaustive applications (again, generally many) such that the weight of the resulting term is minimal. Note (on terms P_1 and P_2 and definition D_1) that $D^*(P)$ and $D^*_{min}(P)$ are not unique, and can be obtained by different application paths, however in what follows we will be interested mostly only in the minimal weight and irreducibility by D.

The proof compression problems: There are several well-defined problems in this setting. The *most compressing definition* problem is, for a given proof P to find the new definition D wrt to P that compresses the proof most, i.e., $w(D) + w(D^*_{min}(P))$ is minimal across all possible definitions D. Since $D^*_{min}(P)$

is non-deterministic, in practice this problem also includes finding the particular sequence of applications of D to P that result in a particular $D^*_{min}(P)$.

The *greatest proof compression* problem is, to find a set of definitions $D_1, \ldots, D_n$ and a sequence of their combined applications $D^*_{1..n}(P)$ such that $w(D_1)+\cdots+w(D_n)+w(D^*_{1..n}(P))$ is minimal wrt to a given proof P and weight assignment w. Let us again denote by $D^*_{1..n_{min}}(P)$ the sequences of definition applications for which this final measure is minimal. In this setting, the definitions can have in their bodies the symbols newly introduced by earlier definitions, however mutual recursivity is not possible, because the definitions applied first cannot refer to the symbols introduced later.

There are two ("greedy") ways to make the general greatest proof compression problem simpler and efficiently implementable. The first simplification consists in restricting the search space to only those sequences of definition applications where each new definition is applied exhaustively, before another new definition is considered. So the sequence of definition applications is then determined by an initial linear ordering of the set of definitions. This restriction can obviously result in worse proof compression than is possible in the general case that allows mixed application of the definitions.

The second simplification applies greediness once more, restricting the initial linear ordering of the set of definition to be done according to the compression power of the definitions. This means that first the most compressing definition D_1 is exhaustively applied to the proof, yielding a new proof $D^*_{1_{min}}(P)$ together with the added clause D_1. Let us denote this new proof P_1. Then again, the most compressing definition D_2 is found for P_1 (containing also D_1), and added and applied exhaustively, yielding proof P_2. This greedy process generates (provided all weights are positive and definitions monotone wrt w) a finite sequence of definitions and proofs. The final proof P_n can then no longer be compressed by introducing any new definition. This greedy algorithm, based on efficiently approximated algorithm for finding the most compressing definition is the basis for our implementation and experiments.

Is compressed proof really a proof? One could argue that, after performing compression on a proof, the result is not a proof anymore. Consider, for example, the following fragment of a resolution proof:

$$\ldots,\ a \mid b,\ \neg a,\ b,\ \ldots$$

Using the definition $d = a \mid b$, we obtain

$$\ldots,\ d,\ \neg a,\ b,\ \ldots$$

Strictly speaking, this is not a resolution proof anymore, the inference is broken. The way we understand a compressed sequence as a proof is, using "macro-inferences", which means inference rules that, first, expand all occurences of definitions, then perform the original inference, and finally fold the result using the

definitions.This is a common phenomenon when dealing with formal proofs and their presentation in e.g. formal proof assistants: Some knowledge (typically the rewriting and the definitional knowledge) is applied implicitly, without explicit reference to it and its explicit application.

2.2 Motivating Example

Let's work out an example of the most compressing definition in a very simple setting: let the input consist of a single term

$$f(f(\dots(f(a)))),$$

or shortly $f^n(a)$, for a single unary symbol f, constant a and some n. The weight of the term is $n+1$. Any compressing definition D has to be

$$d(X) = f^m(X)$$

for some m, and the shortest compression $D^*_{min}(f^n(a))$ is, up to the order of function symbols,

$$d^{n \operatorname{div} m} f^{n \bmod m}(a).$$

The weight of the definition is $m+4$, the weight of the resulting term is $n \operatorname{div} m + n \bmod m + 1$. Hence, finding the most compressing definition is equivalent to finding m minimizing the expression

$$m + n \operatorname{div} m + n \bmod m.$$

This problem has obviously polynomial complexity with respect to the input size, but it suggests that arithmetic can be involved.

3 Implementation

3.1 The Most Compressing Definition and the Least General Generalization

First it is necessary to describe all the possible candidates for a new definition, and count their number (note that we are searching only for the bodies of the definitional clauses, because the heads are formed by a new symbol and a list of free variables occurring in the body). Searching for the most compressing definition for a set of terms M (representing a given proof) corresponds to searching for some least general generalization (lgg - see [Plo69] for exact definition) over a subset of all subterms of M. Not all compressing definitions are lgg's, and not even the most compressing one has to be an lgg, however, the latter case is very rare. Our heuristics for compressing proofs will be based on searching for the most compressing lgg.

Let us look at an example which shows limits of our approach. Let M consist of a single term

$$f(g(X,\ldots,X),\ldots,g(X,\ldots,X)),$$

where f is an n-ary symbol, g is an m-ary symbol, and the total weight is $(m+1)\cdot n+1$. The lgg set consists of the following terms:

$$\{f(g(X,\ldots,X),\ldots,g(X,\ldots,X)),g(X,\ldots,X),X\}$$

Now, there are two reasonable candidates for the most compressing definition:

- $d(X)=f(X,\ldots,X)$. Then $D^*_{min}(M)=d(g(X,\ldots,X))$ and the total weight is $m+n+6$.
- $d(X)=g(X,\ldots,X)$. Then $D^*_{min}(M)=f(d(X),\ldots,d(X))$ and the total weight is $m+2n+5$.

So, if $n=1$, both definitions are the most compressing, while for $n>1$ the first definition wins. However, lgg always gives the second one.

3.2 Finding the Most Compressing Definition Using Substitution Trees

In this subsection, we describe our heuristics for the problem of finding the most compressing definition for a set of terms M. Our approach is based on a data structure called *substitution tree* (see [Gra96]), which has several useful properties:

1. Substitution trees are standard way to effectively save all subterms from M.
2. All nodes of the tree then always represent the use of lgg on a subset of all subterms of M. Moreover, there is always a tree containing a node that represents the body of the most compressing definition.
3. From the tree it is possible to quickly compute the upper estimate of the efficiency of the proof compression in the case of using a particular node as the body of the definition.

Now we will describe Algorithm 1. The input is a set of terms that correspond to some proof. The output is a term, an approximation of the most compressing definition. When such a definition does not exist, the algorithm returns "fail". At line 7 the variable U is used to denote all subterms from the input. Then a substitution tree T is created from U. T additionally remembers in its leaves the frequency of the occurrences of terms from U. At line 9, procedure propagate_freq_into_tree is called with T, described in Algorithm 2. This procedure recursively adds to each node of T the frequency corresponding to the sum of its children. From this information it is possible to compute the upper estimate for the number of application of the definitions that correspond to the node of T.

The function at line 10 described at Algorithm 3 counts recursively the gain from the variable in all nodes of T that appear in the substitutions at the left-hand side. In the leaves the gain is computed from each variable at the left-hand

Algorithm 1. The most compressing definition

```
1. function most_compressing_definition (proof : set of terms) : term;
   % this function returns the most compressing definition as term of the form
   % def(x_1, ..., x_n) :- T(x_1, ..., x_n) of input proof where x_1, ...x_n are variables,
   % T is some term with at least one occurrence of every variable x_1, ...x_n and
   % def is a new function symbol. If there is no compressing definition of proof
   % then the function returns fail.
2. var
3.     U : multiset of terms;
4.     T : substitution tree;
5.     L : list of tuples of the form:
               ⟨gain : integer, tag: (upper_bound, exact), definition : term⟩;
6. {
7.     U := union of all subterms of every element of proof;
8.     T := construct a substitution tree from U with frequencies of all terms from
   U in leaves;
9.     propagate_freqs_into_tree(T);
10.    (root T).substitution_gain :=
                   propagate_gains_of_substitutions_into_tree(T);
11.    L := empty;
12.    for each node N of tree T do {
13.        L := L + ⟨G, upper_bound, D⟩ where        % concatenates tuple to list
14.            D := create_definition_form_node(N),
15.            G := (N.freq−1)∗k(D) + (j(D, N) - p(D) where
16.                k(d :- b) := w(b) − w(d),                    % definition gain
17.                j(d :- b, v) := h′(b) − h′(d) where     % definition gain of subst.
18.                        h′(x) { n    if ⟨x, n⟩ ∈ v.substitution_gain
                                  { w(x) otherwise                    ,
19.                p(d :- b) := w(d :- b)+1,        % penalization of def. declaration
20.    }
21.    sort L with decreasing order by gain, tag where tag exact > tag upper_bound;
22.    while (L[1].tag = upper_bound) and (L[1].gain>0) do {
23.        L[1].gain := calculate the exact L[1].definition gain as: w(proof) -
   w(greedy application of L[1].definition on proof) - p(L[1].definition) where
24.            p(d :- b) := w(d :- b)+1;          % penalization of def. declaration
25.        L[1].tag := exact;              % changes tag of the first element of list L
26.        L := merge L[1] with L[2..];
   % merges the first element of list with its tail by the same rules as at 21.
27.    }
28.    if   L[1].gain > 0   then   return L[1].definition   else   return fail;
29. }
```

Algorithm 2. Propagate frequencies into tree

```
1. procedure propagate_freqs_into_tree (T : substitution_tree) ;
2. {
3.     if T is leaf then exit;              % Frequency of leaf is already calculated.
4.     (root T).substitution_gain := empty;
5.     for each son S of root T do {
6.         propagate_freqs_into_tree(subtree of T where S is root); % Calculates
   freqs in subtree S.
7.         (root T).freq := (root T).freq + S.freq;
8.     }
9. }
```

Algorithm 3. Propagate gains of substitutions into tree

```
1. function propagate_gains_of_substitutions_into_tree
            (T : substitution_tree)
            : set of couples of the form: ⟨var : subst. variable, gain : integer⟩ ;
2. var R : set of couples of the form: ⟨var : subst. variable, gain : integer⟩ ;
3. {
4.      (root T).substitution_gain := empty;   % there are no subst. variables in leaves.
5.      for each son S of root T do {
6.          (root T).substitution_gain := merge (root T).substitution_gain with
            propagate_gains_of_substitutions_into_tree(subtree of T where S is root)
            so that, couples with the same variable is merged into one couple and
            its gain is a sum of gains of all original couples with the same variable;
7.      }
8.      R := empty;
9.      for each substitution θ=T of substitution set in root T do {
10.         R := R∪⟨θ,h(T)⟩ where
11.             h(x) { n   if ⟨x, n⟩ ∈ (root T).substitution_gain
                     { w(x) otherwise ;
12.     }
13.     if T is leaf then
14.         for each couple ⟨x,n⟩ of R do {
15.             R := (R \ ⟨x,n⟩)∪⟨x, n*(root T).freq⟩;
16.         }
17.     return R;
18. }
```

Algorithm 4. The greatest proof compressing

```
1. function greatest_proof_compressing (proof : set of terms) : set of terms;
2. var
3.      R : set of terms;
4.      T : definition;
5. {
6.      R := proof;
7.      T := most_compressing_definition(R);
8.      while T ≠ fail do {
9.          R := T∪(application of definition T on R);
10.         T := most_compressing_definition(R);
11.     }
12.     return R;
13. }
```

side of the substitution as the frequency of the leaf times the weight of the term on the right hand side of each substitution where the weight of the term is computed using the function w. In the nodes that are not leaves the gain from each variable on the left-hand side is computed as the weight of the term on the right-hand side, where the weight of the term is determined using the function h (see line 11 in Algorithm 3). The gains for function h are computed by merging the gains of the node's children by summing the values at the same variables (lines 5, 6 in Algorithm 3). The gains obtained in this way will be used for fast computing of the estimate of the efficiency of the searched-for definition (lines 11 to 20 in Algorithm 1).

Now we will describe the upper estimate of the efficiency of the definition given by node N in the tree T. First we create the definition D corresponding to the node. This is done by composing all substitutions from the root to N. The resulting substitution will define the body of the definition, and all its substitutional variables, and variables that appeared in the terms inserted into the tree (these variables have to be distinguished, see [Gra96]) will be defined as the arguments in the head of the definition.

The upper estimate of the definition's efficiency - the definition's gain (i.e. the upper estimate of w(proof before application of the definition) - w(proof after application of the definition) is described at lines 15 to 19.

Upper estimates are computed, because an exact computing of the definition's gain is quite inefficient (we have to go through the whole proof, and apply the definition). On the other hand, the computation of the upper estimate of all nodes from T in the way described above has the same complexity as just building the substitution tree T.

The resulting estimate is inserted into the list L as a tuple $\langle$upper estimate, tag: upper_bound, definition $D\rangle$ (see line 13). After inserting all the upper estimates of all the nodes of T, the list L is sorted decreasingly by the size of the upper estimate. If the upper estimates are equal, the tuple with tag "exact" is preferred to the tuple with tag "upper_bound".

Now we always test if the first member of the list L is already an exact efficiency value denoted with the tag "exact". If so, the value D in this member is the searched-for most compressing definition. This definition is the best among all the nodes of the tree T, but it does not have to be the best definition absolutely, because we are selecting only from the nodes of the tree T. If the value is not exact, we compute the exact value of D for the proof (by replacing). The result is saved as the first element of the list, and is tagged "exact". Then we sort the list, and repeat, see lines 22 to 27.

If the resulting gain is more than 0, it means that the applied definition shortens the proof and this definition is returned at the output. If not, we return "fail", because no compressing definition appears among the nodes of T.

Algorithm 4 describes the greedy approach (see Subsection 2.1) for finding an approximation of the greatest proof compression.

4 Testing

The initial implementation described above has been tested on the whole TPTP library [Sut09], and on two families of proofs coming from recent research in algebra. In both cases the simplest symbol-counting weight function was used for measuring the proof improvement.

4.1 Testing on the TPTP Library

The TPTP library contains a large number of ATP problems from various areas of ATP use, which makes it suitable for large-scale evaluation of the proof compression algorithm. For the testing described here, TPTP version 4.0.1 was used, containing 16512 problems. All available TPTP proofs found by the EP version 1.1 theorem prover [Sch02] were obtained from the TPTP developer Geoff Sutcliffe. This is a testing set of 7807 ATP proofs in the TPTP syntax, which is a subset of the Prolog syntax. These proofs were postprocessed by a simple Perl script into a list of formulae (i.e., forgetting about the inference structure). This again can be considered to be just a list of Prolog terms, and hence it is already an input to the proof compression algorithm explained above.

The testing was done on an eight-core 64bit Intel Xeon E5520 2.27 GHz Linux machine with 8GB of RAM. SWI Prolog was used to run the proof compression algorithm. SWI Prolog has some internal memory limits that do not allow it to get past 2GB boundary, so for very large proofs the implementation can now fail for lack of memory. Because the implementation can also take quite a long time for large proofs (the longest example we are aware of was about one hour), we have given each of the TPTP proofs a time limit of 60 seconds to be able to finish the large-scale testing in reasonable time. 4890 of the 7807 proofs (63%) were completely compressed within the time limit, i.e., the algorithm has successfully finished in 60 seconds. For the remaining 2917 proofs the algorithm typically has found the initial most compressing definitions, but has not converged to the point where no more compressing definitions exist. The final compression ratios for the 4890 successful runs can be viewed online at our webpage[4], and all the TPTP proofs together with the algorithm inputs and outputs can also be viewed there[5]. The interesting data extracted from the testing are summarized in Table 1, and Figure 1 shows the graph of the compression performance on the 4890 finished proofs.

Table 1. Results of testing the proof compression algorithm on the TPTP library

TPTP problems	proved by EP	compressed in 60s	timeout in 60s
16512	7807	4890	2917
greatest comp. ratio	least comp. ratio	median ratio	comp. below 50%
0.1844	0.9969	0.8100	135

[4] http://mws.cs.ru.nl/~urban/compression/00tstp_final_ratios
[5] http://mws.cs.ru.nl/~urban/compression/Solutions_tstp/

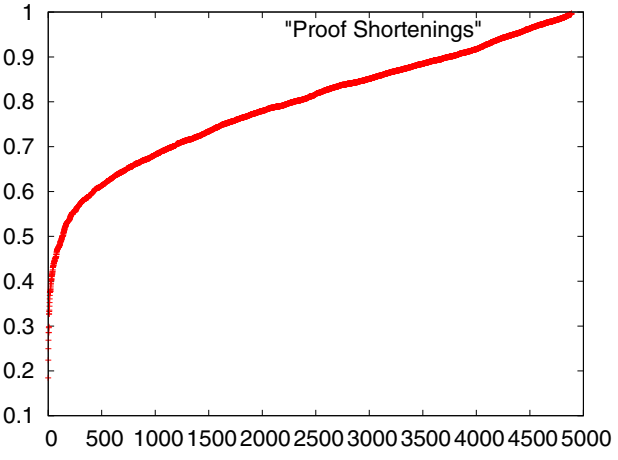


Fig. 1. Proof compression ratios on the TPTP library, sorted from the best to the worst

4.2 Testing on Algebraic Problems

One of the aspects of the present work is, to address the problem of human understanding of machine generated proofs. For this reason, we tested our implementation on two families of proofs, coming from different areas of current research in algebra.

Loops with abelian inner mappings. We investigated a proof, obtained by Waldmeister, that every uniquely 2-divisible loop with abelian inner mapping group of exponent 2, is commutative and associative [PS08][6]. In both cases, the very first definition the implementation found, was the right inverse operation (that is, the term $1/x$), and the left inverse followed soon. Other interesting definitions were shortcuts for various compositions of the inner mappings. Both proofs had final ratio about 0.75.

Symmetric-by-medial groupoids. We investigated three related proofs, obtained in [Sta08] with Prover9[7]. The importance of the term $xy \cdot zu$ in the theory of distributive groupoids was recognized immediately in each case. In the latter two cases, it cuts the proof weight by more than 10%. Sadly, other definitions found by the implementation seem to have little mathematical meaning. The final ratios were 0.65, 0.72, and 0.75, respectively.

Algebraic problems in TPTP. Many algebraic problems were recently submitted to TPTP [PS08], for instance, problems in the interval GRP654 to GRP763. Our notes in Section 5.2 are also based on the inspection of the results on these problems.

[6] http://mws.cs.ru.nl/~urban/compression/aim_2div/
[7] http://mws.cs.ru.nl/~urban/compression/symbymed/

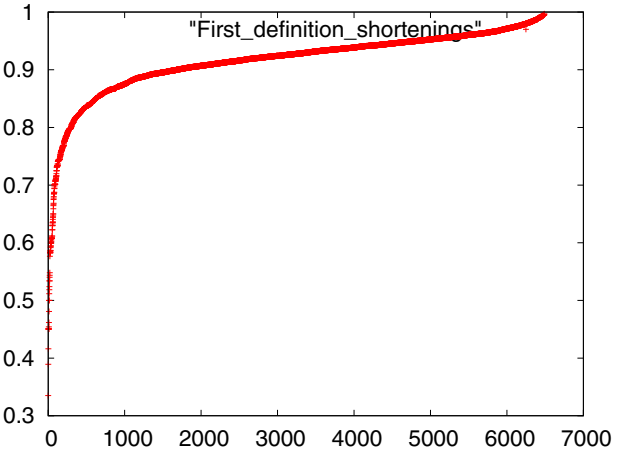


Fig. 2. Proof shortening ratios by the most compressing definition on the TPTP library, sorted from the best to the worst

5 Examples and Discussion

5.1 A Good and a Bad Example

To get a taste of the results, we shall look closer at one of the most successful, and one of the most unsuccessful compressions produced by our implementation on the TPTP problems.

The champion is SWV158, with final compression ratio 0.1844 (from 2277 to 420)[8]. The first definition is of enormous weight, 86, and its application saves almost half of the proof weight. This is an equality, with a variable on one side, and a very nested term on the other side, with many constant leaves and just one free variable with 6 occurrences. Three more heavy definitions of a similar kind, and 10 lighter ones, finish the compression.

To the other extreme, let's mention GRP754, with final compression ratio 0.9710 (from 726 to 705)[9]. There is a single compressing definition, setting `def1(A, B, C)= (A=mult(B, C))`, which is applied on roughly two thirds of the proof lines.

Generally speaking, heavy definitions are rare. Most definitions save just very few symbols, but are applied many times in the proof.

5.2 Understanding Machine Generated Proofs

Our experiments show that introducing a new definition that formally reduces weight of the proof, rarely gives a notion interesting from mathematician's point

[8] http://mws.cs.ru.nl/~urban/compression/Solutions_tstp/SWV/SWV158+1/
[9] http://mws.cs.ru.nl/~urban/compression/Solutions_tstp/GRP/GRP754-1/

of view. Interesting exceptions exist: the implementation discovered notions like left and right inverse, and in some cases isolated important concepts that occur frequently in the proof.

Yet, reading the result of the overall greedy algorithm, it is a mess. The problem seems to have several layers. First of all, only few definitions have a good mathematical meaning. It is desirable, to introduce other measures, to judge which definitions are "good" and which are "bad", perhaps in the spirit of AGInT [PGS06]. The most compressing criterion is a reasonable heuristic, but far from perfect.

Another aspect is that, for human readers, learning new definitions is costly. In fact, looking at the examples, we realized that many definitions save just one character, even the top ones (their choice by the algorithm comes from the fact that they can be used many times). Perhaps we shall add a penalty to each new definition, based on how difficult is it to grasp it, relatively to how useful it is. Too short definitions, or those that are used only few times, shall be discarded.

One particular example of "bad" definition is the following. For the sake of simplicity, assume the signature consists of a single binary function symbol f. Then, (almost) any proof can be simplified introducing the predicate $P(x, y, z)$, defined by $f(x, y) = z$, saving one symbol per (almost every) line. Further in this direction, the formulae in the proof are actually very likely to be in the form $f(\_, \_) = f(\_, \_)$, and the corresponding 4-ary predicate symbol shortens the proof by $5/7$. Indeed, such definitions don't bring any better understanding. The weight function, or the "beauty criterion", shall avoid this sort of definitions.

6 Future Work and Conclusions

The presented system provides a useful means for experimenting with introducing new definitions based on the frequency and weight of subterms in a proof. The general problem of greatest proof compression seems to be quite hard, however our heuristic greedy implementation seems to perform reasonably well already in its first version. It seems to be an interesting open problem, to determine the complexity class of finding the greatest proof compression.

The initial evaluation using the most straightforward weight assignment has allowed us to immediately see some deficiencies in overly greedy introduction of new definitions. The system is sometimes capable of identifying mathematically interesting concepts that significantly compress the proofs, however many times the introduced definitions seem to be of little mathematical value, and only complicate the proof understanding. As already mentioned above, this will likely lead to further research about the proper offset between the benefits of the proof compression, and the benefits of not having to deal with too many similar concepts in one's head. It is obvious that shorter proofs don't always have to be "nicer" (whatever it means), but it is obviously also good to have tools that can produce the best result according to a precisely defined criterion.

The advantage of our system is that a lot of experimenting can be done using the weight assignments. For example, we could try:

- to weight equality symbol with zero (this is sufficient to avoid definitions of the form $f(x, y) = z$),
- to add learning penalties, for instance, by setting the weight of a new symbol by the maximal weight of symbols in its defining term, plus one,
- try to learn weight assignment patterns by data-mining techniques on a large body of available structured mathematics, e.g., the formal Mizar library.

The last option even suggests some more interesting experiments in the context of a large formal body of human-written mathematics. For example, it is feasible (using the MPTP system) to expand the whole Mizar library (or a suitable part of it) into a basic set-theoretical notation, i.e., using just the membership and equality predicates, and eliminating all definitions introduced by humans. This will likely result in a very large, but manageable (e.g. with complete term sharing in the implementation) blow-up of the library. Then the system can be used to search for interesting definitions automatically, and the results can be compared with the definitions introduced by human authors.

Another potential use that we are very interested in, is the use of the subsystem as a "concept developing" component in larger meta-systems (like MaLARea [USPV08]) for theorem proving in structured and large theories. The experience with ATP in large theories so far shows that blind recursive inclusion of all available definitions and all the theorems known about them significantly decreases the ATP performance in the large theories. Introducing new concepts, and only selecting some of their important properties (like commutativity) is also a very common feature of abstract mathematical developments done by human mathematicians. Both the human evidence, and the evidence from doing ATP in large theories therefore points to the importance of including good concept creation into the overall process of mathematical theorem proving and theory development.

Acknowledgements

We would like to thank the anonymous LPAR referees for thorough reading, and a number of useful comments and suggestions to the first version of the paper. Thanks to Geoff Sutcliffe for providing the E proofs of the TPTP problems.

References

[Dahn98] Dahn, I.: Robbins algebras are boolean: A revision of mccune's computer-generated solution of robbins problem. Journal of Algebra 208(2), 526–532 (1998)

[Gon07] Gonthier, G.: The four colour theorem: Engineering of a formal proof. In: Kapur, D. (ed.) ASCM 2007. LNCS (LNAI), vol. 5081, p. 333. Springer, Heidelberg (2008)

[Gra96] Graf, P.: Term Indexing. LNCS, vol. 1053. Springer, Heidelberg (1996)
[Gra01] Grabowski, A.: Robbins algebras vs. boolean algebras. Formalized Mathematics 9(4), 681–690 (2001)
[Hua96] Huang, X.: Translating machine-generated resolution proofs into nd-proofs at the assertion level. In: Foo, N.Y., Göbel, R. (eds.) PRICAI 1996. LNCS, vol. 1114, pp. 399–410. Springer, Heidelberg (1996)
[McC97] McCune, W.: Solution of the Robbins problem. J. Autom. Reasoning 19(3), 263–276 (1997)
[MP08] Meng, J., Paulson, L.C.: Translating higher-order clauses to first-order clauses. J. Autom. Reasoning 40(1), 35–60 (2008)
[NW01] Nonnengart, A., Weidenbach, C.: Computing small clause normal forms. In: Robinson, J.A., Voronkov, A. (eds.) Handbook of Automated Reasoning, vol. I, pp. 335–367. Elsevier and MIT Press (2001)
[PGS06] Puzis, Y., Gao, Y., Sutcliffe, G.: Automated generation of interesting theorems. In: FLAIRS Conference, pp. 49–54 (2006)
[Plo69] Plotkin, G.D.: A note on inductive generalization. Machine Intelligence 5, 153–163 (1969)
[PP95] Proietti, M., Pettorossi, A.: Unfolding–definition–folding, in this order, for avoiding unnecessary variables in logic programs. Theoretical Computer Science 142(1), 89–124 (1995)
[PS08] Phillips, J.D., Stanovský, D.: Automated theorem proving in loop theory. In: Sutcliffe, G., Colton, S., Schulz, S. (eds.) ESARM: Empirically Successful Automated Reasoning in Mathematics. CEUR Workshop Proceedings, vol. 378, pp. 54–54. CEUR (2008)
[Sch02] Schulz, S.: E – a brainiac theorem prover. Journal of AI Communications 15(2-3), 111–126 (2002)
[Sta08] Stanovský, D.: Distributive groupoids are symmetric-by-medial: An elementary proof. Commentationes Mathematicae Universitatis Carolinae 49(4), 541–546 (2008)
[Sut09] Sutcliffe, G.: The tptp problem library and associated infrastructure. J. Autom. Reasoning 43(4), 337–362 (2009)
[TPS07] Trac, S., Puzis, Y., Sutcliffe, G.: An interactive derivation viewer. In: UITP 2006. Electronic Notes in Theoretical Computer Science, vol. 174, pp. 109–123. Elsevier, Amsterdam (2007)
[TS84] Tamaki, H., Sato, T.: Unfold/fold transformations of logic programs. In: Tärnlund, S.-Å. (ed.) Proceedings of The Second International Conference on Logic Programming, pp. 127–139 (1984)
[Urb08] Urban, J.: Automated reasoning for mizar: Artificial intelligence through knowledge exchange. In: Rudnicki, P., Sutcliffe, G., Konev, B., Schmidt, R.A., Schulz, S. (eds.) LPAR Workshops. CEUR Workshop Proceedings, vol. 418. CEUR-WS.org (2008)
[USPV08] Urban, J., Sutcliffe, G., Pudlák, P., Vyskočil, 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)
[Ver01] Veroff, R.: Solving open questions and other challenge problems using proof sketches. J. Autom. Reasoning 27(2), 157–174 (2001)
[VV07] Vyskočil, J., Štěpánek, P.: Improving efficiency of prolog programs by fully automated unfold/fold transformation. In: Gelbukh, A., Kuri Morales, Á.F. (eds.) MICAI 2007. LNCS (LNAI), vol. 4827, pp. 305–315. Springer, Heidelberg (2007)

Atomic Cut Introduction by Resolution: Proof Structuring and Compression

Bruno Woltzenlogel Paleo

Institut für Computersprachen, Vienna University of Technology, Austria
bruno@logic.at
INRIA, LORIA, Nancy, France
Bruno.WoltzenlogelPaleo@loria.fr

Abstract. The careful introduction of cut inferences can be used to structure and possibly compress formal sequent calculus proofs. This paper presents CIRes, an algorithm for the introduction of atomic cuts based on various modifications and improvements of the CERes method, which was originally devised for efficient cut-elimination. It is also demonstrated that CIRes is capable of compressing proofs, and the amount of compression is shown to be exponential in the length of proofs.

1 Introduction

It is well-known that eliminating cuts frequently increases the size and length of proofs. In the worst case, cut-elimination can produce non-elementarily larger and longer proofs [18,17]. Given this fact, it is natural to attempt to devise methods that could introduce cuts and compress sequent calculus proofs. However, this has been a notoriously difficult task. Indeed, the problem of answering, given a proof φ and a number l such that $l \leq \mathit{length}(\varphi)$, whether there is a proof ψ of the same theorem and such that $\mathit{length}(\psi) < l$ is known to be undecidable [8]. Nevertheless, a lower bound for compressibility based on specific cut-introduction methods that are inverse of reductive cut-elimination methods is known [12],[1], and some ad-hoc methods to introduce cuts of restricted forms have been proposed. They are based on techniques from automated theorem proving, such as conflict-driven formula learning [11], and from logic programming, such as tabling [16,15].

Besides compression, cut-introduction can also be used for structuring and extracting interesting concepts from proofs. In [10], for example, it is shown that many translation and pre-processing techniques of automated deduction can be seen as introduction of cuts, from a proof-theoretical point of view. Furthermore, in the formalization of mathematical proofs, lemmas correspond to cuts, and hence the automatic introduction of cuts is, in a formal level, the automatic discovery of lemmas that are potentially useful for structuring mathematical

[1] A cut-introduction method g is inverse of a reductive cut-elimination method if and only if, for any cut-free proof φ, the proof with cuts $g(\varphi)$ rewrites to φ according to the rewriting rules in Appendix B.

E.M. Clarke and A. Voronkov (Eds.): LPAR-16, LNAI 6355, pp. 463–480, 2010.

knowledge. Naturally, the use of cut-introduction techniques could in principle also be applied to the structuring of knowledge in other fields of Science, as argued in [21,22].

This paper presents a new method for the introduction of atomic cuts: CIRes. The method is described in a simplified and self-contained manner[2] in Section 3, and a proof that it is able to provide an exponential compression in the length of proofs is given in Section 4.

2 The Sequent Calculus

The inference rules of the sequent calculus used in this paper are shown below. A *sequent* is a pair of multisets of formulas. The sequent below the line of an inference rule is its *conclusion*, while the sequents above the line are its *premises*. The formulas highlighted in red color are called *main* formulas of the rules, while the formulas highlighted in blue color are called *auxiliary* formulas. Main and auxiliary formulas are called *active*. An inference is said to *operate* on its active formulas. Auxiliary formulas are *immediate ancetors* of main formulas. The non-active formulas in the premises of an inference are immediate ancestors of the correponding formulas in its conclusion. The *ancestor relation* is the reflexive transitive closure of the immediate ancestor relation. A *cut-ancestor* is an ancestor of an auxiliary formula of a cut inference. Note that CERes and the methods described in this paper are robust and work with other kinds of sequent calculi, as long as weakening and contraction are available in some (possibly implicit) form.

– **The Axiom Rule**:

$$\frac{}{A \vdash A}\ axiom$$

where A is any atomic formula.

– **Propositional Rules**:

$$\frac{F_1, F_2, \Gamma \vdash \Delta}{F_1 \wedge F_2, \Gamma \vdash \Delta}\ \wedge_l \qquad \frac{\Gamma_1 \vdash \Delta_1, F_1 \quad \Gamma_2 \vdash \Delta_2, F_2}{\Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2, F_1 \wedge F_2}\ \wedge_r \qquad \frac{F_1, \Gamma_1 \vdash \Delta_1 \quad F_2, \Gamma_2 \vdash \Delta_2}{F_1 \vee F_2, \Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2}\ \vee_l$$

$$\frac{\Gamma_1 \vdash \Delta_1, F_1 \quad F_2, \Gamma_2 \vdash \Delta_2}{F_1 \to F_2, \Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2}\ \to_l \qquad \frac{F_1, \Gamma \vdash \Delta, F_2}{\Gamma \vdash \Delta, F_1 \to F_2}\ \to_r \qquad \frac{\Gamma \vdash \Delta, F_1, F_2}{\Gamma \vdash \Delta, F_1 \vee F_2}\ \vee_r$$

$$\frac{\Gamma \vdash \Delta, F}{\neg F, \Gamma \vdash \Delta}\ \neg_l \qquad \frac{F, \Gamma \vdash \Delta}{\Gamma \vdash \Delta, \neg F}\ \neg_r$$

– **Structural Rules (Weakening and Contraction)**:

$$\frac{\Gamma \vdash \Delta}{F, \Gamma \vdash \Delta}\ w_l \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash \Delta, F}\ w_r \qquad \frac{F, F, \Gamma \vdash \Delta}{F, \Gamma \vdash \Delta}\ c_l \qquad \frac{\Gamma \vdash \Delta, F, F}{\Gamma \vdash \Delta, F}\ c_r$$

– **The Cut Rule**:

$$\frac{\Gamma_1 \vdash \Delta_1, F \quad F, \Gamma_2 \vdash \Delta_2}{\Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2}\ cut$$

[2] A more general and technically more detailed definition of the CIRes method is available in [21].

– **Quantifier Rules**:

$$\frac{F\{x \leftarrow t\}, \Gamma \vdash \Delta}{(\forall x)F, \Gamma \vdash \Delta}\ \forall_l \qquad \frac{\Gamma \vdash \Delta, F\{x \leftarrow \alpha\}}{\Gamma \vdash \Delta, (\forall x)F}\ \forall_r$$

$$\frac{F\{x \leftarrow \alpha\}, \Gamma \vdash \Delta}{(\exists x)F, \Gamma \vdash \Delta}\ \exists_l \qquad \frac{\Gamma \vdash \Delta, F\{x \leftarrow t\}}{\Gamma \vdash \Delta, (\exists x)F}\ \exists_r$$

For the $\forall_r$ and the $\exists_l$ rules, the variable α must not occur in Γ nor in Δ nor in F. This is the *eigenvariable condition*. For the $\forall_l$ and the $\exists_r$ rules the term t must not contain a variable that is bound in F.

3 The CIRes Method

Curiously, even though CIRes aims at introducing cuts, it makes use of the CERes method [4], which was originally developed for efficient cut-elimination [5]. The CERes method extracts an unsatisfiable clause set from the input proof with cuts and refutes it by resolution. The resolution refutation then serves as a skeleton for a proof containing only atomic cuts, obtained by combining the refutation with projections (i.e. cut-free parts) of the input proof with respect to the clauses of the refuted clause set. The essential idea behind CIRes is based on two simple observations about reductive cut-elimination and CERes:

– In a naive attempt to introduce cuts by applying the proof rewriting rules of reductive cut-elimination methods (i.e. gentzen-style cut-elimination methods) shown in Appendix B in an inverse direction, the first step, which is the introduction of atomic cuts on the top of cut-free proofs, is trivial. However, pushing the cuts further down (by applying inverse rank reduction rules), combining the cuts to make more complex cuts (i.e. increasing the grade), and exploiting redundancies in the form of contractions is highly non-trivial.
– If applied to a proof ψ containing only atomic cuts in the top, CERes outputs a proof ψ' containing atomic cuts in the bottom. This is so because ψ' is constructed by adding the projections to the top of the resolution refutation. The refutation occurs in the bottom of ψ', and hence the atomic cuts of ψ', which are nothing else but the resolution inferences of the refutation, also occur in the bottom of ψ'.

The CIRes method exploits these observations in a conceptually simple way: it trivially adds atomic cuts to every leaf of the cut-free proof; and then it applies CERes to push these cuts down.

Compression can be achieved mainly due to two ways by which CERes is able to reduce or avoid redundancies:

– It is possible that the refutation uses only some clauses of the clause set of ψ. The effect is that large parts of ψ (i.e. the projections with respect to the unused clauses) can be deleted and replaced by weakening, thus resulting in a smaller proof.

– If the refutation contains factoring inferences, ψ' will contain contractions operating on ancestors of cut-formulas (note that, in the construction of ψ', resolution inferences become atomic cuts, and factoring inferences become contractions). Since the presence of contractions operating on ancestors of cut-formulas is a major reason for the increase of size and length during cut-elimination, adding such contractions (via factoring) can lead to compression.

However, the original (standard) CERes method [4] also introduces other kinds of redundancies, in the form of unnecessary duplications that occur during the construction of (standard) clause sets and (S-)projections. Therefore, it would be hopeless to expect proof compression by CIRes if it used the standard CERes method. Fortunately, these redundancies can be avoided by using the improved concepts of *swapped clause sets* and *O-projections* that are presented in this paper.

The following subsections define and illustrate all steps of the CIRes method.

3.1 Step 1: Introduction of Atomic Cuts on Top

The first step is the introduction of atomic cuts on the top of the cut-free proof, and it can easily be done according to Definition 1.

Definition 1 (Introduction of Atomic Cuts). *Let φ be a cut-free proof. Then φ^a denotes the proof obtained from φ by replacing every axiom inference with conclusion sequent of the form $A \vdash A$ by a subproof of the form:*

$$\dfrac{A \vdash A \qquad A \vdash A}{A \vdash A}\,cut$$

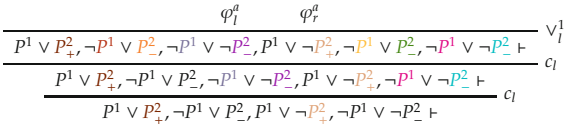
Example 1. Let φ be the cut-free proof[3] below, whose end-sequent was adapted from an instance of a sequence of clause sets used in [9] to show that the resolution calculus can produce significantly shorter proofs than analytic tableaux calculi. Predicate symbols having different subscript or superscript indexes[4] denote distinct atoms.

$$
\begin{prooftree}
\AxiomC{$P^1 \vdash P^1$}
\RightLabel{$\neg_l$}
\UnaryInfC{$P^1, \neg P^1 \vdash$}
\AxiomC{$P^1 \vdash P^1$}
\RightLabel{$\neg_l$}
\UnaryInfC{$P^1, \neg P^1 \vdash$}
\AxiomC{$P^2_- \vdash P^2_-$}
\RightLabel{$\neg_l$}
\UnaryInfC{$P^2_-, \neg P^2_- \vdash$}
\RightLabel{$\vee_l$}
\BinaryInfC{$P^2_-, P^1, \neg P^1 \vee \neg P^2_- \vdash$}
\RightLabel{$\vee_l$}
\BinaryInfC{$P^1, P^1, \neg P^1 \vee P^2_-, \neg P^1 \vee \neg P^2_- \vdash$}
\RightLabel{c_l}
\UnaryInfC{$P^1, \neg P^1 \vee P^2_-, \neg P^1 \vee \neg P^2_- \vdash$}
\AxiomC{$P^1 \vdash P^1$}
\RightLabel{$\neg_l$}
\UnaryInfC{$P^1, \neg P^1 \vdash$}
\AxiomC{$P^1 \vdash P^1$}
\RightLabel{$\neg_l$}
\UnaryInfC{$P^1, \neg P^1 \vdash$}
\AxiomC{$P^2_- \vdash P^2_-$}
\RightLabel{$\neg_l$}
\UnaryInfC{$P^2_-, \neg P^2_- \vdash$}
\RightLabel{$\vee_l$}
\BinaryInfC{$P^2_-, P^1, \neg P^1 \vee \neg P^2_- \vdash$}
\RightLabel{$\vee_l$}
\BinaryInfC{$P^1, P^1, \neg P^1 \vee P^2_-, \neg P^1 \vee \neg P^2_- \vdash$}
\RightLabel{c_l}
\UnaryInfC{$P^1, \neg P^1 \vee P^2_-, \neg P^1 \vee \neg P^2_- \vdash$}
\AxiomC{$P^2_+ \vdash P^2_+$}
\RightLabel{$\neg_l$}
\UnaryInfC{$P^2_+, \neg P^2_+ \vdash$}
\RightLabel{$\vee_l$}
\BinaryInfC{$P^2_+, P^1 \vee \neg P^2_+, \neg P^1 \vee P^2_-, \neg P^1 \vee \neg P^2_- \vdash$}
\RightLabel{$\vee_l$}
\BinaryInfC{$P^1 \vee P^2_+, \neg P^1 \vee P^2_-, \neg P^1 \vee \neg P^2_-, P^1 \vee \neg P^2_+, \neg P^1 \vee P^2_-, \neg P^1 \vee \neg P^2_- \vdash$}
\RightLabel{c_l}
\UnaryInfC{$P^1 \vee P^2_+, \neg P^1 \vee P^2_-, \neg P^1 \vee \neg P^2_-, P^1 \vee \neg P^2_+, \neg P^1 \vee \neg P^2_- \vdash$}
\RightLabel{c_l}
\UnaryInfC{$P^1 \vee P^2_+, \neg P^1 \vee P^2_-, P^1 \vee \neg P^2_+, \neg P^1 \vee \neg P^2_- \vdash$}
\end{prooftree}
$$

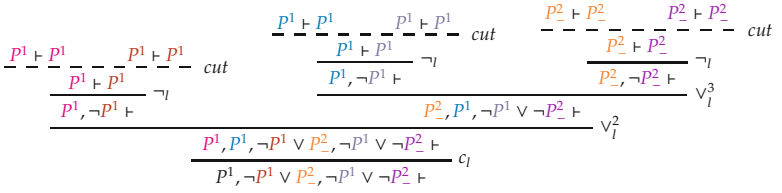
[3] Note that φ is purely propositional: quantifier rules are not used. Even though the examples shown in this paper are propositional and a sequence of propositional proofs is sufficient to show the possibility of exponential compression, CERes and CIRes work in classical first-order logic too. Moreover, extensions of CERes to higher-order logics and non-classical logics exist [20,6,1], and it can be expected that CIRes works for these logics too.

[4] Indexes are used because they are convenient for defining the rule that generates the whole sequence of clause sets, as explained in Section 4.

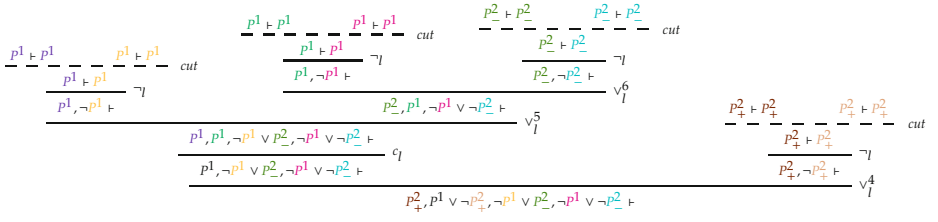
Following the first step of the CIRes method, φ^a is obtained by adding atomic cuts to the leaves of φ:



where φ^a_l is:



and φ^a_r is:



In φ^a above, each axiom sequent was highlighted with a different color. Other atomic formulas were highlighted with the same color of the axiom from which they descend. Atomic formulas that descend from more than one axiom, in case of contractions, were kept in black color. Each of the six $\vee_l$ inferences was marked with a distinct label from 1 to 6 so that they can be conveniently referred.

3.2 Step 2: Extraction of the Struct

The second step is the extraction of the *struct* $\mathcal{S}_{\varphi^a}$ of φ^a. The struct of φ^a contains information about all the relevant atomic sub-formulas (and their polarities) of all cut-formulas of φ^a as well as information about the branching structure of φ^a. A branching inference in φ^a corresponds to either a $\oplus$ or a $\otimes$ connective in $\mathcal{S}_{\varphi^a}$, depending on whether it operates on ancestors of cut-formulas or not. The struct is a compact representation of all information pertinent to cuts in a proof.

Definition 2 (Struct). *The* struct $\mathcal{S}_\psi$ *of a proof* ψ *is defined inductively, as follows:*

- *If* ψ *consists of an axiom inference* ρ *with axiom sequent* $A \vdash A$ *only, then:*
  - *If only the formula in the succedent is a cut-ancestor, then* $\mathcal{S}_\psi \doteq A$.
  - *If only the formula in the antecedent is a cut-ancestor, then* $\mathcal{S}_\psi \doteq \neg A$.

  - *If both formulas of the axiom sequent are cut-ancestors, then* $\mathcal{S}_\psi \doteq \neg A \otimes A$.
  - *If none of the formulas are cut-ancestors, then* $\mathcal{S}_\psi \doteq \epsilon_\otimes$.
- *If* ψ *ends with a unary inference* ρ*, then* $\mathcal{S}_\psi \doteq \mathcal{S}_{\psi'}$*, where* ψ' *is the immediate subproof of* ψ *(i.e. the subproof whose end-sequent is the premise sequent of* ρ*).*
- *If* ψ *ends with a binary inference* ρ *that operates on cut-ancestors: Let* ψ_1 *and* ψ_2 *be the immediate subproofs of* ψ*. Then:*

$$\mathcal{S}_\psi \doteq \mathcal{S}_{\psi_1} \oplus_\rho \mathcal{S}_{\psi_2}$$

- *If* ψ *ends with a binary inference* ρ *that does not operate on cut-ancestors: Let* ψ_1 *and* ψ_2 *be the immediate subproofs of* ψ*. Then:*

$$\mathcal{S}_\psi \doteq \mathcal{S}_{\psi_1} \otimes_\rho \mathcal{S}_{\psi_2}$$

The subscript of a connective may be omitted, if it is clear or unimportant to which inference it corresponds.

Example 2. The struct $\mathcal{S}_{\varphi^a}$ of φ^a is:

$$\mathcal{S}_{\varphi^a} \equiv ((P^1 \oplus \neg P^1) \overset{**}{\otimes}_2 ((P^1 \oplus \neg P^1) \overset{**}{\otimes}_3 (P^2_- \oplus \neg P^2_-))) \overset{***}{\otimes}_1 (((P^1 \oplus \neg P^1) \overset{**}{\otimes}_5 ((P^1 \oplus \neg P^1) \overset{**}{\otimes}_6 (P^2_- \oplus \neg P^2_-))) \overset{***}{\otimes}_4 (P^2_+ \oplus \neg P^2_+))$$

It is easy to verify that the connective $\otimes_i$ indeed corresponds to the $\vee_l$ inference marked with label i. The $\oplus$ connectives correspond to the atomic cuts on the top of φ^a and their subscripts have been omitted. Each $\otimes_i$ connective has been additionally marked with $*$ labels, whose colors are all the colors of ancestors of formulas on which $\vee_l^i$ operates. Although not strictly necessary, these labels are convenient, as shown in Example 3.

3.3 Step 3: Construction of the Swapped Clause Set

The connectives $\otimes$ and $\oplus$ can be interpreted as disjunction and conjunction, respectively. In this case, it is possible to prove that the struct is always unsatisfiable [21,4]. Informally and intuitively, this fact is also not so hard to see, since the struct contains (the atomic components of) all cut-formulas, which always occur in pairs of opposite polarity. Consequently, the struct contains pairs of dual substructs that cannot be simultaneously satisfied.

In order to refute the unsatisfiable struct by resolution, first it has to be transformed into clause normal form. This could be done by a standard conjunctive normal form transformation, as shown in Definition 3. This is essentially what is done in the standard CERes method.

Definition 3 ($\underset{s}{\rightsquigarrow}$)**.** *The standard struct normalization is defined by the following struct rewriting rules:*

$$S \otimes (S_1 \oplus \ldots \oplus S_n) \underset{s}{\rightsquigarrow} (S \otimes S_1) \oplus \ldots \oplus (S \otimes S_n)$$

$$(S_1 \oplus \ldots \oplus S_n) \otimes S \underset{s}{\rightsquigarrow} (S_1 \otimes S) \oplus \ldots \oplus (S_n \otimes S)$$

However, $\underset{s}{\rightsquigarrow}$ causes several duplications as $\otimes$ is distributed over $\oplus$, which can make the normalized struct exponentially bigger [2]. These duplications must be avoided, if proof compression is intended. One possible solution to reduce the duplications would be a pre-processing of φ^a that swaps[5] independent inferences that correspond to $\otimes$ upward. The resulting pre-processed proof would have a struct where $\otimes$ connectives already appear more deeply nested and do not need to be distributed over so many $\oplus$ connectives. This pre-processing of proofs would be computationally expensive, though. Fortunately, there is a better alternative, which involves normalizing the struct while implicitly taking into account the possibility of swapping inferences in the corresponding proof, as shown in the struct rewriting system of Definition 4.

Definition 4 ($\underset{w}{\rightsquigarrow}$)**.** *Let $\Omega_\rho(\varphi)$ be the set of atomic formula occurrences of φ that are descendants of axioms that contain ancestors of active formulas of ρ. The rewriting rules below distribute $\otimes$ only to those $\oplus$-juncts that contain an occurrence in $\Omega_\rho(\varphi)$. More precisely, $S_{n+1}, \ldots, S_{n+m}$ and S must contain at least one occurrence from $\Omega_\rho(\varphi)$ each (i.e. there is an atomic substruct S'_{n+k} of S_{n+k} such that $S'_{n+k} \in \Omega_\rho(\varphi)$), and $S_1, \ldots, S_n$ and S_l and S_r should not contain any occurrence from $\Omega_\rho(\varphi)$. Moreover, an innermost rewriting strategy is enforced: only minimal reducible substructs (i.e. structs having no reducible proper substruct) can be rewritten*[6]*.*

$$S \otimes_\rho (S_1 \oplus \ldots \oplus S_n \oplus S_{n+1} \oplus \ldots \oplus S_{n+m}) \underset{w}{\rightsquigarrow} S_1 \oplus \ldots \oplus S_n \oplus (S \otimes_\rho S_{n+1}) \oplus \ldots \oplus (S \otimes_\rho S_{n+m})$$

$$(S_1 \oplus \ldots \oplus S_n \oplus S_{n+1} \oplus \ldots \oplus S_{n+m}) \otimes_\rho S \underset{w}{\rightsquigarrow} S_1 \oplus \ldots \oplus S_n \oplus (S_{n+1} \otimes_\rho S) \oplus \ldots \oplus (S_{n+m} \otimes_\rho S)$$

$$S_l \otimes_\rho S_r \underset{w}{\rightsquigarrow} S_l \qquad S_l \otimes_\rho S_r \underset{w}{\rightsquigarrow} S_r \qquad S_l \oplus_\rho S_r \underset{w}{\rightsquigarrow} S_l \qquad S_l \oplus_\rho S_r \underset{w}{\rightsquigarrow} S_r$$

$$S \oplus_\rho S_r \underset{w}{\rightsquigarrow} S_r \qquad S_l \oplus_\rho S \underset{w}{\rightsquigarrow} S_l$$

The fact that normalization of the struct according to $\underset{w}{\rightsquigarrow}$ corresponds to inference swapping taking into account the independence of inferences is stated in Lemma 1. This complements the rather technical Definition 4 with a more intuitive understanding of the reason why it works.

[5] The proof rewriting system for swapping inferences is shown in Appendix C and defines the relation $\gg$.

[6] Note that m can be equal to zero, in which case the first two rewriting rules simply degenerate to:

$$S \otimes_\rho (S_1 \oplus \ldots \oplus S_n) \underset{w}{\rightsquigarrow} S_1 \oplus \ldots \oplus S_n \qquad (S_1 \oplus \ldots \oplus S_n) \otimes_\rho S \underset{w}{\rightsquigarrow} S_1 \oplus \ldots \oplus S_n$$

Lemma 1 (Correspondence between $\underset{w}{\rightsquigarrow}$ and $\gg$). *If φ is skolemized*[7] *and $\mathcal{S}_\varphi \underset{w}{\rightsquigarrow} S$, then there exists a proof ψ such that $\varphi \gg^* \psi$ and $\mathcal{S}_\psi = S$.*

Proof. The proof and an example of the correspondence are available in [21]. The first two struct rewriting rules correspond to a simple upward swapping of the inference ρ, where ρ is distributed only to some branches of the proof. The next four struct rewriting rules correspond to cases in which ancestors of both auxiliary formulas of ρ were introduced by weakening inferences. In such cases, it is possible to arbitrarily delete one of the branches of ρ and replace ρ by weakening inferences. Correpondingly, one of the substructs can be deleted. The last two struct rewriting rules correspond to cases in which ancestors of only one of the auxiliary formulas of ρ were introduced by weakening. It is then possible to delete a particular branch and its corresponding substruct.

Example 3. $\mathcal{S}_{\varphi^a}$ can be normalized as follows:

By inspecting φ^a, note that $\vee_l^3$ operates on formulas highlighted in grey and purple. Hence, $\Omega_{\vee_l^3}(\varphi^a)$ contains all formulas highlighted in grey and purple in φ^a (and also some of the formulas in black). Consequently, $\otimes_3$ should only be distributed to substructs that contain formulas highlighted in these colors, and that is why, for convenience, the color label ** was added on top of $\otimes_3$. The first rewriting step is shown below:

$$\begin{aligned}\mathcal{S}_{\varphi^a} \equiv\ & ((P^1 \oplus \neg P^1) \overset{**}{\otimes}_2 ((P^1 \oplus \neg P^1) \overset{**}{\otimes}_3 (P^2_- \oplus \neg P^2_-))) \overset{***}{\otimes}_1 (((P^1 \oplus \neg P^1) \overset{**}{\otimes}_5 ((P^1 \oplus \neg P^1) \overset{**}{\otimes}_6 (P^2_- \oplus \neg P^2_-))) \overset{***}{\otimes}_4 (P^2_+ \oplus \neg P^2_+)) \\ \underset{w}{\rightsquigarrow}\ & ((P^1 \oplus \neg P^1) \overset{**}{\otimes}_2 (P^2_- \oplus ((P^1 \oplus \neg P^1) \overset{**}{\otimes}_3 \neg P^2_-))) \overset{***}{\otimes}_1 (((P^1 \oplus \neg P^1) \overset{**}{\otimes}_5 ((P^1 \oplus \neg P^1) \overset{**}{\otimes}_6 (P^2_- \oplus \neg P^2_-))) \overset{***}{\otimes}_4 (P^2_+ \oplus \neg P^2_+))\end{aligned}$$

Note that $(P^1 \oplus \neg P^1)$ (which contains something highlighted in grey) was distributed only to $\neg P^2_-$ (which is highlighted in *purple*) but not to P^2_- (which is highlighted in neither of those colors but rather in *red-orange*). Analogously, in the next rewriting step, $\neg P^2_-$ is distributed only to $\neg P^1$, but not to P^1:

$$\begin{aligned}\ldots \underset{w}{\rightsquigarrow}\ & ((P^1 \oplus \neg P^1) \overset{**}{\otimes}_2 (P^2_- \oplus ((P^1 \oplus \neg P^1) \overset{**}{\otimes}_3 \neg P^2_-))) \overset{***}{\otimes}_1 (((P^1 \oplus \neg P^1) \overset{**}{\otimes}_5 ((P^1 \oplus \neg P^1) \overset{**}{\otimes}_6 (P^2_- \oplus \neg P^2_-))) \overset{***}{\otimes}_4 (P^2_+ \oplus \neg P^2_+)) \\ \underset{w}{\rightsquigarrow}\ & ((P^1 \oplus \neg P^1) \overset{**}{\otimes}_2 (P^1 \oplus P^2_- \oplus (\neg P^1 \overset{**}{\otimes}_3 \neg P^2_-))) \overset{***}{\otimes}_1 (((P^1 \oplus \neg P^1) \overset{**}{\otimes}_5 ((P^1 \oplus \neg P^1) \overset{**}{\otimes}_6 (P^2_- \oplus \neg P^2_-))) \overset{***}{\otimes}_4 (P^2_+ \oplus \neg P^2_+))\end{aligned}$$

The rest of the normalization procedure is analogous, as shown below:

$$\begin{aligned}\ldots \underset{w}{\rightsquigarrow}^* & (P^1 \oplus P^1 \oplus (\neg P^1 \overset{**}{\otimes} P^2_-) \oplus (\neg P^1 \overset{**}{\otimes} \neg P^2_-)) \overset{***}{\otimes} (((P^1 \oplus \neg P^1) \overset{*}{\otimes} ((P^1 \oplus \neg P^1) \overset{*}{\otimes} (P^2_- \oplus \neg P^2_-))) \overset{**}{\otimes} (P^2_+ \oplus \neg P^2_+)) \\ \underset{w}{\rightsquigarrow}^* & (P^1 \oplus P^1 \oplus (\neg P^1 \overset{**}{\otimes} P^2_-) \oplus (\neg P^1 \overset{**}{\otimes} \neg P^2_-)) \overset{***}{\otimes} (((P^1 \oplus \neg P^1) \overset{*}{\otimes} (P^1 \oplus P^2_- \oplus (\neg P^1 \overset{**}{\otimes} \neg P^2_-))) \overset{**}{\otimes} (P^2_+ \oplus \neg P^2_+)) \\ \underset{w}{\rightsquigarrow}^* & (P^1 \oplus P^1 \oplus (\neg P^1 \overset{**}{\otimes} P^2_-) \oplus (\neg P^1 \overset{**}{\otimes} \neg P^2_-)) \overset{***}{\otimes} (((P^1 \oplus P^1 \oplus (\neg P^1 \overset{*}{\otimes} P^2_-) \oplus (\neg P^1 \overset{**}{\otimes} \neg P^2_-)) \overset{**}{\otimes} (P^2_+ \oplus \neg P^2_+)) \\ \underset{w}{\rightsquigarrow}^* & (P^1 \oplus P^1 \oplus (\neg P^1 \overset{**}{\otimes} P^2_-) \oplus (\neg P^1 \overset{**}{\otimes} \neg P^2_-)) \overset{***}{\otimes} (((P^1 \overset{**}{\otimes} \neg P^2_+) \oplus (P^1 \overset{***}{\otimes} \neg P^2_+) \oplus (\neg P^1 \overset{*}{\otimes} P^2_-) \oplus (\neg P^1 \overset{**}{\otimes} \neg P^2_-) \oplus P^2_+) \\ \underset{w}{\rightsquigarrow}^* & ((P^1 \overset{***}{\otimes} P^2_+) \oplus (P^1 \overset{***}{\otimes} P^2_+) \oplus (\neg P^1 \overset{**}{\otimes} P^2_-) \oplus (\neg P^1 \overset{**}{\otimes} \neg P^2_-)) \oplus ((P^1 \overset{**}{\otimes} \neg P^2_+) \oplus (P^1 \overset{***}{\otimes} \neg P^2_+) \oplus (\neg P^1 \overset{*}{\otimes} P^2_-) \oplus (\neg P^1 \overset{**}{\otimes} \neg P^2_-)) \\ \equiv\ & (P^1 \overset{***}{\otimes} P^2_+) \oplus (P^1 \overset{***}{\otimes} P^2_+) \oplus (\neg P^1 \overset{**}{\otimes} P^2_-) \oplus (\neg P^1 \overset{**}{\otimes} \neg P^2_-) \oplus (P^1 \overset{***}{\otimes} \neg P^2_+) \oplus (P^1 \overset{***}{\otimes} \neg P^2_+) \oplus (\neg P^1 \overset{*}{\otimes} P^2_-) \oplus (\neg P^1 \overset{**}{\otimes} \neg P^2_-)\end{aligned}$$

[7] A proof is skolemized if and only if it contains no $\forall_r$ or $\exists_l$ inferences operating on ancestors of formulas occurring in the end-sequent. Skolemization of proofs plays no role in the propositional proofs of the examples shown in this paper, and hence it is not discussed in detail here. There are algorithms that can easily skolemize first-order proofs; skolemization of higher-order proofs, on the other hand, is significantly harder [20].

Definition 5 (Swapped Clause Set). *A* swapped clause set[8] $C^W_{\varphi|S}$ *of a proof* φ *with respect to a* $\underset{w}{\rightsquigarrow}$*-normal-form* S *of* $\mathcal{S}_\varphi$ *is the set of clauses (in sequent notation) obtained from* S *by interpreting* $\otimes$ *as* $\vee$ *and* $\oplus$ *as* $\wedge$*. In cases where a proof* φ *has only one cut-pertinent swapped clause set, it can be denoted simply as* C^W_φ*.*

Example 4. The swapped clause set $C^W_{\varphi^a}$ is shown below. Note how each $\oplus$-junct of the normal form of $\mathcal{S}_{\varphi^a}$ shown in Example 3 corresponds to a clause in $C^W_{\varphi^a}$.

$$C^W_{\varphi^a} \equiv \{ \vdash P^1, P^2_+ \;;\; \vdash P^1, P^2_+ \;;\; P^1 \vdash P^2_- \;;\; P^1, P^2_- \vdash \;;\; P^2_+ \vdash P^1 \;;\; P^2_+ \vdash P^1 \;;\; P^1 \vdash P^2_- \;;\; P^1, P^2_- \vdash \}$$

3.4 Step 4: Refutation of the Swapped Clause Set by Resolution

The fourth step is the search for a resolution refutation of the swapped clause set.

Example 5. $C^W_{\varphi^a}$ can be refuted by the refutation δ below:

$$\dfrac{\dfrac{\dfrac{\vdash P^1, P^2_+}{\vdash P^1, P^1}\, r \qquad \dfrac{P^2_+ \vdash P^1}{\vdash P^1, P^1}\, r}{\vdash P^1}\, f_r \qquad \dfrac{\dfrac{P^1 \vdash P^2_-}{P^1, P^1 \vdash}\, r \qquad \dfrac{P^1, P^2_- \vdash}{P^1, P^1 \vdash}\, r}{P^1 \vdash}\, f_r}{\vdash}\, r$$

3.5 Step 5: Construction of O-Projections

A projection's purpose is to replace a leaf in a refutation of the clause set. Therefore, its end-sequent must contain the leaf's clause as a subsequent. Moreover, if we consider the composition of projections on the refutation described in Subsection 3.6, it is clear that any other formula F in the end-sequent of a projection will also appear in the end-sequent of the proof with atomic cuts produced by CIRes. Since the end-sequent of the proof produced by CIRes should be the same as the end-sequent of the original proof without cuts, it must not be the case that the end-sequent of a projection contains a formula F that does not already appear in the end-sequent of the original proof without cuts or in the leaf clause that the projection will replace. These conditions are formally expressed in Definition 6.

Definition 6 (Projection). *Let* φ *be a proof with end-sequent* $\Gamma \vdash \Delta$ *and* $c \equiv \Gamma_c \vdash \Delta_c \in C^W_\varphi$*. Any cut-free proof of* $(\Gamma', \Gamma_c \vdash \Delta', \Delta_c)\sigma$*, where* $\Gamma' \subseteq \Gamma$*,* $\Delta' \subseteq \Delta$ *and* σ *is a substitution, is a* projection *of* φ *with respect to* c*.*

[8] Historically, swapped clause sets were developed during an attempt to give a more intuitive definition for *profile clause sets* [13]. The deeper understanding acquired during this attempt allowed the development of swapped clause sets, which are technically better than profile clause sets in handling proofs with weakening inferences [21].

The standard method for the construction of projections is usually presented together with descriptions of the CERes method [3,7,4,14]. It can be easily described, but unfortunately results in redundant projections, because parts of φ tend to appear several times in different projections of φ, even though it would suffice if they appeared in only one of these projections. Projections of this standard but redundant kind are here called *S-projections*. This paper describes an alternative method that constructs so-called *O-projections* (Definition 9). They are less redundant and thus essential for compressing proofs via CIRes. Their construction relies on the auxiliary Y rule (Definition 7) and on the concept of *axiom-linkage* (Definition 8).

Definition 7 (Y Rule). *The Y rule of inference is shown below:*

$$\dfrac{\begin{array}{ccc}\varphi_1 & & \varphi_n \\ \Gamma_1 \vdash \Delta_1 & \dots & \Gamma_n \vdash \Delta_n\end{array}}{\Gamma_1,\dots,\Gamma_n \vdash \Delta_1 \dots,\Delta_n}\, Y$$

Definition 8 (Axiom Linkage). *Two atomic (sub)formulas A_1 and A_2 in a proof φ are* axiom-linked[9] *if and only if they have ancestors in the same axiom sequent.*

Definition 9 (O-Projection). *The* O-projection[10] $\lfloor\varphi\rfloor_c^O$ *of the proof φ with respect to the clause c is constructed according to the following steps:*

1. *replace inferences that operate on cut-ancestors by Y-inferences.*
2. *replace by Y-inferences also those inferences such that none of its active formulas is axiom-linked to a formula of φ appearing in c.*
3. *delete formulas that are not axiom-linked to the formulas appearing in c.*
4. *if the previous step deleted an auxiliary formula of an inference, fix the inference by adding a weakening inference that re-introduces the auxiliary formula.*
5. *eliminate the Y-inferences, according to Definition 10.*

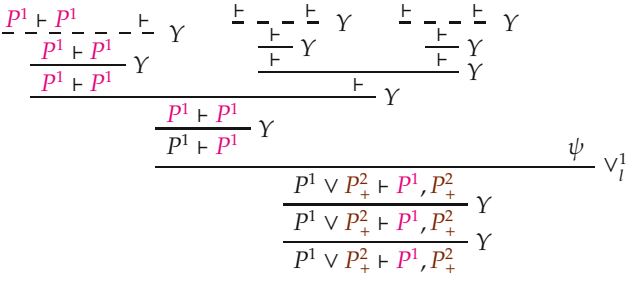
Definition 10 (Y-Elimination). *The elimination of Y inferences follows the proof rewriting rule shown below. For the rewriting rule to be applicable, φ_j is required to be a correct proof containing no Y-inferences.*

$$\dfrac{\begin{array}{ccc}\varphi_1 & & \varphi_n \\ \Gamma_1 \vdash \Delta_1 & \dots & \Gamma_n \vdash \Delta_n\end{array}}{\Gamma_1,\dots,\Gamma_n \vdash \Delta_1 \dots,\Delta_n}\, Y \quad \Downarrow \quad \dfrac{\begin{array}{c}\varphi_j \\ \Gamma_j \vdash \Delta_j\end{array}}{\Gamma_1,\dots,\Gamma_n \vdash \Delta_1 \dots,\Delta_n}\, w^*$$

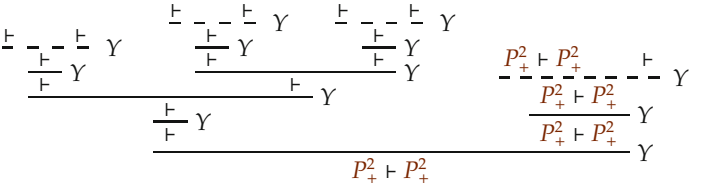
[9] By definition of axiom-linkage, it is clear that formulas highlighted with the same color in Example 1 are axiom-linked to each other.

[10] A technically more detailed definition of O-projection is available in [21].

Example 6. In the construction of the projection $\lfloor \varphi^a \rfloor^O_{\vdash P^1, P^2_+}$, the first four steps result in the (partial) proof with Y inferences shown below. Note that the only remaining formulas are those axiom linked to P^1 and P^2_+. All other were deleted. And the only remaining non-Y inference is $\vee^1_l$, because it is the only one that operates on formulas that are axiom-linked to P^1 and P^2_+.



where ψ is:



The last step in the construction of O-projections is the elimination of Y-inferences, and it gives the O-projection $\lfloor \varphi^a \rfloor^O_{\vdash P^1, P^2_+}$ shown below. The projections with respect to the other three clauses that appear in the leaves of the refutation are also shown below:

$\lfloor \varphi^a \rfloor^O_{\vdash P^1, P^2_+}$:

$$\frac{P^1 \vdash P^1 \qquad P^2_+ \vdash P^2_+}{P^1 \vee P^2_+ \vdash P^1, P^2_+} \vee_l$$

$\lfloor \varphi^a \rfloor^O_{P^2_+ \vdash P^1}$:

$$\frac{P^1 \vdash P^1 \qquad \dfrac{P^2_+ \vdash P^2_+}{P^2_+, \neg P^2_+ \vdash} \neg_l}{P^2_+, P^1 \vee \neg P^2_+ \vdash P^1} \vee_l$$

$\lfloor \varphi^a \rfloor^O_{P^1 \vdash P^2_-}$:

$$\frac{\dfrac{P^1 \vdash P^1}{P^1, \neg P^1 \vdash} \neg_l \qquad P^2_- \vdash P^2_-}{P^1, \neg P^1 \vee P^2_- \vdash P^2_-} \vee_l$$

$\lfloor \varphi^a \rfloor^O_{P^1, P^2_- \vdash}$:

$$\frac{\dfrac{P^1 \vdash P^1}{P^1, \neg P^1 \vdash} \neg_l \qquad \dfrac{P^2_- \vdash P^2_-}{P^2_-, \neg P^2_- \vdash} \neg_l}{P^1, P^2_-, \neg P^1 \vee \neg P^2_- \vdash} \vee_l$$

Theorem 1 (Correctness of O-Projections). *If φ is a skolemized proof, then $\lfloor\varphi\rfloor_c^O$ is a projection of φ with respect to c.*

Proof. In order to prove this theorem, it is necessary to show that the algorithm for construction of O-projections described in Definition 9 outputs a proof that satisfies the requirements expressed in Definition 6. A detailed technical proof is available in [21], and only a few informal remarks are presented here. Note that $\lfloor\varphi\rfloor_c^O$ is obviously cut-free, because of step 1 in Definition 9. Step 1 also guarantees that c appears as a subsequent of the end-sequent of $\lfloor\varphi\rfloor_c^O$: the atoms of c originate from atomic formulas that occur in axiom sequents and are ancestors of cut-formulas, and since all inferences that operate on ancestors of cut-formulas are replaced by Y-inferences, these atoms are free to propagate down to the end-sequent of $\lfloor\varphi\rfloor_c^O$ (i.e. they will not be used by propositional or quantifier inferences to compose more complex formulas and they will not be consumed by cut inferences, because all these inferences are replaced). Step 3 guarantees that propagated atoms of other clauses of C_φ are deleted from the end-sequent of $\lfloor\varphi\rfloor_c^O$, so that only the propagated atoms of c remain. Moreover, note that $\lfloor\varphi\rfloor_c^O$ will still contain those inferences of φ that operate on formulas that are axiom-linked to formulas of c and are not ancestors of cut-formulas. If these formulas are not ancestors of cut-formulas, they have (also axiom-linked) descendants occurring in the end-sequent of φ, which were not deleted by any step in the construction of the projection. Therefore, the end-sequent of $\lfloor\varphi\rfloor_c^O$ is of the form $\Gamma', \Gamma_c \vdash \Delta', \Delta_c$, where $\Gamma' \vdash \Delta'$ is the subsequent of the end-sequent of φ whose formulas are axiom-linked to formulas that appear in c, and $\Gamma_c \vdash \Delta_c = c$, since other propagated atoms are deleted by step 3. φ is required to be skolemized in order to prevent violations of eigen-variable conditions by the atoms that are propagated down after step 1.

3.6 Step 6: Composing the O-Projections on Top of the Refutation

The last step is the replacement of each leaf of the ground[11] refutation by its corresponding projection, and the renaming of resolution inferences to cuts and of factoring inferences to contractions. It may be necessary to add contractions and weakening inferences in the bottom of the resulting proof in order to correct the multiplicity of the formulas in its end-sequent (i.e. to guarantee that each formula appears as many times as it appeared in the input proof).

Example 7. The final proof with atomic cuts, obtained by composing the refutation and the projections, is $\mathsf{CIRes}_W^O(\varphi, \delta)$ shown below:

[11] Although the example shown here is purely propositional, CIRes works in predicate logic too. In this case, it is necessary to ground the resolution refutation by applying a substitution that is the composition of all most general unifiers used in the refutation. Only then the resolution inferences can be converted to atomic cuts. The grounding substitution must be applied to the projections too.

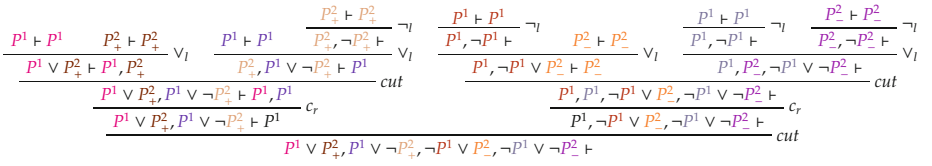


Table 1 compares the sizes of φ and $\mathsf{CIRes}^{O}_{W}(\varphi, \delta)$, and thus shows that CIRes is indeed able to compress proofs. Three different measures are used: *proof length*, which is the number of inferences in the proof; *symbolic proof size*, which counts the total number of symbols in formulas occurring in the proof; and *atomic proof size*, which counts only the total number of predicate symbols in formulas occurring in the proof.

Table 1. Compression by CIRes

	φ	$\mathsf{CIRes}^{O}_{W}(\varphi, \delta)$
Proof Length	17	13
Symbolic Proof Size	169	105
Atomic Proof Size	97	70

Theorem 2 (Correctness of CIRes). *Let ψ be a skolemized proof with end-sequent s. Then, for any refutation δ of any swapped clause set $C^{W}_{\psi^{\sigma}|S}$, $\mathsf{CIRes}^{O}_{W}(\psi, \delta)$ is a correct proof (with atomic cuts) with end-sequent s.*

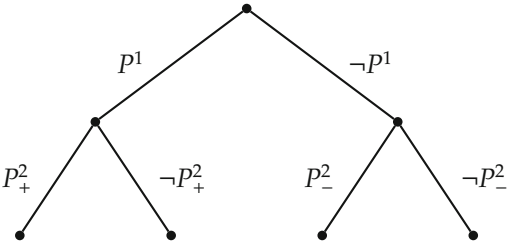
Proof. The correctness of CIRes follows immediately from the correctness of CERes. Note that every inference of $\mathsf{CIRes}^{O}_{W}(\psi, \delta)$ is either a cut or contraction originating from resolution or factoring inferences in the refutation of the clause, or an inference occurring in a projection, or a contraction or weakening to fix the multiplicity of formulas in the end-sequent. In any of these cases, the inference is clearly sound: inferences occurring in projections are sound, due to the correctness of projections; and the cuts and contractions are sound, given that the ground resolution refutation of the clause set is correct. The end-sequent of any projection contains only formulas of s or formulas of a clause of the end-sequent, but these are all eliminated by the resolutions/cuts in the refutation. Hence, the only formulas that remain in the end-sequent after the projections are combined with the refutation are formulas of s, possibly with a different multiplicity. The sound contractions and weakenings in the bottom of $\mathsf{CIRes}^{O}_{W}(\psi, \delta)$ therefore guarantee that the end-sequent of $\mathsf{CIRes}^{O}_{W}(\psi, \delta)$ is exactly s.

4 Exponential Proof Compression

The following lemmas and theorems use a set of disjunctions D_m that is associated with the complete binary tree of depth m, as described in [9,19]. D_m contains 2^m disjunctions of the form[12] $\circ P^1 \vee \circ P^2_{\pm} \vee \circ P^3_{\pm\pm} \vee \ldots \vee \circ P^m_{\pm\ldots\pm}$, where $\circ$ is either

[12] Parentheses have been ommited from these disjunctions, and the $\vee$ connective is assumed to be left-associative.

empty or $\neg$ and the i-th $\pm$ is either $+$, if the $\circ$ preceding $P^i_{\pm\ldots\pm}$ is empty, or $-$, if the $\circ$ preceding $P^i_{\pm\ldots\pm}$ is $\neg$. For example, $D_2 = \{P^1 \vee P^2_+, \neg P^1 \vee P^2_-, P^1 \vee \neg P^2_+, \neg P^1 \vee \neg P^2_-\}$. C_m is defined as the set of clauses corresponding to the disjunctions of D_m (e.g. $C_2 = \{\vdash P^1, P^2_+ \;;\; P^1 \vdash P^2_- \;;\; P^2_+ \vdash P^1 \;;\; P^1, P^2_- \vdash\}$). And T_m is defined as the sequent having all the disjunctions of C_m in its antecedent (e.g. $T_2 = P^1 \vee P^2_+, \neg P^1 \vee P^2_-, P^1 \vee \neg P^2_+, \neg P^1 \vee \neg P^2_- \vdash$).



Note that T_2 is exactly the end-sequent of the proofs considered in the examples of the previous section. The asymptotic results about the compression achievable by CIRes are obtained by quantitatively analyzing what happens in the general case, when CIRes is applied to T_m. The general phenomenon is essentially the same as what has been observed for T_2, and hence the examples of the previous section are helpful for the comprehension of the lemmas in this section.

Lemma 2. *Let ψ_m be a shortest analytic tableaux refutation of D_m. Then $length(\psi_m) > 2^{k_1 2^m}$, for some positive rational constant k_1.*

Proof. This lemma was mentioned in [9] and then proved in [19].

Lemma 3. *Let φ_m be the shortest cut-free sequent calculus proof of T_m corresponding to ψ_m. Then $length(\varphi_m) > 2^{k_2 2^m}$,for some positive rational constant k_2.*

Proof. This lemma follows immediately from Lemma 2 and from the fact that cut-free sequent calculus p-simulates analytic tableaux [19].

Lemma 4. *Let δ_m be the shortest resolution refutation of C_m. Then $length(\delta_m) < 2^{k_3 m}$, for some positive rational constant k_3.*

Proof. This lemma is mentioned without proof in [9]. Its proof is easy, though. δ_m can be constructed by resolving first the literals that correspond to the deepest nodes in the complete binary tree that generates D_m and C_m, then resolving the literals that correspond to the nodes on the level immediately above, and so on, until all literals have been resolved. In this way, it is clear that the number of resolution and factoring inferences in δ_m is linearly related to the number of nodes in the binary tree, which is exponential in m.

Lemma 5. $C_m \subseteq C^W_{\varphi^a_m}$.

Proof. Let $c_d \in C_m$. By definition of C_m, D_m and T_m, there is a disjunction $d \doteq L_1 \vee^1 \ldots \vee^{m-1} L_m$ in the antecedent of the end-sequent T_m of the proof φ^a_m such that c_d is the clause form of d in sequent notation. Let A_i be one of the atomic ancestors of L_i occurring in an axiom sequent s_i of φ^a_m. Let A'_i be the formula occurring in the other cedent of s_i. Note that:

- A_i and A'_i are syntactically equal, by definition of axiom sequents.
- A'_i is a cut-ancestor.
- A'_i occurs in the succedent of s_i, if L_i is a positive literal, and in the antecedent, otherwise.

Let S_i be the substruct of $\mathcal{S}_{\varphi^a_m}$ corresponding to the axiom rule having conclusion sequent s_i. By definition, $S_i = A'_i = L_i$, if A'_i occurs in the succedent of s_i, and $S_i = \neg A'_i = L_i$, otherwise. Let ρ_j be the $\vee_l$ inference in φ^a_m which operates on descendents of $A_1, \ldots, A_m$ and introduces the connective $\vee^j$ in the disjunction d. Let $\otimes_j$ be the connective in $\mathcal{S}_{\varphi^a_m}$ that corresponds to ρ_j. By Definition 4, the normalization of $\mathcal{S}_{\varphi^a_m}$ is such that $\otimes_j$ is distributed to substructs containing formulas that are axiom-linked to ancestors of formulas on which ρ_j operates (i.e. containing $A'_1, \ldots, A'_m$). Consequently, the normal form S of $\mathcal{S}_{\varphi^a_m}$ contains the substruct $S_d \doteq S_1 \otimes_1 \ldots \otimes_{m-1} S_m$. When S is transformed to $C^W_{\varphi^a_m}$, S_d becomes the clause c_d, because $S_i = L_i$, for all i, and $\otimes$ is interpreted as $\vee$. Therefore, $c_d \in C^W_{\varphi^a_m}$ and hence $C_m \subseteq C^W_{\varphi^a_m}$.

Lemma 6. *δ_m is a refutation of $C^W_{\varphi^a_m}$.*

Proof. This lemma follows immediately from Lemma 5.

Lemma 7. *Let $c \in C^W_{\varphi^a_m}$. Then $length(\lfloor \varphi^a_m \rfloor^O_c) < k_4 m$, for some positive constant k_4.*

Proof. By definition, the O-projection $\lfloor \varphi^a_m \rfloor^O_c$ contains only those inferences of φ^a_m that operate on descendants of axiom sequents that contain occurences of c. These are the inferences that construct one of the disjunctions in the end-sequent T_m of φ^a_m, namely the disjunction whose clause form in sequent notation is equal to c. c has exactly m literals, and thus $\lfloor \varphi^a_m \rfloor^O_c$ contains exactly $m-1$ inferences of $\vee_l$ kind. Since literals can appear negated, $\lfloor \varphi^a_m \rfloor^O_c$ can contain at most m inferences of $\neg_l$ kind. No other inferences appear in $\lfloor \varphi^a_m \rfloor^O_c$. Therefore, $length(\lfloor \varphi^a_m \rfloor^O_c) < 2m - 1$.

Theorem 3 (Exponential Proof Compression via CIRes). *There exists a sequence of sequents T_m such that:*

- *if φ_m is a sequence of shorthest cut-free proofs of T_m, then $length(\varphi_m) > 2^{k_5 2^m}$ (for some positive constant k_5).*
- *there exists δ_m such that $length(\mathrm{CIRes}(\varphi_m, \delta_m)) < m.2^{k_6 m}$ (for some positive constant k_6).*

Proof. The first item of this theorem is just Lemma 3. For the second item, let δ_m be the shortest refutation of C_m, as in Lemma 4. By lemma 6, δ_m is also a refutation of $C^W_{\varphi^a_m}$, and hence it can be used in the construction of the proof with atomic cuts by CIRes. Then note that CIRes(φ_m, δ_m) is the composition of δ_m, whose length is exponentially upperbounded (Lemma 4), and 2^m O-projections of linear size (as in Lemma 7). Therefore:

$$length(\text{CIRes}(\varphi_m, \delta_m)) < 2^{k_3 m} + 2^m(2m-1) < m.2^{k_6 m}$$

for some constant k_6.

5 Conclusions

This paper has introduced the CIRes method of cut-introduction and shown that it can compress proofs exponentially. This was only possible with the development of swapped clause sets and O-projections, which are less redundant than the standard clause sets and projections traditionally used by CERes. These new concepts could be employed for cut-elimination as well.

The further development of CIRes can proceed in various directions. Firstly, similarly to what has already been done for cut-elimination [21], the swapped clause set could be enriched with additional information from the proof, and this information could then be used to define refinements of the resolution calculus in order to facilitate the search for refutations and, consequently, the introduction of cuts.

Secondly, O-projections and the method for combining them with the refutation could still be improved. In the example considered in this paper, the (antecedent of the) end-sequent of cut-free proof is essentially already a set of clauses and the shortest proof with cuts is a proof whose atomic cuts all occur in the bottom. These facts are particularly convenient for a method like CIRes, that uses resolution and outputs proofs with cuts in the bottom. However, in other cases (i.e. when the optimal proof with atomic cuts is such that the atomic cuts do not occur in the bottom of the proof; or, equivalently, when the end-sequent of the cut-free proof is not in such a clause form), CIRes will produce sub-optimally compressed proofs, because the O-projections will contain redundancies that are only necessary because CIRes currently requires the atomic cuts to be in the very bottom and the projections to be on the top. This indicates that CIRes could be improved by developing different notions of projections and more flexible ways of composing them with the refutation. However, this is highly non-trivial.

Finally, much more significant (i.e. non-elementary) compression could in principle be obtained via introduction of quantified cuts. The CIRes method described in this chapter introduces only atomic cuts and is therefore just a first step toward the harder task of introducing complex quantified cuts. An intermediary step could be the introduction of propositional cuts, possibly by using definitional and swapped definitional clause sets [21]. But even then, (sub-optimal) compressive quantified-cut-introduction would still be a distant goal,

and an algorithm that would generally guarantee optimal compression is forever out of reach; it cannot exist, due to the undecidability results in [8].

References

1. Baaz, M., Ciabattoni, A., Fermüller, C.: Cut elimination for first order gödel logic by hyperclause resolution. In: Logic for Programming, Artificial Intelligence, and Reasoning, pp. 451–466 (2008)
2. Baaz, M., Egly, U., Leitsch, A.: Normal form transformations. In: Voronkov, A., Robinson, A. (eds.) Handbook of Automated Reasoning, pp. 275–333. Elsevier, Amsterdam (2001)
3. Baaz, M., Hetzl, S., Leitsch, A., Richter, C., Spohr, H.: Cut-Elimination: Experiments with CERES. In: Baader, F., Voronkov, A. (eds.) LPAR 2004. LNCS (LNAI), vol. 3452, pp. 481–495. Springer, Heidelberg (2005)
4. Baaz, M., Leitsch, A.: Cut-elimination and Redundancy-elimination by Resolution. Journal of Symbolic Computation 29(2), 149–176 (2000)
5. Baaz, M., Leitsch, A.: Comparing the complexity of cut-elimination methods. Proof Theory in Computer Science, 49–67 (2001)
6. Baaz, M., Leitsch, A.: Ceres in many-valued logics. In: Baader, F., Voronkov, A. (eds.) LPAR 2004. LNCS (LNAI), vol. 3452, pp. 1–20. Springer, Heidelberg (2005)
7. Baaz, M., Leitsch, A.: Methods of Cut-Elimination (2009) (to appear)
8. Baaz, M., Zach, R.: Algorithmic structuring of cut-free proofs. In: Martini, S., Börger, E., Kleine Büning, H., Jäger, G., Richter, M.M. (eds.) CSL 1992. LNCS, vol. 702, pp. 29–42. Springer, Heidelberg (1993)
9. Cook, S., Reckhow, R.: On the lengths of proofs in the propositional calculus (preliminary version). In: STOC 1974: Proceedings of the Sixth Annual ACM Symposium on Theory of Computing, pp. 135–148. ACM, New York (1974)
10. Egly, U., Genther, K.: Structuring of computer-generated proofs by cut introduction. In: Gottlob, G., Leitsch, A., Mundici, D. (eds.) KGC 1997. LNCS, vol. 1289, pp. 140–152. Springer, Heidelberg (1997)
11. Finger, M., Gabbay, D.M.: Equal rights for the cut: Computable non-analytic cuts in cut-based proofs. Logic Journal of the IGPL 15(5-6), 553–575 (2007)
12. Hetzl, S.: Proof Fragments, Cut-Elimination and Cut-Introduction (manuscript)
13. Hetzl, S.: Proof Profiles. Characteristic Clause Sets and Proof Transformations. VDM (2008)
14. 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)
15. Miller, D., Nigam, V.: Incorporating tables into proofs. In: Duparc, J., Henzinger, T.A. (eds.) CSL 2007. LNCS, vol. 4646, pp. 466–480. Springer, Heidelberg (2007)
16. Nigam, V.: Using tables to construct non-redundant proofs. In: CiE 2008: Abstracts and extended abstracts of unpublished papers (2008)
17. Orevkov, V.P.: Lower bounds for increasing complexity of derivations after cut-elimination (translation). Journal Sov. Math. (1982), 2337–2350 (1979)
18. Statman, R.: Lower bounds on Herbrand's theorem. Proceedings of the American Mathematical Society 75, 104–107 (1979)
19. Urquhart, A.: The complexity of propositional proofs. Bulletin of Symbolic Logic 1(4), 425–467 (1995)

20. Weller, D.: Skolemization Problems and Cut-elimination (In Preparation). PhD thesis, Technische Universitaet Wien (Vienna University of Technology) (2009)
21. Woltzenlogel Paleo, B.: A General Analysis of Cut-Elimination by CERes. PhD thesis, Vienna University of Technology (2009)
22. Woltzenlogel Paleo, B.: Physics and proof theory. In: International Workshop on Physics and Computation (2010)

Appendix A: Cut-Elimination Proof Rewriting Rules

Due to the page limit, the rewriting rules are included only in the extended electronic version available in `http://www.logic.at/people/bruno/`.

Appendix B: Inference Swapping

Due to the page limit, the inference swapping rewriting rules are included only in the extended electronic version available in `http://www.logic.at/people/bruno/`.

Satisfiability of Non-linear (Ir)rational Arithmetic*

Harald Zankl and Aart Middeldorp

Institute of Computer Science, University of Innsbruck, Austria

Abstract. We present a novel way for reasoning about (possibly ir)rational quantifier-free non-linear arithmetic by a reduction to SAT/SMT. The approach is incomplete and dedicated to satisfiable instances only but is able to produce models for satisfiable problems quickly. These characteristics suffice for applications such as termination analysis of rewrite systems. Our prototype implementation, called MiniSmt, is made freely available. Extensive experiments show that it outperforms current SMT solvers especially on rational and irrational domains.

Keywords: non-linear arithmetic, SMT solving, term rewriting, termination, matrix interpretations.

1 Introduction

Establishing termination of programs (automatically) is essential for many aspects of software verification. Contemporary termination analyzers for term rewrite systems (TRSs) rely on solving (non-)linear arithmetic, mostly—but not exclusively—over the natural numbers. Hence designated solvers for non-linear arithmetic are very handy when implementing termination criteria for term rewriting (e.g. recently in [7,13,14,20,31]) but also in different research domains dealing with verification (e.g. recently in [18]).

In this paper we explain the theory underlying our SMT solver MiniSmt, which is freely available under terms of the GNU lesser general public license version 3 from `http://cl-informatik.uibk.ac.at/software/minismt`. This tool is designed to find models for satisfiable instances of non-linear arithmetic quickly. Integral domains are handled by bit-blasting to SAT and, alternatively, by an appropriate transformation to bit-vector arithmetic before solvers for these logics are employed. Non-integral domains are also supported by a suitable reduction to the integral setting. To solve constraints over *rational* domains efficiently we propose a heuristic which is easy to implement. Experiments on various benchmarks show gains in power and efficiency compared to contemporary existing approaches. The support for irrational domains (by approximating comparisons involving $\sqrt{2}$) distinguishes our tool.

We expect two major effects of our contribution: MiniSmt eases the job to develop a new termination tool (fast reasoning about arithmetic is also relevant

* This research is supported by FWF (Austrian Science Fund) project P18763.

E.M. Clarke and A. Voronkov (Eds.): LPAR-16, LNAI 6355, pp. 481–500, 2010.

for implementing other termination criteria) and our test benches will spark further research in the SMT community on non-linear arithmetic.

The remainder of the paper is organized as follows. How we reduce non-linear non-integral arithmetic over (possibly ir)rational domains to integral arithmetic is outlined in Section 2. Experiments showing the benefit of our approach are presented in Section 3 before Section 4 compares our approach with related work. For the convenience of the reader an encoding of (bounded) integral non-linear arithmetic in SAT is given in Appendix A. These encodings are similar to known ones but take overflows into account. To obtain benchmarks for non-integral domains we generalize a popular termination criterion for rewrite systems—matrix interpretations [13,20]—to non-negative *real* coefficients in Appendix B. To automate the method, models for non-linear arithmetic constraints must be found quickly for satisfiable instances.

2 Encoding Non-linear Non-integral Arithmetic

In this section we introduce a grammar for non-linear arithmetic constraints (which appear when automating matrix interpretations, among other termination criteria for term rewriting) and show how to reduce constraints over non-integral arithmetic to the integral case.

Definition 1. *An* arithmetic constraint φ *is described by the BNFs*

$$\varphi ::= \bot \mid \top \mid p \mid (\neg\varphi) \mid (\varphi \circ \varphi) \mid (\alpha \star \alpha) \quad \textit{and} \quad \alpha ::= \mathrm{a} \mid r \mid (\alpha \diamond \alpha) \mid (\varphi\,?\,\alpha : \alpha)$$

where $\circ \in \{\vee, \wedge, \to, \leftrightarrow\}$, $\star \in \{>, =\}$, *and* $\diamond \in \{+, -, \times\}$.

Here $\bot$ ($\top$) denotes contradiction (tautology), p (a) ranges over Boolean (arithmetical) variables, $\neg$ ($\vee$, $\wedge$, $\to$, $\leftrightarrow$) is logical not (or, and, implication, bi-implication), $>$ ($=$) greater (equal), r ranges over the real numbers, and $+$ ($-$, $\times$) denotes addition (subtraction, multiplication). If-then-else is written as $(\cdot\,?\,\cdot : \cdot)$. The following example shows some (non-)well-formed constraints.

Example 2. The expressions 5, p_{100}, $(p_{10}\,?\,(2.1 \times \mathrm{a}_{12}) : 0)$, and $((((\mathrm{a}_{12} + (\sqrt{2} \times \mathrm{a}_{30})) + 7.2) > (0 - \mathrm{a}_5)) \wedge p_2)$ are well-formed whereas $-\mathrm{a}_{10}$ (unary minus) and $\mathrm{a} + 3$ (parenthesis missing) are not.

The binding precedence $\times \succ +, - \succ >, = \succ \neg \succ \vee, \wedge \succ \to, \leftrightarrow, (\cdot\,?\,\cdot : \cdot)$ allows to save parentheses. Furthermore the operators $+$, $\times$, $\vee$, $\wedge$, and $\leftrightarrow$ are left-associative while $-$ and $\to$ associate to the right. Taking these conventions into account the most complex constraint from the previous example simplifies to $\mathrm{a}_{12} + \sqrt{2} \times \mathrm{a}_{30} + 7.2 > 0 - \mathrm{a}_5 \ \wedge \ p_2$. To obtain smaller constraints already at the time of encoding trivial simplifications like $\varphi \wedge \top \to \varphi$, $\varphi \wedge \bot \to \bot$, $\cdots$ are performed.

In Appendix A we show how to mimic arithmetic over $\mathbb{N}$ and $\mathbb{Z}$ in SAT. Similar encodings have been presented (either for fixed bit width or for non-negative numbers only) in [13,14,23,31]. To our knowledge the two's complement

encoding taking overflows into account is new. In the remainder of this section we show how arithmetic over $\mathbb{Q}$ and (a fragment of) $\mathbb{R}$ can be reduced to the integral case. By $\mathbf{N}$, $\mathbf{Z}$, $\mathbf{Q}$, and $\mathbf{R}$ we denote the encodings of numbers from $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, and $\mathbb{R}$, respectively. To clarify the domain we index operations by these sets whenever confusion can arise.

2.1 Rational Arithmetic

Rational numbers are represented as a pair consisting of the numerator and denominator similar as in [15]. The numerator is a bit-vector representing an integer (compared to a natural number in [15]) whereas the denominator is a positive integer (negative denominators would demand a case analysis for $>_{\mathbf{Q}}$). We also experimented with a fixed point representation, yielding slightly worse performance and less flexibility. All operations with the exception of $\times_{\mathbf{Q}}$ require identical denominators. This can easily be established by expanding the fractions beforehand (as demonstrated in Example 4).

Comparisons are performed just on the numerators if the denominators coincide. The operations $+_{\mathbf{Q}}$, $-_{\mathbf{Q}}$, and $\times_{\mathbf{Q}}$ are inspired from arithmetic over fractions.

Definition 3. *For* $(\mathbf{a}, d)$, $(\mathbf{b}, d)$, *and* $(\mathbf{b}, d')$ *representing rationals we define:*

$$\begin{aligned}
(\mathbf{a}, d) >_{\mathbf{Q}} (\mathbf{b}, d) &:= \mathbf{a} >_{\mathbf{Z}} \mathbf{b} \\
(\mathbf{a}, d) =_{\mathbf{Q}} (\mathbf{b}, d) &:= \mathbf{a} =_{\mathbf{Z}} \mathbf{b} \\
(\mathbf{a}, d) +_{\mathbf{Q}} (\mathbf{b}, d) &:= (\mathbf{a} +_{\mathbf{Z}} \mathbf{b}, d) \\
(\mathbf{a}, d) -_{\mathbf{Q}} (\mathbf{b}, d) &:= (\mathbf{a} -_{\mathbf{Z}} \mathbf{b}, d) \\
(\mathbf{a}, d) \times_{\mathbf{Q}} (\mathbf{b}, d') &:= (\mathbf{a} \times_{\mathbf{Z}} \mathbf{b}, d \times d')
\end{aligned}$$

Next we demonstrate addition.

Example 4. Consider $\frac{3}{2} +_{\mathbb{Q}} \frac{-1}{4} = \frac{5}{4}$. In the sequence below first both denominators are made equal. Then addition of the numerators is performed using $+_{\mathbf{Z}}$ (see Appendix A for an explanation of the notation):

$$\begin{aligned}
(\langle \bot, \top, \top \rangle, 2) +_{\mathbf{Q}} (\langle \top, \top \rangle, 4) &= (\langle \bot, \top, \top, \bot \rangle, 4) +_{\mathbf{Q}} (\langle \top, \top \rangle, 4) \\
&= (\langle \bot, \top, \top, \bot \rangle +_{\mathbf{Z}} \langle \top, \top \rangle, 4) = (\langle \bot, \bot, \top, \bot, \top \rangle, 4)
\end{aligned}$$

We conclude this subsection by introducing a concept that drastically improves performance of rational arithmetic. Consider the following computation where (intermediate) results are not canceled:

$$\left(\frac{1}{2} \times \frac{4}{2}\right) \times \frac{3}{2} + \frac{1}{2} \;=\; \frac{4}{4} \times \frac{3}{2} + \frac{1}{2} \;=\; \frac{12}{8} + \frac{1}{2} \;=\; \frac{16}{8} \tag{1}$$

Exactly this happens in the implementation since there the numerator is a bit-vector consisting of Boolean formulas. Hence its concrete value is unknown and

no cancellation is possible. We propose the following elegant escape which is very easy to implement and has positive effects on run-times (as shown in Section 3). We force that a fraction is canceled if the denominator exceeds some given limit. Computation (2) shows the positive aspects of this heuristic by allowing a denominator of at most 2:

$$\left(\frac{1}{2} \times \frac{4}{2}\right) \times \frac{3}{2} + \frac{1}{2} \;=\; \frac{2}{2} \times \frac{3}{2} + \frac{1}{2} \;=\; \frac{3}{2} + \frac{1}{2} \;=\; \frac{4}{2} \tag{2}$$

After every addition or multiplication the fraction is canceled whenever the denominator exceeds 2. The negative aspects become apparent if the denominator is chosen too small. Then some computations can no longer be performed, e.g., when allowing a denominator of 1, computation (3) gets stuck in the second step since $\frac{3}{2}$ cannot be canceled:

$$\left(\frac{1}{2} \times \frac{4}{2}\right) \times \frac{3}{2} + \frac{1}{2} \;=\; \frac{1}{1} \times \frac{3}{2} + \frac{1}{2} \;=\; \frac{?}{1} + \frac{1}{2} \tag{3}$$

In the implementation, canceling by two is achieved by dividing the denominator by two and dropping the least significant bit of the numerator while demanding that this bit evaluates to false. The latter is achieved by adding a suitable constraint. Hence in contrast to the example above, computations do not get stuck but may produce unsatisfiable formulas. In Section 3 we will see that this does not happen very frequently. Furthermore, there also the effectiveness of this very simple but efficient heuristic is demonstrated.

Some remarks are in order. Although our representation of rationals allows fractions like $\frac{\mathbf{a}}{3}$, choosing the denominators as multiples of two is beneficial. This allows to efficiently extend fractions to equal denominators by bit-shifting the numerators. Furthermore, the heuristic (canceling by two) is most effective for even denominators. Obviously the technique extends to different denominators in principle but division by two can again be performed by bit-shifting while division by, e.g., three cannot and hence is more costly. However, for termination analysis the main benefit of rationals is that some number between zero and one can be represented while the exact value of this number is usually not so important.

2.2 Extending Rational Arithmetic by Roots of Numbers

Arithmetic over irrational numbers is the most challenging. To allow a finite representation we only consider a subset of $\mathbb{R}$ using a pair $(\mathbf{c}, \mathbf{d})$ where $\mathbf{c}$ and $\mathbf{d}$ are numbers from $\mathbf{Q}$. Such a pair $(\mathbf{c}, \mathbf{d})$ has the intended semantics of $\mathbf{c} + \mathbf{d}\sqrt{2}$. But problems arise when comparing two abstract numbers. Therefore the definition of $>_{\mathbf{R}}$ given below is just an approximation of $>_{\mathbb{R}}$. The idea is to under-approximate $\mathbf{d}\sqrt{2}$ on the left-hand side while over-approximating it on the right-hand side. We under-approximate $\mathbf{d}\sqrt{2}$ by $(\mathbf{5}, 4) \times_{\mathbf{Q}} \mathbf{d}$ if $\mathbf{d}$ is not negative and by $(\mathbf{3}, 2) \times_{\mathbf{Q}} \mathbf{d}$ if $\mathbf{d}$ is negative.[1] The approach is justified since $\frac{5}{4} = 1.25 <_{\mathbb{R}} 1.41 \approx \sqrt{2}$ and

[1] We abbreviate numbers by denoting them in boldface, i.e., $\mathbf{5}$ represents $\langle \bot, \top, \bot, \top \rangle$ from $\mathbf{Z}$ and $(\langle \bot, \top, \bot, \top \rangle, 1)$ from $\mathbf{Q}$. The context clarifies which one is meant.

similarly $-\frac{3}{2} = -1.5 <_{\mathbb{R}} -1.41 \approx -\sqrt{2}$. Analogous reasoning yields the over-approximation. This trick allows to implement $>_{\mathbf{R}}$ (an approximation of $>_{\mathbb{R}}$) based on $>_{\mathbf{Q}}$ which can be expressed exactly (cf. Definition 3).

Next we formally define the under- and over-approximation of $\mathbf{a}\sqrt{2}$ based on $\mathbf{a}$ from $\mathbf{Q}$ depending on the sign (denoted $\mathrm{sign}(\mathbf{a})$ with obvious definition) using the if-then-else operator. Recall that sign $\top$ indicates negative numbers (cf. Appendix A).

Definition 5. *For a number* $\mathbf{a}$ *from* $\mathbf{Q}$ *we define:*

$$\begin{aligned} \mathrm{under}(\mathbf{a}) &:= (\mathrm{sign}(\mathbf{a})\,?\,(\mathbf{3},2) : (\mathbf{5},4)) \times_{\mathbf{Q}} \mathbf{a} \\ \mathrm{over}(\mathbf{a}) &:= (\mathrm{sign}(\mathbf{a})\,?\,(\mathbf{5},4) : (\mathbf{3},2)) \times_{\mathbf{Q}} \mathbf{a} \end{aligned}$$

Using the under- and over-approximations we define $>_{\mathbf{R}}$ and $=_{\mathbf{R}}$. Note that since $>_{\mathbf{R}}$ is just an approximation, it may not appear at negative positions in Boolean formulas. We remark that designing suitable approximations for $>_{\mathbf{R}}$ at negative positions is easy and, moreover, that $>_{\mathbf{R}}$ does not appear at negative positions in the benchmarks that we consider.

Definition 6. *For pairs* $(\mathbf{c},\mathbf{d})$ *and* $(\mathbf{e},\mathbf{f})$ *from* $\mathbf{R}$ *we define:*

$$\begin{aligned} (\mathbf{c},\mathbf{d}) >_{\mathbf{R}} (\mathbf{e},\mathbf{f}) &:= \mathbf{c} +_{\mathbf{Q}} \mathrm{under}(\mathbf{d}) >_{\mathbf{Q}} \mathbf{e} +_{\mathbf{Q}} \mathrm{over}(\mathbf{f}) \\ (\mathbf{c},\mathbf{d}) =_{\mathbf{R}} (\mathbf{e},\mathbf{f}) &:= \mathbf{c} =_{\mathbf{Q}} \mathbf{e} \wedge \mathbf{d} =_{\mathbf{Q}} \mathbf{f} \end{aligned}$$

For readability we unravel the pair notation in the sequel whenever useful, i.e., $(\mathbf{3},\mathbf{1})$ is identified with $3 +_{\mathbb{R}} \sqrt{2}$.

Example 7. The expression $(\mathbf{1},\mathbf{1}) >_{\mathbf{R}} (\mathbf{2},\mathbf{0})$ approximates $1+\sqrt{2} >_{\mathbb{R}} 2$. The $\sqrt{2}$ on the left-hand side is under-approximated by $\frac{5}{4}$ which allows to replace $>_{\mathbb{R}}$ by $>_{\mathbb{Q}}$. The resulting $1+\frac{5}{4} >_{\mathbb{Q}} 2$, i.e., $\frac{9}{4} >_{\mathbb{Q}} \frac{8}{4}$ shows that the above comparison is valid. Note that $(\mathbf{0},\mathbf{6}) >_{\mathbf{R}} (\mathbf{0},\mathbf{5})$ does not hold since obviously $6 \times \frac{5}{4} >_{\mathbb{Q}} 5 \times \frac{3}{2}$ evaluates to false.

The definitions for $+_{\mathbf{R}}$, $-_{\mathbf{R}}$, and $\times_{\mathbf{R}}$ are directly inspired from the intended semantics of pairs.

Definition 8. *For pairs* $(\mathbf{c},\mathbf{d})$ *and* $(\mathbf{e},\mathbf{f})$ *from* $\mathbf{R}$ *we define:*

$$\begin{aligned} (\mathbf{c},\mathbf{d}) +_{\mathbf{R}} (\mathbf{e},\mathbf{f}) &:= (\mathbf{c} +_{\mathbf{Q}} \mathbf{e}, \mathbf{d} +_{\mathbf{Q}} \mathbf{f}) \\ (\mathbf{c},\mathbf{d}) -_{\mathbf{R}} (\mathbf{e},\mathbf{f}) &:= (\mathbf{c} -_{\mathbf{Q}} \mathbf{e}, \mathbf{d} -_{\mathbf{Q}} \mathbf{f}) \\ (\mathbf{c},\mathbf{d}) \times_{\mathbf{R}} (\mathbf{e},\mathbf{f}) &:= (\mathbf{c} \times_{\mathbf{Q}} \mathbf{e} +_{\mathbf{Q}} \mathbf{2} \times_{\mathbf{Q}} \mathbf{d} \times_{\mathbf{Q}} \mathbf{f}, \mathbf{c} \times_{\mathbf{Q}} \mathbf{f} +_{\mathbf{Q}} \mathbf{d} \times_{\mathbf{Q}} \mathbf{e}) \end{aligned}$$

The next example demonstrates addition and multiplication for reals.

Example 9. The equality $(\mathbf{1},\mathbf{2}) +_{\mathbf{R}} (\mathbf{5},\mathbf{3}) = (\mathbf{6},\mathbf{5})$ is justified since the left-hand side represents the calculation $1+2\sqrt{2}+5+3\sqrt{2}$ which simplifies to $6+5\sqrt{2}$ corresponding to the right-hand side. The product $(\mathbf{1},\mathbf{2}) \times_{\mathbf{R}} (\mathbf{5},\mathbf{3}) = (\mathbf{17},\mathbf{13})$ is justified by $(1+2\sqrt{2}) \times (5+3\sqrt{2}) = 5+10\sqrt{2}+3\sqrt{2}+6\sqrt{2}\sqrt{2} = 17+13\sqrt{2}$.

A natural question is if the approach from this section can be extended to a larger fragment of the reals. Before the limitations of the approach are discussed we mention possible generalizations. Triples $(\mathbf{c}, \mathbf{d}, \mathbf{n})$ with $\mathbf{c}, \mathbf{d} \in \mathbf{Q}$ and $\mathbf{n} \in \mathbf{N}$ (i.e., $\mathbf{n}$ is a variable taking non-negative integral values) allow to represent numbers of the shape $\mathbf{c} + \mathbf{d}\sqrt{\mathbf{n}}$. Adapting the constant factor for multiplication in Definition 8 and providing suitable under- and over-approximations for $\sqrt{\mathbf{n}}$ allows to replace $\sqrt{2}$ by a square root of some arbitrary natural number. But intrinsic to the approach is that the same (square) root must be used within all constraints to keep the triple representation of numbers. The reason is that e.g., $(\mathbf{a}, \mathbf{b}, \mathbf{n}) \times_{\mathbf{R}} (\mathbf{c}, \mathbf{d}, \mathbf{n}) = (\mathbf{ac} +_{\mathbf{R}} \mathbf{nbd}, \mathbf{ad} +_{\mathbf{R}} \mathbf{bc}, \mathbf{n})$ but in general there is no triple corresponding to $(\mathbf{a}, \mathbf{b}, \mathbf{n}) \times_{\mathbf{R}} (\mathbf{c}, \mathbf{d}, \mathbf{m})$ if $\mathbf{n}$ and $\mathbf{m}$ represent different numbers. Similar problems occur if non-square roots should be considered. Although the shape of real numbers allowed appears restricted at first sight, it suffices to prove the key system of [25] (see Example 10).

3 Experimental Evaluation

In the experiments[2] we considered the 470 problems from the quantifier-free non-linear integer arithmetic benchmarks (QF_NIA) of SMT-LIB 2009[3] and the 1391 TRSs in the termination problems database (TPDB) version 5.0 (available via `http://termination-portal.org/wiki/TPDB`). All tests have been performed on a server equipped with 8 dual-core AMD Opteron® processors 885 running at a clock rate of 2.6 GHz and 64 GB of main memory. Unless stated otherwise only a single core of the server was used.

We implemented the approach presented in Appendix A and Section 2 and used MiniSat [11] as back-end (after a satisfiability preserving transformation to CNF [28]). The result, called MiniSmt, accepts the SMT-LIB syntax for quantifier-free non-linear arithmetic. For a comparison of MiniSmt with other SMT solvers see Section 3.1. We also integrated matrix interpretations as presented in Appendix B in the termination prover T$_{\text{T}}$T$_2$ [22] based on the constraint language of Definition 1. The constraints within T$_{\text{T}}$T$_2$ are solved with an interfaced version of MiniSmt. Experiments are discussed in Section 3.2.

3.1 Comparison with SMT Solvers

First we compare MiniSmt with other recent SMT solvers. Since 2009 the QF_NIA category is part of SMT-COMP in which Barcelogic [27] and CVC3 [5] participated. The results when comparing these tools on the QF_NIA benchmarks of SMT-LIB are given in Table 1. The column labeled yes (no) counts how many systems could be proved (un)satisfiable while time indicates the total time needed by the tool in seconds. A "–" indicates that the solver does not support the corresponding setting. If no answer was produced within 60 seconds the execution is killed (column t/o). The row labeled $\sum$ shows the accumulative

[2] See `http://cl-informatik.uibk.ac.at/ttt2/arithmetic` for full details.
[3] See `http://www.smtcomp.org/2009` for information on SMT-COMP and SMT-LIB.

Table 1. SMT solvers on 470 problems from SMT-LIB

	yes	no	time	t/o
Barcelogic	266	189	1188	15
CVC3	113	139	13169	218
MiniSmt(sat)	267	–	6427	54
MiniSmt(bv)	268	–	3190	42
$\sum$	269	194		

Table 2. Statistics on various benchmarks

	#	#var		#add		#mul	
		avg	max	avg	max	avg	max
SMT-LIB (calypto)	303	10	50	6	36	6	33
SMT-LIB (leipzig)	167	301	2606	113	1136	164	1420
SMT-LIB (calypto + leipzig)	470	113	2606	44	1136	62	1420
matrices (dimension 1)	1391	25	811	145	5824	226	10688
matrices (dimension 2)	1391	78	2726	1420	164276	1863	164452

yes (no) score for the corresponding column. In Table 1 MiniSmt makes use of the multi-core architecture of the server and searches for satisfying assignments based on two different settings. Instances where small domains suffice are handled by the configuration which uses 3 bits for arithmetic variables and 4 bits for intermediate results. The second setting employs 33 and 50 bits, respectively. As an alternative to the SAT back-end (denoted MiniSmt(sat)) in MiniSmt we also developed a transformation that allows to use SMT solvers for bit-vector logic to solve arithmetic constraints (called MiniSmt(bv)). Although bit-vector arithmetic cannot be used blindly for our setting—it does not take overflows into account and hence can produce unsound results—it can be adapted for solving non-linear arithmetic by sign-extension operations. Given the details in Appendix A this transformation is straightforward to implement. As a back-end for MiniSmt(bv) we use Yices [10] as designated SMT solver for bit-vector logic. As can be inferred from Table 1 even when dealing with large numbers MiniSmt performs competitively, i.e., it solves the most satisfiable instances. MiniSmt(bv) finds models for the problems calypto/problem-006547.cvc.1, leipzig/term-gZE9f0, and leipzig/term-lFYv5w while Barcelogic finds a model for leipzig/term-BKc7xf which MiniSmt(bv) misses. MiniSmt(sat) cannot handle leipzig/term-XbWQfu in contrast to its bit-vector pendant.

Since the SMT-LIB benchmarks consider only the integers as domain we also generated (with $\mathsf{T_TT_2}$) typical constraints from termination analysis. More precisely we generated for every TRS a constraint that is satisfiable if and only if a direct proof with matrix interpretations over a non-negative carrier of a fixed dimension removes at least one rewrite rule.[4] This constraint is then solved

[4] Our benchmarks are available from the URL in Footnote 2.

Table 3. SMT solvers on 1391 matrix constraints (dimension 1)

	$\mathbb{N}$				$\mathbb{Z}$				$\mathbb{Q}$			
	yes	no	time	t/o	yes	no	time	t/o	yes	no	time	t/o
Barcelogic	335	3	23k	172	414	3	16k	95	–	–	–	–
CVC3	168	415	45k	748	197	380	45k	754	120	409	48k	802
MiniSmt(sat)	337	–	1701	5	539	–	7279	40	337	–	1592	3
MiniSmt(bv)	337	–	902	5	553	–	1387	8	337	–	1258	6
nlsol	332	–	3203	14	479	–	8158	83	333	–	3924	20
$\sum$	338	415			553	380			338	409		

Table 4. SMT solvers on 1391 matrix constraints (dimension 2)

	$\mathbb{N}$				$\mathbb{Z}$				$\mathbb{Q}$			
	yes	no	time	t/o	yes	no	time	t/o	yes	no	time	t/o
Barcelogic	408	3	41k	578	832	3	17k	204	–	–	–	–
CVC3	117	130	66k	1084	112	84	68k	1135	58	125	69k	1147
MiniSmt(sat)	402	–	7193	63	995	–	23k	214	407	–	6505	60
MiniSmt(bv)	405	–	5248	57	1068	–	13k	140	400	–	6418	69
nlsol	301	–	18k	190	454	–	51k	771	289	–	20k	240
$\sum$	412	130			1142	84			409	125		

over various domains, to allow a comprehensive comparison with nlsol [6], a recent solver for polynomial arithmetic, which follows a similar approach as Barcelogic but in addition handles non-integral domains. Our benchmarks are in the QF_NIA syntax but are of a different structure than the SMT-LIB instances. Specifically, our problems admit more arithmetic operations while typically having less variables. In Table 2 some statistics and comparisons regarding the different test benches are given. There the column # indicates the number of systems in the respective benchmark family and the other columns give accumulated information on the size (i.e., number of variables, additions, and multiplications) of the problems. Since nlsol requires a slightly different input format our benchmarks are preprocessed for this tool.

Table 3 presents the results for the matrix benchmarks of dimension one. Times postfixed with "k" should be multiplied by a factor of 1000. For the solvers nlsol and MiniSmt variables range over the domain $\{0,\ldots,15\}$ ($\mathbb{N}$), $\{-16,\ldots,15\}$ ($\mathbb{Z}$), and $\{\frac{0}{2},\frac{1}{2},\frac{2}{2},\ldots,\frac{15}{2}\}$ ($\mathbb{Q}$). Table 4 considers the benchmarks with matrices of dimension two. Since these constraints are much larger, MiniSmt and nlsol use one bit less for representing numbers. We also considered different domains which produced similar results. In Tables 3 and 4 only the benchmarks considering the domains $\mathbb{N}$ and $\mathbb{Q}$ correspond to valid (parts of) termination proofs. The reason for including the $\mathbb{Z}$ benchmarks in the tables is that they allow the bit-vector back-end of MiniSmt to show its performance best.

We note that nlsol allows more flexibility in choosing the variable domain since MiniSmt bounds variables by powers of two. However our approach admits more freedom in bounding intermediate results which reduces the search space

Table 5. Matrices with dependency pairs for 1391 TRSs

	1×1			2×2			3×3			
	yes	time	t/o	yes	time	t/o	yes	time	t/o	$\sum$
$\mathbb{N}$	545	8885	83	618	23820	326	627	25055	349	659
$\mathbb{Q}$	599	8574	67	597	20238	261	496	19490	252	638
$\mathbb{Q}_1$	606	5906	46	655	15279	173	643	14062	164	685
$\mathbb{Q}_2$	627	10109	93	651	23102	308	619	23806	330	687
$\mathbb{R}$	535	17029	198	630	16517	200	599	29346	415	648
$\sum$	639			674			664			703

(e.g. for columns $\mathbb{Q}$ in Tables 3 and 4 we require that intermediate results are integers) which results in efficiency gains. For the sake of a fair comparison we configured our tool such that arithmetic variables are represented in as many bits as intermediate results. However, usually it is a good idea to allow more bits for intermediate results.

We summarize the tables with the following observations: (a) MiniSmt is surprisingly powerful on the SMT-LIB benchmarks (containing few multiplications but large numbers, i.e., requiring more than 30 bits), (b) our tool performs best (note its speed) on the matrix benchmarks (containing many multiplications and usually small domains suffice), (c) MiniSmt is by far the most powerful tool on rational domains, (d) while in general the SMT back-end of MiniSmt is favorable, for column $\mathbb{Q}$ in Table 4 the SAT back-end shows more problems satisfiable than the SMT counterpart, and (e) CVC3 could be used to cancel termination proof attempts early due to its power concerning unsatisfiability.

Our tool fills two gaps that current SMT solvers admit. It is fastest on small and rational domains and to our knowledge the only solver that efficiently supports irrational domains, e.g., only our tool can solve the constraint $2 = x \times x$ for a real-valued variable x. Due to a lack of interesting benchmarks and competitor tools, MiniSmt cannot show its full strength here.

3.2 Evaluation within a Termination Prover

Next we compare matrix interpretations over $\mathbb{N}$, $\mathbb{Q}$, and $\mathbb{R}$ (cf. Appendix B) and show that MiniSmt admits a fast automation of the method. The coefficients of a matrix over dimension d are represented in $\max\{2, 5-d\}$ bits (for reals we allow $\max\{1, 3-d\}$ bits due to the more expensive pair representation). Every rational coefficient is represented as a fraction with denominator two. Hence a matrix of dimension two admits natural coefficients $\{0, 1, \ldots, 7\}$, rational coefficients $\{0, \frac{1}{2}, 1, 1\frac{1}{2}, 2, 2\frac{1}{2}, 3, 3\frac{1}{2}\}$, and real coefficients $\{0, 1, \sqrt{2}, 1+\sqrt{2}\}$. The number of bits for representing intermediate computations was chosen to be one more than the number of bits allowed for the coefficients. Restricting the bit-width is essential for performance, especially for larger dimensions. It is well-known that for interpretation based termination criteria usually small coefficients suffice.

In Table 5 matrices of dimensions one to three are considered. The rows labeled $\mathbb{N}$ indicate that only natural numbers are allowed as coefficients whereas $\mathbb{Q}$ refers to the naive representation of rationals without canceling the fractions

and $\mathbb{R}$ to the subset of real coefficients mentioned above. The rows $\mathbb{Q}_n$ indicate that a fraction is canceled if its denominator exceeds n. The column labeled yes shows the number of successful termination proofs while time indicates the total time needed by the tool in seconds. If no answer was produced within 60 seconds the execution is killed (column t/o). The row (column) labeled $\sum$ shows the accumulative yes score for the corresponding column (row).

Matrix interpretations over $\mathbb{N}$ are used by most contemporary termination tools and serve as a reference. The performance of $\mathbb{Q}$ is satisfactory for matrices with dimension one (which correspond to linear polynomial interpretations and confirms the results in [15]) but poor for larger dimensions. In contrast, the overall performance of $\mathbb{Q}_1$ is excellent, i.e., it is much faster than $\mathbb{N}$ and more powerful. The combination of all 15 methods from Table 5 together can prove 703 systems terminating, yielding a gain of almost 50 systems compared to the standard setting allowing natural coefficients only. This number is remarkable since Jambox [12]—a powerful termination prover based on various termination criteria—proved 750 systems terminating in the 2008 competition and took 3rd place. Since the competition execution software allows to run 16 processes in parallel the (single!) method we propose is a good starting point for new termination analyzers. Looking beyond TPDB, our implementation also shows its strength for real coefficients. It masters the TRS $\mathcal{R}_{\mathbb{R}}$ from Example 10 below. This system stems from [25] where it was proved that no direct termination proof based on polynomial interpretations over the natural or rational numbers can exist which orients all rules strictly. However a proof over the reals is possible and our implementation finds such a proof fully automatically. Due to the statement "... only the techniques [...] which concern *non-negative rational numbers* have been included in Mu-term ..." in [26], we believe that T$_{\text{T}}$T$_2$ is the only automatic termination analyzer for TRSs that supports reasoning about irrational domains.

Example 10. For the TRS $\mathcal{R}_{\mathbb{R}}$ from [25] consisting of the seven rules

$$\begin{aligned}
\mathsf{k}(x,x,\mathsf{b}_1) &\to \mathsf{k}(\mathsf{g}(x),\mathsf{b}_2,\mathsf{b}_2) & \mathsf{g}(\mathsf{c}(x)) &\to \mathsf{f}(\mathsf{c}(\mathsf{f}(x)))\\
\mathsf{k}(x,\mathsf{a}_2,\mathsf{b}_1) &\to \mathsf{k}(\mathsf{a}_1,x,\mathsf{b}_1) & \mathsf{f}(\mathsf{f}(x)) &\to \mathsf{g}(x)\\
\mathsf{k}(\mathsf{a}_4,x,\mathsf{b}_1) &\to \mathsf{k}(x,\mathsf{a}_3,\mathsf{b}_1) & \mathsf{f}(\mathsf{f}(\mathsf{f}(\mathsf{f}(x)))) &\to \mathsf{k}(x,x,x)\\
\mathsf{k}(\mathsf{g}(x),\mathsf{b}_3,\mathsf{b}_3) &\to \mathsf{k}(x,x,\mathsf{b}_4)
\end{aligned}$$

T$_{\text{T}}$T$_2$ finds the following interpretation that orients all rules strictly

$$\begin{aligned}
\mathsf{a}_{1\mathbb{R}} &= 0 & \mathsf{b}_{1\mathbb{R}} &= 2+\sqrt{2} & \mathsf{f}_{\mathbb{R}}(x) &= \sqrt{2}x+\sqrt{2}\\
\mathsf{a}_{2\mathbb{R}} &= 1+2\sqrt{2} & \mathsf{b}_{2\mathbb{R}} &= 0 & \mathsf{g}_{\mathbb{R}}(x) &= 2x+1+\sqrt{2}\\
\mathsf{a}_{3\mathbb{R}} &= 0 & \mathsf{b}_{3\mathbb{R}} &= 1+\sqrt{2} & \mathsf{c}_{\mathbb{R}}(x) &= x+1+2\sqrt{2}\\
\mathsf{a}_{4\mathbb{R}} &= 1+\sqrt{2} & \mathsf{b}_{4\mathbb{R}} &= \sqrt{2} & \mathsf{k}_{\mathbb{R}}(x,y,z) &= x+y+\sqrt{2}z+3\sqrt{2}
\end{aligned}$$

within a fraction of a second. While a direct proof with polynomials over $\mathbb{N}$ is not possible, natural coefficients suffice in the dependency pair setting (after computing the SCCs of the dependency graph). Hence all modern termination tools can prove this system terminating.

4 Related Work and Concluding Remarks

First we discuss related approaches for solving non-linear arithmetic. Barcelogic follows [6] where constraints are linearized by assigning a finite domain to variables occurring in non-linear constraints. The resulting linear arithmetic formula is solved by a variant of the simplex algorithm. In contrast to the work in [6] which mentions heuristics (decide which variables should be used for the linearization) our approach does not require such considerations. However both approaches require some fixed domain for the variables. CVC3 implements a Fourier-Motzkin procedure for linear arithmetic while treating non-linear terms as if they were linear (Dejan Jovanović, personal conversation, 2009). The last tool we considered for comparison, nlsol, uses a similar approach as Barcelogic but also supports non-integral domains. It transforms non-linear constraints to linear arithmetic before it calls Yices as back-end. The difference to MiniSmt(bv) is that nlsol employs Yices for solving linear arithmetic whereas our tool uses it for bit-vector arithmetic. We are aware of the fact that the first order theory of real arithmetic is decidable [29] but because of the underlying computational complexity of the method the result is mainly of theoretical interest. Improvements of the original procedure are still of double exponential time complexity [8]. Nevertheless it might be interesting to investigate how this worst case complexity affects the performance for applications. Note that SAT solving techniques and the simplex method admit exponential time worst case complexity but are surprisingly efficient in practice.

Next we discuss related work on matrix interpretations. An extension to rational domains was already proposed in 2007 [16] (for termination proofs of string rewrite systems) where evolutionary algorithms [3] were suggested to find suitable rational coefficients. However, no benchmarks are given there that show a gain in power. In [15] polynomial interpretations are extended to rational coefficients. This work is related since linear polynomial interpretations coincide with matrix interpretations of dimension one. Our experiments confirm the gains in power when using matrices of dimension one but the method from [15] results in a poor performance for larger dimensions without further ado. Independently to our research, [1] extends the theory of matrix interpretations to coefficients over the reals. However their (preliminary) implementation can only deal with rationals. Furthermore no benchmarks are given in [1] showing any gains in power by allowing rationals. Hence our contribution for the first time gives evidence that matrix interpretations over the non-negative reals do really extend the power of termination criteria in practice. Recently non-linear matrix interpretations have been introduced [9] by considering matrix domains instead of vector domains. We would like to investigate if this more general setting can also benefit from rational domains.

Before we conclude this section with ideas for future work we mention one specialty of MiniSmt. Due to the bottom-up representation of domains (naturals, integers, rationals, reals) our solver can be used for instances that require arithmetic variables of different types. This distinguishes MiniSmt from the other solvers that do currently not support such problems appropriately. In the future

we would like to add support for reasoning about unsatisfiable instances. As an immediate consequence this would improve the cumulative execution time of MiniSmt and as a side effect this would also be beneficial for termination analysis of rewriting; a termination proof can be aborted immediately if the corresponding constraints are unsatisfiable and a different termination criterion can be considered. Another extension aims at improving the handling of real domains. Instead of restricting to approximations of $\sqrt{2}$ one could consider $\sqrt{\mathbf{n}}$ where $\mathbf{n}$ is some abstract expression representing a non-negative integer (as discussed at the end of Section 2.2). Moreover, allowing $\mathbf{n}$ to be negative admits reasoning about complex domains. However, we are not aware of any termination criteria that require such a domain.

Acknowledgments. We thank Nikolaj Bjørner for encouraging us to investigate the bit-vector back-end, René Thiemann for pointing out a bug, and the anonymous referees for numerous suggestions that helped to improve the presentation.

References

1. Alarcón, B., Lucas, S., Navarro-Marset, R.: Proving termination with matrix interpretations over the reals. In: WST 2009, pp. 12–15 (2009)
2. Arts, T., Giesl, J.: Termination of term rewriting using dependency pairs. TCS 236(1-2), 133–178 (2000)
3. Ashlock, D.: Evolutionary Computation for Modeling and Optimization. Springer, Heidelberg (2006)
4. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press, Cambridge (1998)
5. Barrett, C., Tinelli, C.: CVC3. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 298–302. Springer, Heidelberg (2007)
6. Borralleras, C., Lucas, S., Navarro-Marset, R., Rodriguez-Carbonell, E., Rubio, A.: Solving non-linear polynomial arithmetic via SAT modulo linear arithmetic. In: Schmidt, R.A. (ed.) Automated Deduction – CADE-22. LNCS(LNAI), vol. 5663, pp. 294–305. Springer, Heidelberg (2009)
7. Codish, M., Lagoon, V., Stuckey, P.: Solving partial order constraints for LPO termination. JSAT 5, 193–215 (2008)
8. Collins, G.E.: Quantifier elimination for real closed fields by cylindrical algebraic decomposition. In: Brakhage, H. (ed.) GI-Fachtagung 1975. LNCS, vol. 33, pp. 134–183. Springer, Heidelberg (1975)
9. Courtieu, P., Gbedo, G., Pons, O.: Improved matrix interpretation. In: SOFSEM 2010. LNCS, vol. 5901, pp. 283–295. Springer, Heidelberg (2010)
10. Dutertre, B., de Moura, L.: A fast linear-arithmetic solver for DPLL(T). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer, Heidelberg (2006)
11. Eén, N., Sörensson, N.: An extensible SAT-solver. In: SAT 2004. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)
12. Endrullis, J.: (Jambox), http://joerg.endrullis.de
13. Endrullis, J., Waldmann, J., Zantema, H.: Matrix interpretations for proving termination of term rewriting. JAR 40(2-3), 195–220 (2008)
14. 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)

15. Fuhs, C., Navarro-Marset, R., Otto, C., Giesl, J., Lucas, S., Schneider-Kamp, P.: Search techniques for rational polynomial orders. 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. 109–124. Springer, Heidelberg (2008)
16. Gebhardt, A., Hofbauer, D., Waldmann, J.: Matrix evolutions. In: WST 2007, pp. 4–8 (2007)
17. Giesl, J., Thiemann, R., Schneider-Kamp, P., Falke, S.: Mechanizing and improving dependency pairs. JAR 37(3), 155–203 (2006)
18. 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)
19. Hirokawa, N., Middeldorp, A.: Automating the dependency pair method. I&C 199(1-2), 172–199 (2005)
20. Hofbauer, D., Waldmann, J.: Termination of string rewriting with matrix interpretations. In: Pfenning, F. (ed.) RTA 2006. LNCS, vol. 4098, pp. 328–342. Springer, Heidelberg (2006)
21. Hofbauer, D.: Termination proofs by context-dependent interpretations. In: Middeldorp, A. (ed.) RTA 2001. LNCS, vol. 2051, pp. 108–121. Springer, Heidelberg (2001)
22. 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)
23. Kroening, D., Strichman, O.: Decision Procedures. Springer, Heidelberg (2008)
24. Lucas, S.: Polynomials over the reals in proofs of termination: From theory to practice. TIA 39(3), 547–586 (2005)
25. Lucas, S.: On the relative power of polynomials with real, rational, and integer coefficients in proofs of termination of rewriting. AAECC 17(1), 49–73 (2006)
26. Lucas, S.: Practical use of polynomials over the reals in proofs of termination. In: PPDP 2007, pp. 39–50 (2007)
27. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT modulo theories: from an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T). J. ACM 53(6), 937–977 (2006)
28. Plaisted, D., Greenbaum, S.: A structure-preserving clause form translation. JSC 2(3), 293–304 (1986)
29. Tarski, A.: A Decision Method for Elementary Algebra and Geometry, 2nd edn. University of California Press, Berkeley (1957)
30. Zankl, H.: Lazy Termination Analysis. PhD thesis, University of Innsbruck (2009)
31. Zankl, H., Hirokawa, N., Middeldorp, A.: KBO orientability. JAR 43(2), 173–201 (2009)
32. Zantema, H.: Termination. In: TeReSe (ed.) Term Rewriting Systems, pp. 181–259. Cambridge University Press, Cambridge (2003)

A Encoding Non-linear Integral Arithmetic in SAT

In this section arithmetic constraints (cf. the grammar in Definition 1) are reduced to SAT. To obtain formulas of finite size, every arithmetic variable is represented by a given number of bits. Then operations such as $+_{\mathbf{N}}$ and $\times_{\mathbf{N}}$ are unfolded according to their definitions using circuits. Such definitions for

$+_{\mathbf{N}}$ and $\times_{\mathbf{N}}$ have already been presented in [13] for bit-vectors of a fixed width. In contrast, we take overflows into account. The encodings of $>_{\mathbf{N}}$ and $=_{\mathbf{N}}$ for bit-vectors given below are similar to the ones in [7].

A.1 Arithmetic over $\mathbb{N}$

We fix the number k of bits that is available for representing natural numbers in binary. Let $a < 2^k$. We denote by $\mathbf{a}_k = \langle a_k, \ldots, a_1 \rangle$ the binary representation of a where a_k is the most significant bit. Hence e.g. $\langle \top, \top, \bot \rangle = 6$. Whenever k is not essential we abbreviate $\mathbf{a}_k$ to $\mathbf{a}$. Furthermore the operation $(\cdot)_k$ on bit-vectors is used to drop bits, i.e., $\langle a_4, a_3, a_2, a_1 \rangle_2 = \langle a_2, a_1 \rangle$.

Definition 11. *For natural numbers in binary representation we define:*

$$\mathbf{a}_k >_{\mathbf{N}} \mathbf{b}_k \;:=\; \begin{cases} \bot & \text{if } k = 0 \\ (a_k \wedge \neg b_k) \vee \big((b_k \to a_k) \wedge \mathbf{a}_{k-1} >_{\mathbf{N}} \mathbf{b}_{k-1}\big) & \text{if } k > 0 \end{cases}$$

$$\mathbf{a}_k =_{\mathbf{N}} \mathbf{b}_k \;:=\; \bigwedge_{i=1}^{k} (a_i \leftrightarrow b_i)$$

Since two k-bit bit-vectors sum up to a $(k+1)$-bit number an additional bit is needed for the result. Hence the case arises when two summands are not of equal bit-width. Thus, before adding $\mathbf{a}_k$ and $\mathbf{b}_{k'}$ the shorter one is padded with $|k - k'|$ $\bot$'s. To keep the presentation simple we assume that $\bot$-padding is implicitly performed before the operations $+_{\mathbf{N}}$, $>_{\mathbf{N}}$, and $=_{\mathbf{N}}$.

Definition 12. *We define* $\mathbf{a}_k +_{\mathbf{N}} \mathbf{b}_k$ *as* $\langle c_k, s_k, \ldots, s_1 \rangle$ *for* $1 \leqslant i \leqslant k$ *with*

$$c_0 = \bot \qquad s_i = a_i \otimes b_i \otimes c_{i-1} \qquad c_i = (a_i \wedge b_i) \vee (a_i \wedge c_{i-1}) \vee (b_i \wedge c_{i-1})$$

were $\otimes$ *denotes exclusive or, i.e.,* $x \otimes y \;:=\; \neg(x \leftrightarrow y)$.

Note that in practice it is essential to introduce new variables for the carry and the sum since in consecutive additions each bit a_i and b_i is duplicated (twice for the carry and once for the sum). Using fresh variables for the sum prevents an exponential blowup of the resulting formula. A further method to keep formulas small is to limit the bit-width when representing naturals. This can be accomplished after addition (or multiplication) by fixing a maximal number m of bits. To restrict $\mathbf{a}_k$ to m bits we demand that all a_i for $m+1 \leqslant i \leqslant k$ are $\bot$ as a side constraint. Then it is sound (i.e., restricting bits can result in unsatisfiable formulas but never produce models for unsatisfiable input) to continue any computations with $\mathbf{a}_m$ instead of $\mathbf{a}_k$.

The next example demonstrates addition. To ease readability we only use $\bot$ and $\top$ in the following examples and immediately simplify formulas.

Example 13. We compute $3 +_{\mathbb{N}} 14 = 17$. In the sequence below the first step performs $\bot$-padding. Afterwards Definition 12 applies.

$$\langle \top, \top \rangle +_{\mathbf{N}} \langle \top, \top, \top, \bot \rangle = \langle \bot, \bot, \top, \top \rangle +_{\mathbf{N}} \langle \top, \top, \top, \bot \rangle = \langle \top, \bot, \bot, \bot, \top \rangle$$

Multiplication is implemented by addition and bit-shifting. Here $\mathbf{a} \ll n$ denotes a left-shift of $\mathbf{a}$ by n bits, e.g., $\langle x, y\rangle \ll 3$ yields $\langle x, y, \bot, \bot, \bot\rangle$. The operation $(\cdot)$ takes a bit-vector and a Boolean variable and performs scalar multiplication, i.e., $\mathbf{a}_k \cdot x = \langle a_k \wedge x, \ldots, a_1 \wedge x\rangle$. In the sequel the operator $(\cdot)$ binds stronger than $\ll$, i.e., $\mathbf{a} \cdot x \ll 2$ abbreviates $(\mathbf{a} \cdot x) \ll 2$.

The product of two bit-vectors with m and n bits has $m + n$ bits.

Definition 14. *For bit-vectors* $\mathbf{a}_m$ *and* $\mathbf{b}_n$ *we define:*

$$\mathbf{a}_m \times_{\mathbf{N}} \mathbf{b}_n \;:=\; \big((\mathbf{a}_m \cdot b_1 \ll 0) +_{\mathbf{N}} \cdots +_{\mathbf{N}} (\mathbf{a}_m \cdot b_n \ll (n-1))\big)_{m+n}$$

In the following example we demonstrate multiplication.

Example 15. Let $\mathbf{a} = \langle \top, \bot, \top\rangle$ and $\mathbf{b} = \langle \top, \top, \bot\rangle$, i.e., we compute $5 \times_{\mathbb{N}} 6 = 30$. The first step below unfolds Definition 14. Then the scalar multiplications are evaluated before shifting is performed. After addition (using $+_{\mathbf{N}}$) the sum is restricted to six bits.

$$\begin{aligned}
\mathbf{a} \times_{\mathbf{N}} \mathbf{b} &= \big((\mathbf{a} \cdot \bot \ll 0) +_{\mathbf{N}} (\mathbf{a} \cdot \top \ll 1) +_{\mathbf{N}} (\mathbf{a} \cdot \top \ll 2)\big)_6 \\
&= \big((\langle \bot, \bot, \bot\rangle \ll 0) +_{\mathbf{N}} (\mathbf{a} \ll 1) +_{\mathbf{N}} (\mathbf{a} \ll 2)\big)_6 \\
&= \big(\langle \bot, \bot, \bot\rangle +_{\mathbf{N}} \langle \top, \bot, \top, \bot\rangle +_{\mathbf{N}} \langle \top, \bot, \top, \bot, \bot\rangle\big)_6 = \langle \bot, \top, \top, \top, \top, \bot\rangle
\end{aligned}$$

If-then-else is an abbreviation, i.e., $x \,?\, \mathbf{a} : \mathbf{b} \;:=\; (\mathbf{a} \cdot x) +_{\mathbf{N}} (\mathbf{b} \cdot \neg x)$. This expression evaluates to $\mathbf{a}$ if x is true and to $\mathbf{b}$ otherwise. We omit its straightforward redefinitions when considering arithmetic over $\mathbb{Z}$, $\mathbb{Q}$, and $\mathbb{R}$. Note that this operator can encode the maximum of two numbers, i.e., $\max(\mathbf{a}, \mathbf{b}) \;:=\; \mathbf{a} >_{\mathbf{N}} \mathbf{b} \,?\, \mathbf{a} : \mathbf{b}$.

A.2 Arithmetic over $\mathbb{Z}$

We represent integers using two's complement which allows a straightforward encoding of arithmetic operations. For a k-bit number the most significant bit denotes the sign, e.g., $\mathbf{a}_k = \langle a_k, \ldots, a_1\rangle$ with sign a_k and bits $a_{k-1}, \ldots, a_1$. Sign $\top$ indicates negative values. Again some definitions expect operands to be of equal bit-width. This is accomplished by implicitly sign-extending the shorter operand. The operation $(\cdot)_k$ is abused for both sign-extending and discarding bits, e.g., $\langle \bot, \top\rangle_4 = \langle \bot, \bot, \bot, \top\rangle$, $\langle \top, \top\rangle_4 = \langle \top, \top, \top, \top\rangle$, and $\langle \top, \top, \top\rangle_2 = \langle \top, \top\rangle$. The integer represented by the bit-vector does not change when sign-extending. Similar to the case for $\mathbf{N}$, a bit-vector $\mathbf{a}_k$ can be restricted to m bits. If the dropped bits take the same value as the sign, then $\mathbf{a}_k$ and $\mathbf{a}_m$ denote the same number. Adding a side constraint $a_k \leftrightarrow a_i$ for $m \leqslant i \leqslant k$ allows to proceed with $\mathbf{a}_m$ instead of $\mathbf{a}_k$.

Comparisons are defined based on the corresponding operations over $\mathbf{N}$. For $>_{\mathbf{Z}}$ a separate check on the sign is needed, i.e., $\mathbf{a}$ is greater than $\mathbf{b}$ if $\mathbf{b}$ is negative while $\mathbf{a}$ is not and otherwise the bits are compared using $>_{\mathbf{N}}$. For $+_{\mathbf{Z}}$ and $\times_{\mathbf{Z}}$ numbers are first sign-extended before the corresponding operation over $\mathbf{N}$ is employed. Superfluous bits are discarded afterwards.

Definition 16. *The operations* $>_{\mathbf{Z}}$, $=_{\mathbf{Z}}$, $+_{\mathbf{Z}}$, *and* $\times_{\mathbf{Z}}$ *are defined as follows:*

$$\begin{aligned}
\mathbf{a}_k >_{\mathbf{Z}} \mathbf{b}_k &:= \big(\neg a_k \wedge b_k\big) \vee \big((a_k \to b_k) \wedge \mathbf{a}_{k-1} >_{\mathbf{N}} \mathbf{b}_{k-1}\big)\\
\mathbf{a}_k =_{\mathbf{Z}} \mathbf{b}_k &:= \mathbf{a}_k =_{\mathbf{N}} \mathbf{b}_k\\
\mathbf{a}_k +_{\mathbf{Z}} \mathbf{b}_k &:= (\mathbf{a}_{k+1} +_{\mathbf{N}} \mathbf{b}_{k+1})_{k+1}\\
\mathbf{a}_m \times_{\mathbf{Z}} \mathbf{b}_n &:= (\mathbf{a}_{m+n} \times_{\mathbf{N}} \mathbf{b}_{m+n})_{m+n}
\end{aligned}$$

Subtraction is encoded using addition and two's complement, i.e., $\mathbf{a} -_{\mathbf{Z}} \mathbf{b} := \mathbf{a} +_{\mathbf{Z}} \mathrm{tc}_{\mathbf{Z}}(\mathbf{b})$ with $\mathrm{tc}_{\mathbf{Z}}(\cdot)$ as defined below.

Definition 17. *For a bit-vector* $\mathbf{a}_k$ *we define* ones' *and* two's *complement as:*

$$\mathrm{oc}(\mathbf{a}_k) := \langle \neg a_k, \ldots, \neg a_1 \rangle \qquad \mathrm{tc}_{\mathbf{Z}}(\mathbf{a}_k) := \big(\mathrm{oc}(\mathbf{a}_{k+1}) +_{\mathbf{N}} \langle \top \rangle\big)_{k+1}$$

Ones' complement flips all bits and two's complement computes ones' complement incremented by one. To avoid a case distinction on the sign for two's complement the operand first is sign-extended by one auxiliary bit. After computing ones' complement, one is added and then the overflow bit is discarded as shown in the next example.

Example 18. Since -2^k can be represented in k bits but 2^k cannot, $\mathrm{tc}_{\mathbf{Z}}(\mathbf{a}_k)$ must have $k+1$ bits (recall that we take overflows into account). For two's complement of 0 it is essential to first sign-extend the operand and then restrict the result to $k+1$ bits. We demonstrate this with 0 represented by two bits, using an additional bit for the sign:

$$\begin{aligned}
\mathrm{tc}_{\mathbf{Z}}(\langle \bot, \bot, \bot \rangle) &= \big(\mathrm{oc}(\langle \bot, \bot, \bot \rangle_4) +_{\mathbf{N}} \langle \top \rangle\big)_4 = \big(\mathrm{oc}(\langle \bot, \bot, \bot, \bot \rangle) +_{\mathbf{N}} \langle \top \rangle\big)_4\\
&= \big(\langle \top, \top, \top, \top \rangle +_{\mathbf{N}} \langle \top \rangle\big)_4 = \big(\langle \top, \bot, \bot, \bot, \bot \rangle\big)_4 = \langle \bot, \bot, \bot, \bot \rangle
\end{aligned}$$

Next we calculate two's complement of -4 which evaluates to 4:

$$\begin{aligned}
\mathrm{tc}_{\mathbf{Z}}(\langle \top, \bot, \bot \rangle) &= \big(\mathrm{oc}(\langle \top, \bot, \bot \rangle_4) +_{\mathbf{N}} \langle \top \rangle\big)_4 = \big(\mathrm{oc}(\langle \top, \top, \bot, \bot \rangle) +_{\mathbf{N}} \langle \top \rangle\big)_4\\
&= \big(\langle \bot, \bot, \top, \top \rangle +_{\mathbf{N}} \langle \top \rangle\big)_4 = \big(\langle \bot, \bot, \top, \bot, \bot \rangle\big)_4 = \langle \bot, \top, \bot, \bot \rangle
\end{aligned}$$

The next example illustrates addition/subtraction and multiplication.

Example 19. We compute $5 -_{\mathbb{Z}} 2 = 3$. The sequence below translates subtraction ($-_{\mathbf{Z}}$) into addition ($+_{\mathbf{Z}}$) in the first step. Then two's complement of 2 is calculated. Afterwards addition for integers is performed by first sign-extending both operands by one additional bit and then performing addition for naturals ($+_{\mathbf{N}}$). After this step the superfluous carry bit is disregarded, i.e.,

$$\begin{aligned}
\langle \bot, \top, \bot, \top \rangle -_{\mathbf{Z}} \langle \bot, \top, \bot \rangle &= \langle \bot, \top, \bot, \top \rangle +_{\mathbf{Z}} \mathrm{tc}(\langle \bot, \top, \bot \rangle)\\
&= \langle \bot, \top, \bot, \top \rangle +_{\mathbf{Z}} \langle \top, \top, \top, \bot \rangle = \big(\langle \bot, \top, \bot, \top \rangle_5 +_{\mathbf{N}} \langle \top, \top, \top, \bot \rangle_5\big)_5\\
&= \big(\langle \bot, \bot, \top, \bot, \top \rangle +_{\mathbf{N}} \langle \top, \top, \top, \top, \bot \rangle\big)_5 = \big(\langle \top, \bot, \bot, \bot, \top, \top \rangle\big)_5\\
&= \langle \bot, \bot, \bot, \top, \top \rangle.
\end{aligned}$$

Multiplication is similar, i.e., both operands $\mathbf{a}_m$ and $\mathbf{b}_n$ are first sign-extended to have $m+n$ bits. After multiplication ($\times_{\mathbf{N}}$) only the relevant $m+n$ bits are taken. We demonstrate multiplication by computing $-2 \times_{\mathbb{Z}} 5 = -10$:

$$\begin{aligned}\langle \top, \top, \bot \rangle \times_{\mathbf{Z}} \langle \bot, \top, \bot, \top \rangle &= \big(\langle \top, \top, \bot \rangle_7 \times_{\mathbf{N}} \langle \bot, \top, \bot, \top \rangle_7\big)_7 \\ &= \big(\langle \top, \top, \top, \top, \top, \top, \bot \rangle \times_{\mathbf{N}} \langle \bot, \bot, \bot, \bot, \top, \bot, \top \rangle\big)_7 \\ &= \big(\langle \bot, \bot, \bot, \bot, \top, \bot, \bot, \top, \top, \top, \bot, \top, \top, \bot \rangle\big)_7 = \langle \top, \top, \top, \bot, \top, \top, \bot \rangle\end{aligned}$$

B Matrix Interpretations over the Reals

This section unifies two termination criteria for rewrite systems—matrix interpretations [13,20] and polynomial interpretations over the non-negative reals [24,25]—to obtain matrix interpretations over the reals.

B.1 Preliminaries

We assume familiarity with the basics of rewriting [4] and termination [32].

A *signature* $\mathcal{F}$ is a set of function symbols with fixed arities. Let $\mathcal{V}$ denote an infinite set of variables disjoint from $\mathcal{F}$. Then $\mathcal{T}(\mathcal{F}, \mathcal{V})$ forms the set of terms over the signature $\mathcal{F}$ using variables from $\mathcal{V}$. Next we shortly recapitulate the key features of the dependency pair framework [2,17,19]. Let $\mathcal{R}$ be a finite TRS over a signature $\mathcal{F}$. Function symbols that appear as a root of a left-hand side are called *defined*. The signature $\mathcal{F}$ is extended with *dependency pair symbols* $f^\sharp$ for every defined symbol f, where $f^\sharp$ has the same arity as f, resulting in the signature $\mathcal{F}^\sharp$. If $l \to r \in \mathcal{R}$ and t is a subterm of r with a defined root symbol that is not a proper subterm of l then the rule $l^\sharp \to t^\sharp$ is a *dependency pair* of $\mathcal{R}$. Here $l^\sharp$ and $t^\sharp$ are the result of replacing the root symbols in l and t by the corresponding dependency pair symbols. The dependency pairs of $\mathcal{R}$ are denoted by $\mathsf{DP}(\mathcal{R})$.

A *DP problem* $(\mathcal{P}, \mathcal{R})$ is a pair of TRSs $\mathcal{P}$ and $\mathcal{R}$ such that the root symbols of rules in $\mathcal{P}$ do neither occur in $\mathcal{R}$ nor in proper subterms of the left- and right-hand sides of rules in $\mathcal{P}$. The problem is said to be *finite* if there exists no infinite sequence $s_1 \to_{\mathcal{P}} t_1 \to^*_{\mathcal{R}} s_2 \to_{\mathcal{P}} t_2 \to^*_{\mathcal{R}} \cdots$ such that all terms $t_1, t_2, \ldots$ are terminating with respect to $\mathcal{R}$. The main result underlying the dependency pair approach states that termination of a TRS $\mathcal{R}$ is equivalent to finiteness of the DP problem $(\mathsf{DP}(\mathcal{R}), \mathcal{R})$.

To prove a DP problem finite, a number of *DP processors* have been developed. DP processors are functions that take a DP problem $(\mathcal{P}, \mathcal{R})$ as input and return a set of DP problems as output. In order to be employed for proving termination DP processors must be *sound*, i.e., if all DP problems returned by a DP processor are finite then $(\mathcal{P}, \mathcal{R})$ is finite.

Reduction pairs provide a standard approach for obtaining sound DP processors. Formally, a *reduction pair* $(\gtrsim, >)$ consists of a rewrite pre-order $\gtrsim$ (a pre-order on terms that is closed under contexts and substitutions) and a well-founded order $>$ that is closed under substitutions such that the inclusion $> \cdot \gtrsim \subseteq >$ (compatibility) holds. Here $\cdot$ denotes composition of relations.

Theorem 20 (cf. [2,17,19]). *Let $(\gtrsim, >)$ be a reduction pair. The processor that maps a DP problem $(\mathcal{P}, \mathcal{R})$ to $\{(\mathcal{P} \setminus >, \mathcal{R})\}$ if $\mathcal{P} \subseteq \gtrsim \cup >$ and $\mathcal{R} \subseteq \gtrsim$ and to $\{(\mathcal{P}, \mathcal{R})\}$ otherwise is sound.* □

Next we address how to obtain reduction pairs. For a signature $\mathcal{F}$ an $\mathcal{F}$-algebra $\mathcal{A}$ consists of a *carrier* A and an interpretation f_A for every $f \in \mathcal{F}$. If $\mathcal{F}$ is irrelevant or clear from the context we call an $\mathcal{F}$-algebra simply algebra.

Definition 21. *An $\mathcal{F}$-algebra $\mathcal{A}$ over the non-empty carrier A together with two relations $\geqslant$ and $>$ on A is called* weakly monotone *if f_A is monotone in all its coordinates with respect to $\geqslant$, $>$ is well-founded, and $> \cdot \geqslant \subseteq >$.*

Let $\mathcal{A}$ be an algebra over a non-empty carrier A. An assignment α for $\mathcal{A}$ is a mapping from the set of term variables $\mathcal{V}$ to A. Interpretations are lifted from function symbols to terms, using assignments, as usual. The induced mapping is denoted by $[\alpha]_{\mathcal{A}}(\cdot)$. For two terms s and t we define $s >_{\mathcal{A}} t$ if $[\alpha]_{\mathcal{A}}(s) > [\alpha]_{\mathcal{A}}(t)$ holds for all assignments α. The comparison $\geqslant_{\mathcal{A}}$ is similarly defined. Whenever α is irrelevant we abbreviate $[\alpha]_{\mathcal{A}}(s)$ to $[s]_{\mathcal{A}}$.

Weakly monotone algebras give rise to reduction pairs.

Theorem 22. *If $(\mathcal{A}, \geqslant, >)$ is weakly monotone then $(\geqslant_{\mathcal{A}}, >_{\mathcal{A}})$ is a reduction pair.*

Proof. Immediate from [13, Theorem 2, part 2] which is a stronger result. □

B.2 Matrix Interpretations

Next we present a DP processor based on matrix interpretations over the reals. Formally, matrix interpretations are weakly monotone algebras $(\mathcal{M}, \geqslant, >)$ where $\mathcal{M}$ is an algebra over some carrier M^d for a fixed $d \in \mathbb{N}^{>0}$. In the sequel we consider $M = \mathbb{R}^{\geqslant 0}$. To define the relations $\geqslant$ and $>$ that compare elements from M^d, i.e., vectors with non-negative real entries, we must fix how to compare elements from M first. The obvious candidate $>_{\mathbb{R}}$ is not suitable because it is not well-founded. As already suggested in earlier works on polynomial interpretations [21,24,25], $>_{\mathbb{R}}$ can be approximated by $>^{\delta}_{\mathbb{R}}$ defined as $x >^{\delta}_{\mathbb{R}} y \;:=\; x - y \geqslant_{\mathbb{R}} \delta$ for $x, y \in \mathbb{R}$ and any $\delta \in \mathbb{R}^{>0}$. The next lemma shows that $>^{\delta}_{\mathbb{R}}$ has the desired property.

Lemma 23. *The order $>^{\delta}_{\mathbb{R}}$ is well-founded on $\mathbb{R}^{\geqslant 0}$ for any $\delta \in \mathbb{R}^{>0}$.*

Proof. Obvious. □

With the help of $>^{\delta}_{\mathbb{R}}$ it is now possible to define a well-founded order on M^d similar as in [13].

Definition 24. *For vectors $\mathbf{u}$ and $\mathbf{v}$ from M^d we define:*

$$\mathbf{u} \geqslant \mathbf{v} \;:=\; u_i \geqslant_{\mathbb{R}} v_i \text{ for } 1 \leqslant i \leqslant d \qquad \mathbf{u} >^{\delta} \mathbf{v} \;:=\; u_1 >^{\delta}_{\mathbb{R}} v_1 \text{ and } \mathbf{u} \geqslant \mathbf{v}$$

Next the shape of the interpretations is fixed. For an n-ary function symbol $f \in \mathcal{F}^\sharp$ we consider linear interpretations $f_{M^d}(\mathbf{x}_1, \ldots, \mathbf{x}_n) = F_1\mathbf{x}_1 + \cdots + F_n\mathbf{x}_n + \mathbf{f}$ where $F_1, \ldots, F_n \in M^{d\times d}$ and $\mathbf{f} \in M^d$ if $f \in \mathcal{F}$ and $F_1, \ldots, F_n \in M^{1\times d}$ and $\mathbf{f} \in M$ if $f \in \mathcal{F}^\sharp \setminus \mathcal{F}$. (As discussed in [13], using matrices of a different shape for dependency pair symbols reduces the search space while preserving the power of the method.) Before addressing how to compare terms with respect to some interpretation we fix the comparison of matrices. Let $m, n \in \mathbb{N}$. For $B, C \in M^{m\times n}$ we define:

$$B \geqslant C \;:=\; B_{ij} \geqslant_{\mathbb{R}} C_{ij} \text{ for all } 1 \leqslant i \leqslant m,\ 1 \leqslant j \leqslant n$$

Because of the linear shape of the interpretations, for a rewrite rule $l \to r$ with variables $x_1, \ldots, x_k$, matrices $L_1, \ldots, L_k$, $R_1, \ldots, R_k$ and vectors $\mathbf{l}$ and $\mathbf{r}$ can be computed such that

$$[\alpha]_{\mathcal{M}}(l) = L_1\mathbf{x}_1 + \cdots + L_k\mathbf{x}_k + \mathbf{l} \tag{4}$$
$$[\alpha]_{\mathcal{M}}(r) = R_1\mathbf{x}_1 + \cdots + R_k\mathbf{x}_k + \mathbf{r} \tag{5}$$

where $\alpha(x) = \mathbf{x}$ for $x \in \mathcal{V}$. The next lemma states how to test $s >^\delta_{\mathcal{M}} t$ (i.e., $[\alpha]_{\mathcal{M}}(s) >^\delta [\alpha]_{\mathcal{M}}(t)$ for all assignments α) and $s \geqslant_{\mathcal{M}} t$ effectively.

Lemma 25. *Let $l \to r$ be a rewrite rule with $[\alpha]_{\mathcal{M}}(l)$ and $[\alpha]_{\mathcal{M}}(r)$ as in (4) and (5), respectively. Then for any $\delta \in \mathbb{R}^{>0}$*

- *$l \geqslant_{\mathcal{M}} r$ if and only if $L_i \geqslant R_i$ $(1 \leqslant i \leqslant k)$ and $\mathbf{l} \geqslant \mathbf{r}$,*
- *$l >^\delta_{\mathcal{M}} r$ if and only if $L_i \geqslant R_i$ $(1 \leqslant i \leqslant k)$ and $\mathbf{l} >^\delta \mathbf{r}$.*

Proof. Immediate from the proof of [13, Lemma 4]. □

Matrix interpretations over the reals yield weakly monotone algebras.

Theorem 26. *Let $\mathcal{F}$ be a signature, $M = \mathbb{R}^{\geqslant 0}$, and $\mathcal{M}$ an $\mathcal{F}$-algebra over the carrier M^d for some $d \in \mathbb{N}^{>0}$ with f_{M^d} of the shape described above for all $f \in \mathcal{F}$. Then for any $\delta \in \mathbb{R}^{>0}$ the algebra $(\mathcal{M}, \geqslant, >^\delta)$ is weakly monotone.*

Proof. The interpretation functions are monotone with respect to $\geqslant$ because of the non-negative carrier. From Definition 24 it is obvious that $>^\delta$ is well-founded (on the carrier M^d) since $>^\delta_{\mathbb{R}}$ is well-founded on $\mathbb{R}^{\geqslant 0}$ for any $\delta \in \mathbb{R}^{>0}$. The latter holds by Lemma 23. The last condition for a weakly monotone algebra is compatibility, i.e., $>^\delta \cdot \geqslant \subseteq >^\delta$, which trivially holds. □

Matrix interpretations yield reduction pairs due to Theorems 26 and 22, making them suitable for termination proofs in the dependency pair setting.

Corollary 27. *If $(\mathcal{M}, \geqslant, >^\delta)$ is a weakly monotone algebra then $(\geqslant_{\mathcal{M}}, >^\delta_{\mathcal{M}})$ is a reduction pair.* □

We demonstrate matrix interpretations on a simple example.

Example 28. The DP problem $(\{\mathsf{f}^\sharp(\mathsf{s}(x),\mathsf{s}(y)) \to \mathsf{f}^\sharp(x,y)\},\varnothing)$ can be solved by the following interpretation of dimension 2 with $\delta = 1$:

$$\mathsf{f}^\sharp_{M^2}(\mathbf{x},\mathbf{y}) = \begin{pmatrix}1 & 0\end{pmatrix}\mathbf{x} \qquad \mathsf{s}_{M^2}(\mathbf{x}) = \begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix}\mathbf{x} + \begin{pmatrix}\sqrt{2}\\ 0\end{pmatrix}$$

We have $[\mathsf{f}^\sharp(\mathsf{s}(x),\mathsf{s}(y))]_{\mathcal{M}} = \begin{pmatrix}1 & 0\end{pmatrix}\mathbf{x} + \sqrt{2} >^1 \begin{pmatrix}1 & 0\end{pmatrix}\mathbf{x} = [\mathsf{f}^\sharp(x,y)]_{\mathcal{M}}$. By Lemma 25 and Definition 24 we get $\begin{pmatrix}1 & 0\end{pmatrix} \geqslant \begin{pmatrix}1 & 0\end{pmatrix}$ and $\sqrt{2} >^1_{\mathbb{R}} 0$. The latter holds since $\sqrt{2} - 0 \geqslant_{\mathbb{R}} 1$.

Since δ influences if a rule can be oriented strictly or not, it cannot be chosen arbitrarily. E.g., the interpretation from Example 28 with $\delta = 2$ can no longer orient the rule strictly since $\sqrt{2} \not>^2_{\mathbb{R}} 0$. For DP problems containing only finitely many rules (this is the usual setting) a suitable δ can easily be computed. The reason is that for such DP problems only finitely many rules are involved in the strict comparison, i.e., to test for a rule $s \to t$ if $s >^\delta_{\mathcal{M}} t$ the comparison $\mathbf{s} >^\delta \mathbf{t}$ is needed (cf. Lemma 25) which boils down to $s_1 >^\delta_{\mathbb{R}} t_1$ (cf. Definition 24). Since $s_1 - t_1 \geqslant_{\mathbb{R}} \delta$ is tested for only finitely many rules $s \to t$, the minimum of all $s_1 - t_1$ is well-defined and provides a suitable δ. The next lemma (generalizing [24, Section 5.1] to matrices) states that actually there is no need to compute δ explicitly.

Lemma 29. *Let $(\mathcal{P},\mathcal{R})$ be a DP problem. If $\mathcal{P}$ contains finitely many rules then δ need not be computed.*

Proof. The discussion preceding Lemma 29 allows to obtain a $\delta \in \mathbb{R}^{>0}$ such that for every $s \to t \in \mathcal{P}$ we have $s_1 >^\delta_{\mathbb{R}} t_1$ if and only if $s_1 >_{\mathbb{R}} t_1$. Hence for all strict comparisons that occur the relations $>^\delta_{\mathbb{R}}$ and $>_{\mathbb{R}}$ coincide. Consequently it is safe if an implementation uses $>_{\mathbb{R}}$ instead of $>^\delta_{\mathbb{R}}$ in Definition 24 and ignores the exact δ. □

This section is concluded with some comments on automating matrix interpretation, i.e., the problem to find for a given DP problem a matrix interpretation that achieves some progress in the termination proof. Implementing matrix interpretations is a search problem. After fixing the dimension d, for every n-ary function symbol f we obtain matrices $F_1,\ldots,F_n$ and a vector $\mathbf{f}$ filled with arithmetic variables. Lifting addition, multiplication, and comparisons from coefficients to matrices as usual allows to interpret terms. Comparing the term interpretations using Lemma 25 yields an encoding of the DP processor from Theorem 20. For details see [13,30]. From a model returned by the underlying solver the rules which are deleted by the DP processor and the corresponding (part of the) termination proof can be determined. We stress that for matrix interpretations (and many other termination criteria) a plain YES/NO answer from the underlying SMT solver is not sufficient whenever a (modular) proof should be constructed.

Coping with Selfish On-Going Behaviors

Orna Kupferman[1] and Tami Tamir[2]

[1] School of Engineering and Computer Science, Hebrew University, Jerusalem, Israel
[2] School of Computer Science, The Interdisciplinary Center, Herzliya, Israel
orna@cs.huji.ac.il, tami@idc.ac.il

Abstract. A rational and selfish environment may have an incentive to cheat the system it interacts with. Cheating the system amounts to reporting a stream of inputs that is different from the one corresponding to the real behavior of the environment. The system may cope with cheating by charging penalties to cheats it detects. In this paper, we formalize this setting by means of weighted automata and their resilience to selfish environments. Automata have proven to be a successful formalism for modeling the on-going interaction between a system and its environment. In particular, weighted finite automata (WFAs), which assign a cost to each input word, are useful in modeling an interaction that has a quantitative outcome. Consider a WFA $\mathcal{A}$ over the alphabet Σ. At each moment in time, the environment may cheat $\mathcal{A}$ by reporting a letter different from the one it actually generates. A penalty function $\eta : \Sigma \times \Sigma \to \mathbb{R}^{\geq 0}$ maps each possible false-report to a penalty, charged whenever the false-report is detected. A detection-probability function $p : \Sigma \times \Sigma \to [0, 1]$ gives the probability of detecting each false-report. We say that $\mathcal{A}$ is (η, p)-resilient to cheating if $\langle \eta, p \rangle$ ensures that the minimal expected cost of an input word is achieved with no cheating. Thus, a rational environment has no incentive to cheat $\mathcal{A}$.

We study the basic problems arising in the analysis of this setting. In particular, we consider the problem of deciding whether a given WFA $\mathcal{A}$ is (η, p)-resilient with respect to a given penalty function η and a detection-probability function p; and the problem of achieving resilience with minimum resources, namely, given $\mathcal{A}$ and η, finding the minimal (with respect to $\sum_{\sigma,\sigma'} \eta(\sigma, \sigma') \cdot p(\sigma, \sigma')$) detection-probability function p, such that $\mathcal{A}$ is (η, p)-resilient. While for general WFAs both problems are shown to be PSPACE-hard, we present polynomial-time algorithms for deterministic WFAs.

1 Introduction

The environment of modern systems often consists of other systems, having objectives of their own. For example, an e-commerce applications interacts with sellers and buyers. A seller may provide a non-reliable description of the goods he is selling. Furthermore, sellers may provide false feedback and twisted rating of their competitors. Buyers may commit to some transaction but not accomplish it, or may provide a bid that is lower than the real value they are willing to pay, hoping to win even with it. As another example, the environment of

E.M. Clarke and A. Voronkov (Eds.): LPAR-16, LNAI 6355, pp. 501–516, 2010.

various service-providing systems are clients that wish to minimize their payment. Clients' payments may be based on their self-reports, which are usually screened but may be false. In the same way, biased users may affect the quality of recommendation systems for various products or services.

The above examples demonstrate the fact that environments have two types of behaviors: the *truthful* behavior – the one they would produce if they follow their protocol, and the *reported* behavior – the one they actually output, hoping it would lead to a better outcome for them. While the design of systems cannot assume that the environment would take its truthful behavior, we can assume that environments are *rational*, in the sense they always take a behavior that maximizes their outcome.

Mechanism design is a field in game theory and economics studying the design of games for rational players. A game is *incentive compatible* if no player has an incentive to deviate from his truthful behavior [NR99, NRTV07]. The outcome of traditional games depend on the final position of the game. In contrast, the systems we want to reason about maintain an *on-going interaction* with their environment [HP85], and reasoning about their behavior refer not to their final state (in fact, much of the research in the area considers non-terminating systems, with no final state) but rather to the *language* of computations that they generate. In [FKL10], the authors study *rational synthesis*, where the synthesized systems are guaranteed to satisfy their specifications when they interact with rational environments (rather than with hostile environments that do not have objectives other than to fail the system [PR89]). In this paper, we suggest and study a possible model for reasoning about incentive capacity in the context of on-going behaviors and quantitative properties, or formal power series. Reporting of trustworthy information is an essential component also in service-providing systems.

Automata have proven to be a successful formalism for modelling on-going behaviors. Consider a system with a set P of atomic propositions. Each assignment to the atomic propositions corresponds to a letter σ in the alphabet 2^P. Accordingly, a computation of the system, which is a sequence of such assignments, is a word over the alphabet 2^P, and a specification for the system is a language over this alphabet, describing the desired properties of the system. By translating specifications to automata, it is possible to reduce questions about systems and their specifications to questions about automata [VW94]. For example, a system S satisfies a specification ψ if the language that contains exactly all the computations generated by S is contained in the language of an automaton that accepts exactly all words satisfying ψ.

A boolean language maps words to true or false. A *qualitative language* maps words to values from a richer domain [CCH+05, Hen07]. A *Weighted automaton* $\mathcal{A}$ on finite words (WFAs, for short) [Eil74, SS78, Moh97, DKe09] defines a quantitative language $L : \Sigma^* \to \mathbb{R}^{\geq 0} \cup \{\infty\}$. Technically, each transition of $\mathcal{A}$ has a traversal cost, each state has an acceptance cost, and the cost of a run is the sum of the costs of the transitions taken along the run plus the acceptance

cost of its last state. The cost of a word is then the minimum cost over all runs on it (note that the cost may be infinite).

A rational and selfish environment may have an incentive to cheat the WFA and report a word different from the one generated by its truthful behavior. The WFA may cope with cheating by charging penalties to cheats it detects. Formally, at each moment in time, the environment may cheat the WFA by reporting a letter different from the one its truthful behavior generates. A *detection-probability function* $p : \Sigma \times \Sigma \to [0,1]$ gives the probability of detecting each false-report. A *penalty function* $\eta : \Sigma \times \Sigma \to \mathbb{R}^{\geq 0}$ gives the penalty charged whenever a particular false-report is detected. Thus, when the environment reports that a letter σ is σ', then the WFA detects the cheating with probability $p(\sigma, \sigma')$, in which case the environment is charged $\eta(\sigma, \sigma')$. The expected cost of a word w is then the minimum (over all words w' of the same length as w) cost of w' plus the expected cost of reporting w to be w'. We say that a WFA $\mathcal{A}$ is (η, p)-*resilient to cheating* if $\langle \eta, p \rangle$ ensures that, for all words, the above minimal expected cost is achieved in a cheat-free run. Thus, a dominant strategy for the environment is one that does not cheat.

We study the basic problems arising in the analysis of this setting. First, we observe that, by linearity of expectation, a detection probability function p and a penalty function η can be combined to a single *expected-fee* function $\theta = \eta \circ p$; that is, for all $\sigma, \sigma' \in \Sigma$, we have $\theta(\sigma, \sigma') = \eta(\sigma, \sigma') \cdot p(\sigma, \sigma')$. Accordingly, we can study θ-resilience, which simplifies the probabilistic reasoning. Second, we make use of the fact it is possible to construct, given a WFA $\mathcal{A}$ and an expected-fee function θ, a WFA $\mathit{Cheat}(\mathcal{A}, \theta)$ that takes cheating into account and in which the cost of a word is its minimal possible cost (achieved by a best cheating strategy). We show that θ-resilience to cheating is a semantic property. Thus, given a weighted language $L : \Sigma^* \to \mathbb{R}^{\geq 0} \cup \{\infty\}$, and a penalty function θ, then either all WFAs for $\mathcal{A}$ are θ-resilient to cheating, or none of them is. It follows that the natural problem of translating a given WFA $\mathcal{A}$ that need not be θ-resilient to cheating to an equivalent WFA that is θ-resilient to cheating is not interesting, as equivalent WFAs have the same resilience.

With these observations and constructions, we turn to study the practical problems of the setting. From the environment's point of view, we consider the problem of finding, given $\mathcal{A}$, θ, and a word $w \in \Sigma^*$, a word w' such that the environment can minimize the cost of w in $\mathcal{A}$ by reporting it to be w'. We show that the problem can be reduced to the problem of finding a shortest path in a graph, which can be solved in polynomial time [Dij59].

We then turn to study problems from the designer's point of view. We start with the problem of deciding whether a given WFA $\mathcal{A}$ is θ-resilient to cheating with respect to a given expected fee function θ. We show that the problem is PSPACE-hard, but present a polynomial-time solution for the case $\mathcal{A}$ is deterministic. Our solution is based on dynamic programming, taking into account words of increasing lengths. In particular, we show that cycles along which cheating is beneficial (and can therefore lead to an unbounded incentive to cheat) can be detected after quadratically many iterations.

A system with no limits on penalties and with unbounded resources can prevent cheating by fixing a high expected-fee function. In practice, penalties may be limited by an external authority, and increasing the probability of detecting cheats requires resources. Consider a WFA $\mathcal{A}$ and two expected-fee functions θ_1 and θ_2 such that $\theta_1 \leq \theta_2$ (that is $\theta_1(\sigma, \sigma') \leq \theta_2(\sigma, \sigma')$ for all $\sigma, \sigma' \in \Sigma$). If $\mathcal{A}$ is θ_1-resilient to cheating, then $\mathcal{A}$ is clearly also θ_2-resilient to cheating, yet θ_1 achieves resilience more efficiently. In particular, θ_1 can be obtained from θ_2 by reducing the probability of cheat detection, hence saving on resources required for cheat detection. Recall that $\theta = \eta \circ p$, for a penalty function η and a detection probability function p. Assuming that the penalty function η is determined by an external authority, and that system's resources are allocated to increase the detection probability, we consider the following problem of *minimal resources resilience*: Given a WFA $\mathcal{A}$ and a penalty function η, find a probability detection function p such that $\mathcal{A}$ is $(\eta \circ p)$-resilient, and the detection budget, given by $\sum_{\sigma,\sigma'} \eta(\sigma, \sigma')p(\sigma, \sigma')$, is minimal. Note that the probabilities in our objective function are weighted by η. This reflects the fact that detecting a cheat with a high penalty tends to require high resources. Indeed, in practice, the higher is the responsibility of a guard, the higher is his salary. We study the minimal resources resilience problem and show that it is PSPACE-hard. As in resilience testing, the problem is easier in the deterministic case, for which we present a polynomial-time solution, based on describing the problem as a linear program. Essentially, the constraints of the linear program are induced by the restrictions used in the testing algorithm, with the expected-fee values being variables. The same method can be used in order to solve additional minimal-budget problems, with any desired linear objective function over the detection-probability function or the penalty function.

We also consider two variants of the setting. In the *rising-penalty* variant, the expected penalty for cheating increases with the number of cheats. This variant reflects the realistic response of systems to user's false report: allocating more resources to cheat detection, or formally, increasing the detection probability with each detected cheat. In the *bounded cheating* variant the number of times the environment can cheat or the total budget it can invest in penalties is bounded.

2 Preliminaries

In this section we give a formal description of the model we consider, and present several observations and constructions that will be used throughout the paper.

2.1 Weighted Finite Automaton

Given an alphabet Σ, a weighted language is a function $\mathcal{L} : \Sigma^* \to \mathbb{R}^{\geq 0} \cup \{\infty\}$ mapping each word in Σ^* to a positive (possibly ∞) cost. A *weighted finite automaton* (WFA, for short) is $\mathcal{A} = \langle \Sigma, Q, \Delta, c, Q_0, \tau \rangle$, where Σ is a finite input alphabet, Q is a finite set of states, $\Delta \subseteq Q \times \Sigma \times Q$ is a transition relation, $c : \Delta \to \mathbb{R}^{\geq 0}$ is a cost function, $Q_0 \subseteq Q$ is a set of initial states, and $\tau : Q \to$

$\mathbb{R}^{\geq 0} \cup \{\infty\}$ is a final cost function. A transition $d = \langle q, \sigma, p \rangle \in \Delta$ (also written $\Delta(q, \sigma, p)$) can be taken when reading the input letter $\sigma \in \Sigma$, and it causes $\mathcal{A}$ to move from state q to state p with *cost* $c(d)$. The transition relation Δ induces a transition function $\delta : Q \times \Sigma \to 2^Q$, where for a state $q \in Q$ and a letter $\sigma \in \Sigma$, we have $\delta(q, \sigma) := \{p : \Delta(q, \sigma, p)\}$. We extend δ to sets of states, by letting $\delta(S, a) := \bigcup_{q \in S} \delta(q, a)$, and recursively to words in Σ^*, by letting $\delta(q, \varepsilon) = q$, and $\delta(q, u \cdot \sigma) := \delta(\delta(q, u), \sigma)$, for every $u \in \Sigma^*$ and $\sigma \in \Sigma$.

Note that a WFA $\mathcal{A}$ may be nondeterministic in the sense that it may have many initial states, and the transition function may lead to several successor states. If $|Q_0| = 1$ and for every state $q \in Q$ and letter $\sigma \in \Sigma$ we have $|\delta(q, \sigma)| \leq 1$, then $\mathcal{A}$ is a *deterministic* WFA (for short, DWFA).

For a word $w = w_1 \dots w_n \in \Sigma^*$, a run of $\mathcal{A}$ on w is a sequence $r = r_0, r_1, \dots, r_n \in Q^{n+1}$, where $r_0 \in Q_0$ and for every $1 \leq i \leq n$, we have $\Delta(r_{i-1}, w_i, r_i)$. The cost of a run is the sum of the costs of the transitions that constitute the run, along with the final cost. [1] Formally, let $r = r_0, r_1, \dots, r_n$ be a run of $\mathcal{A}$ on w, and let $d = d_1 \dots d_n \in \Delta^*$ be the corresponding sequence of transitions. The cost of r is $cost(\mathcal{A}, r) = \sum_{i=1}^{n} c(d_i) + \tau(r_n)$. For two indices $1 \leq j_1 < j_2 \leq n$, we use $cost(\mathcal{A}, r, j_1, j_2)$ to denote the cost of the sub-run leading from $q_{j_1 - 1}$ to q_{j_2}. Thus, $cost(\mathcal{A}, r, j_1, j_2) = \sum_{i=j_1}^{j_2} c(d_i)$ The cost of w in $\mathcal{A}$, denoted $cost(\mathcal{A}, w)$, is the minimal cost over all runs of $\mathcal{A}$ on w. Thus, $cost(\mathcal{A}, w) = \min\{cost(\mathcal{A}, r) : r \text{ is an accepting run of } \mathcal{A} \text{ on } w\}$. Note that while WFAs do not have a set of acceptance states, runs that reach states q for which $\tau(q) = \infty$ have cost ∞, thus the function τ can be viewed as a refinement of the partition of the state space to accepting and rejecting states. The weighted language of $\mathcal{A}$, denoted $L(\mathcal{A})$, maps each word $w \in \Sigma^*$ to $cost(\mathcal{A}, w)$.

We assume that all states $q \in Q$ are reachable in $\mathcal{A}$. We assume that all states, except maybe the initial states are not empty, in the sense they map at least one word to a finite cost. Thus, for all $q \in Q$ there is $w \in \Sigma^*$ such that the cost of w in $\mathcal{A}$ with initial state q is in $\mathbb{R}$. Finally, given two WFAs $\mathcal{A}$ and $\mathcal{A}'$, we say that $\mathcal{A}$ is cheaper than $\mathcal{A}'$, denoted $\mathcal{A} \preceq \mathcal{A}'$, if for every word $w \in \Sigma^*$, we have that $cost(\mathcal{A}, w) \leq cost(\mathcal{A}', w)$.

2.2 Input Cheating and Resilience of Automata

Recall that a WFA induces a weighted language that maps each word to a cost in $\mathbb{R}^{\geq 0} \cup \{\infty\}$. Words may cheat the automaton hoping to be mapped to a lower cost: When the automaton runs on a word $w = w_1 \dots w_n \in \Sigma^*$, then in each position $1 \leq i \leq n$, the word can cheat the automaton and report that the letter w_i is a different letter $w'_i \in \Sigma$. Cheating has a price, and the setting includes a *penalty function* $\eta : \Sigma \times \Sigma \to \mathbb{R}^{\geq 0}$, satisfying $\eta(\sigma, \sigma) = 0$, and a *detection-probability function* $p : \Sigma \times \Sigma \to [0, 1]$ indicating the probability of catching

[1] In general, a WFA may be defined with respect to any semiring $\langle \mathbb{K}, \oplus, \otimes, \overline{0}, \overline{1} \rangle$. The cost of a run is then the semiring product of the weights along it, and the cost of a word is the semiring sum over all runs on it. For our purposes, we focus on weighted automata defined with respect to the *min-sum semiring*, $\langle \mathbb{R}^{\geq 0} \cup \{\infty\}, \min, +, \infty, 0 \rangle$ (sometimes called the *tropical* semiring), as defined above.

each specific cheat. Formally, whenever σ is reported to be σ', the automaton detects the cheating with probability $p(\sigma, \sigma')$, in which case it charges $\eta(\sigma, \sigma')$. The expected penalty for reporting σ to be σ' is therefore $\eta(\sigma, \sigma') \cdot p(\sigma, \sigma')$.

For two words $w = w_1, w_2, \ldots, w_n$ and $w' = w'_1, w'_2, \ldots, w'_n$, the expected cost of reporting w to be w' is $\sum_{i=1}^{n} \eta(w_i, w'_i) \cdot p(w_i, w'_i)$. Given a WFA $\mathcal{A}$, a penalty function η, a detection-probability function p, and two words w, w' such that $|w| = |w'|$, the expected cost of w in $\mathcal{A}$ when w is reported to be w', denoted $\mathit{expected\_faked\_cost}(\mathcal{A}, \eta, p, w, w')$, is $\mathit{cost}(\mathcal{A}, w') + \sum_{i=1}^{n} \eta(w_i, w'_i) \cdot p(w_i, w'_i)$. Finally, $\mathit{expected\_best\_cost}(\mathcal{A}, \eta, p, w)$ is the lowest expected cost with which w can be read by $\mathcal{A}$ (with or without cheating). Thus, $\mathit{expected\_best\_cost}(\mathcal{A}, \eta, p, w) = \min_{w':|w'|=|w|} \mathit{expected\_faked\_cost}(\mathcal{A}, \eta, p, w, w')$. We refer to the word w' with which the minimum is achieved as the *cheating pattern* for w.

We say that $\mathcal{A}$ is (η, p)-*resilient to cheating* if it is not worthwhile to cheat $\mathcal{A}$ given the penalty function η and the detection-probability function p. Formally, $\mathcal{A}$ is (η, p)-resilient to cheating if for every input word w, it holds that $\mathit{cost}(\mathcal{A}, w) = \mathit{expected\_best\_cost}(\mathcal{A}, \eta, p, w)$.

Studying resilience of automata, it is convenient to consider a non-probabilistic setting in which cheats are always detected. We use $\hat{1}$ denote the detection-probability function satisfying $\hat{1}(\sigma, \sigma') = 1$ for all $\sigma, \sigma' \in \Sigma$. As argued in Theorem 1 below, the probabilistic setting can be easily reduced to the non-probabilistic one. The theorem follows easily from the linearity of expectation.

Theorem 1. *Consider a WFA $\mathcal{A}$, penalty function η, and detection-probability function p. Let $\theta = \eta \circ p$. Thus, $\theta : \Sigma \times \Sigma \to \mathbb{R}^{\geq 0}$ is such that for all $\sigma, \sigma' \in \Sigma$, we have that $\theta(\sigma, \sigma') = \eta(\sigma, \sigma') \cdot p(\sigma, \sigma')$. Then, for every $w \in \Sigma^*$, we have $\mathit{expected\_best\_cost}(\mathcal{A}, \eta, p, w) = \mathit{expected\_best\_cost}(\mathcal{A}, \theta, \hat{1}, w)$.*

Thus, by considering the penalty function $\theta = \eta \circ p$, we can reduce a probabilistic setting with η and p to a non-probabilistic one. The cost of a word in $\mathcal{A}$ is still an expected one, but for simplicity of notations, we use the terms $\mathit{faked\_cost}(\mathcal{A}, \theta, w, w')$ and $\mathit{best\_cost}(\mathcal{A}, \theta, w)$, which are analogue to the terms $\mathit{expected\_faked\_cost}(\mathcal{A}, \eta, p, w, w')$ and $\mathit{expected\_best\_cost}(\mathcal{A}, \eta, p, w)$, and refer to θ-resilience to cheating, rather than (η, p)-resilience.

Example 1. Consider the DWFA $\mathcal{A}$ in Figure 1. Every state q_i in the figure is labelled by its final cost. For example, $\tau(q_4) = 4$, and $\tau(q_3) = x$, for some $x \in \mathbb{R}$. Every transition is labelled by the letter and cost associated with it. For example, $\Delta(q_2, b, q_5)$ and $c(q_2, b, q_5) = 1$. Assume that the penalty function is uniform and for all $\sigma, \sigma' \in \{a, b, c\}$ with $\sigma \neq \sigma'$, we have $\theta(\sigma, \sigma') = 2$.

The DWFA $\mathcal{A}$ demonstrates two of the phenomenon that makes the analysis of cheating challenging. First, testing an WFA for θ-resilience (even a DWFA, and even with a uniform θ) may not be local. In our example, if we take $x = 0$, then it is easy to see that for every three states q, q', and q'', and two letters σ and σ', it holds that $c(q, \sigma, q') + \tau(q') \leq c(q, \sigma', q'') + \tau(q'') + \theta(\sigma, \sigma')$; that is, for all words of length 1 it is not beneficial to cheat, independent of the initial state. Clearly, this is a necessary condition for $\mathcal{A}$ to be θ-resilient: if there are q, q', q'', σ, and σ' that violate the condition, then the word $w \cdot \sigma$

for which $\delta(q_0, w) = q$, has $faked\_cost(\mathcal{A}, \theta, w \cdot \sigma, w \cdot \sigma') < cost(\mathcal{A}, w \cdot \sigma)$, thus $best\_cost(\mathcal{A}, \theta, w \cdot \sigma) < cost(\mathcal{A}, w \cdot \sigma)$ and $w \cdot \sigma$ has an incentive to cheat and pretend to be $w \cdot \sigma'$. This condition, however, is not sufficient. For example, $cost(\mathcal{A}', aa) = 8$ while $faked\_cost(\mathcal{A}, \theta, aa, bb) = 2 + 2\theta(a, b) = 6$. That is, aa has an incentive to cheat and pretend to be bb.

Second, $\mathcal{A}$ demonstrates that cheating may be beneficial only for words that are unboundedly long. To see this, note that $cost(\mathcal{A}, bc^k) = k + 1$ and $cost(\mathcal{A}, c^{k+1}) = x + 1$. Since cheating in the first letter costs 2, we have that $best\_cost(\mathcal{A}, \theta, bc^k) = \min(k + 1, x + 3)$ and $best\_cost(\mathcal{A}, \theta, c^{k+1}) = \min(k + 3, x + 1)$. Thus, the larger x is, the longer are the shortest input words that have an incentive to cheat.

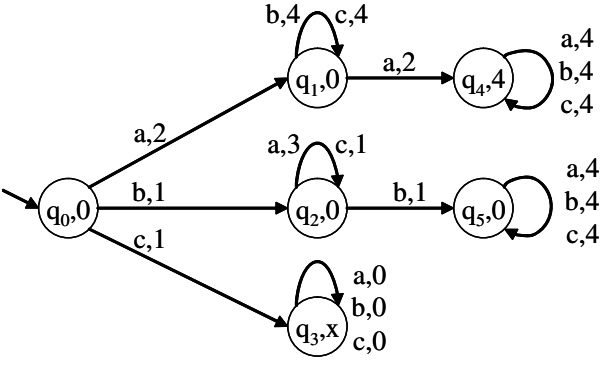


Fig. 1. The DWFA $\mathcal{A}$

A basic challenge in the setting of rational environments is to design systems in which the environment has no incentive to cheat. In our setting, one could ask whether a given WFA $\mathcal{A}$ that is not θ-resilient to cheating can be modified to an equivalent WFA $\mathcal{A}'$ that is θ-resilient to cheating. Theorem 2 below states that this is impossible.

Theorem 2. *Resilience to cheating is a semantic property. That is, given a weighted language $L : \Sigma^* \to \mathbb{R}^{\geq 0} \cup \{\infty\}$ and a penalty function θ, either all WFAs for $\mathcal{L}$ are θ-resilient to cheating, or none of them is θ-resilient to cheating.*

Note that Theorem 2 applies for both nondeterministic and deterministic WFAs. Thus, nondeterminism cannot help a WFA to cope with cheats. Note also that Theorem 2 considers a given penalty function θ and does not include the possibility of achieving resilience by modifying the penalty function, possibly using the same budget. We will get back to this problem in Section 4.

2.3 The Cheating-Allowed Automaton

Reasoning about a WFA $\mathcal{A}$ and its resilience to cheating, one has to take into account the infinitely many possible cheating patterns that $\mathcal{A}$ should be resilient

too. In this section we show that these patterns can be modelled by a single WFA obtained from $\mathcal{A}$ by adding transitions that mimics cheating.

Theorem 3. *Consider a WFA $\mathcal{A}$ and a penalty function $\theta : \Sigma \times \Sigma \to \mathbb{R}^{\geq 0}$. There is a WFA $\mathcal{A}'$, with the same state space as $\mathcal{A}$, such that $cost(\mathcal{A}', w) = best\_cost(\mathcal{A}, \theta, w)$.*

Proof. Let $\mathcal{A} = \langle \Sigma, Q, \Delta, c, q_0, \tau \rangle$. We define $\mathcal{A}' = \langle \Sigma, Q, \Delta', c', q_0, \tau \rangle$, where the transition relation Δ' and the cost function c' are defined as follows. For every two states $q, q' \in Q$, if there is $\sigma' \in \Sigma$ such that $\Delta(q, \sigma', q')$, then $\Delta'(q, \sigma, q')$ for every $\sigma \in \Sigma$, and $c'(q, \sigma, q') = \min_{\sigma' : \Delta(q, \sigma', q')} \{c(q, \sigma', q') + \theta(\sigma, \sigma')\}$. That is, if the set Σ' of letters with which $\mathcal{A}$ can move from q to q' is not empty, then $\mathcal{A}'$ can move from q to q' with all letters – by reporting them to be some letter in Σ'. The cost of this transition for a letter σ is calculated by taking the most beneficial replacement from Σ': the one that minimizes the sum of the cost of the transition and the cost of cheating.

It is not hard to see the correspondence between the nondeterminism of $\mathcal{A}'$ and the choices of cheating patterns. Formally, for every word w, a cheating pattern w' for w induces a run of $\mathcal{A}'$ on w whose cost is $faked\_cost(\mathcal{A}, \theta, w, w')$. Likewise, every run of $\mathcal{A}'$ on w induces a word w' that can serve as a cheating pattern for w. Hence, since the cost of w in $\mathcal{A}'$ is the minimal cost of some run of $\mathcal{A}'$ on w, we have that $best\_cost(\mathcal{A}, \theta, w) = cost(\mathcal{A}', w)$, and we are done.

Given a WFA $\mathcal{A}$ and a penalty function θ, we refer to the WFA $\mathcal{A}'$ constructed in Theorem 3 as $Cheat(\mathcal{A}, \theta)$. For example, the WFA in Figure 2 is $Cheat(\mathcal{A}, \theta)$, for the WFA $\mathcal{A}$ described in Figure 1 and $\theta(\sigma, \sigma') = 2$ for all $\sigma, \sigma' \in \Sigma$ with $\sigma \neq \sigma'$.

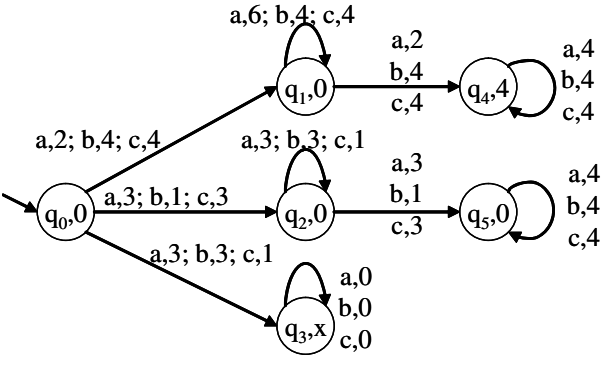


Fig. 2. The WFA $\mathcal{A}' = Cheat(\mathcal{A}, \theta)$, with uniform $\theta = 2$

Corollary 1. *For every WFA $\mathcal{A}$ and penalty function θ, we have that $\mathcal{A}$ is θ-resilient to cheating iff $\mathcal{A} \preceq Cheat(\mathcal{A}, \theta)$, that is, for every word $w \in \Sigma^*$, we have that $cost(\mathcal{A}, w) \leq cost(Cheat(\mathcal{A}, \theta), w)$.*

Theorem 4. *Given a WFA $\mathcal{A}$, a penalty function θ, and a word $w \in \Sigma^*$, the problem of finding best_cost$(\mathcal{A}, \theta, w)$ and a cheating pattern for it, can be solved in polynomial time.*

Proof. Given a WFA $\mathcal{A}$ and a word $w \in \Sigma^*$, it is possible to find $cost(\mathcal{A}, w)$ as follows (note that we refer here to cost without cheating). If $\mathcal{A}$ is deterministic, we traverse the single run of $\mathcal{A}$ on w and find its cost. If $\mathcal{A}$ is nondeterministic, we first restrict $\mathcal{A}$ to runs along which w is read, and then find the cheapest such run. Formally, we define the product $\mathcal{A}_w$ of $\mathcal{A}$ with an un-weighted automaton with $|w|+1$ states whose language is $\{w\}$. The WFA $\mathcal{A}_w$ describes exactly all the run of $\mathcal{A}$ on w and it has no cycles. We apply to $\mathcal{A}_w$ a shortest-path algorithm [Dij59] and find the shortest path from an initial state to a final state.

Now, given $\mathcal{A}$ and θ, let $\mathcal{A}'$ be $Cheat(\mathcal{A}, \theta)$. Then, for every word w, we have that $best\_cost(\mathcal{A}, \theta, w) = cost(\mathcal{A}', w)$, which can be calculated as described above. Also, the run r' of $\mathcal{A}'$ on w for which $cost(\mathcal{A}', w) = cost(r', w)$ reveals the cheating pattern.

Limited Cheating and Rising Penalty Variants: In the above described setting, an input word can cheat as many times as it wants. Also, the penalties are fixed throughout the interaction. It is easy to modify the construction of $Cheat(\mathcal{A}, \theta)$ and, consequently, our results below, to account for variant models. For example, by taking several copies of $Cheat(\mathcal{A}, \theta)$, it is possible to give a constant bound on the number of allowed cheats (the states maintain the number of cheats detected so far) or constant bound on the budget a word can use for cheating (the states maintain the total cheating costs detected so far). By taking several copies of $Cheat(\mathcal{A}, \theta)$ and modifying the costs in the different copies, it is possible to let $\mathcal{A}$ increase the penalties when cheats are detected (this corresponds to increasing either the detection-probability function or the penalties themselves; as indeed happens in practice when cheats are detected).

3 Resilience Testing

In this section we study the problem of deciding, given a WFA $\mathcal{A}$ and a penalty function θ, whether $\mathcal{A}$ is θ-resilient to cheating. Recall that $\mathcal{A}$ is θ-resilient to cheating if $cost(\mathcal{A}, w) = best\_cost(\mathcal{A}, \theta, w)$. We show that the problem is PSPACE-hard for WFA but can be solved in polynomial time for DWFA.

3.1 Hardness Proof for WFA

Theorem 5. *Consider a WFA $\mathcal{A}$ and a penalty function θ. The problem of deciding whether $\mathcal{A}$ is θ-resilient is PSPACE-hard.*

Proof. We do a reduction from the universality problem for NFAs, proven to be PSPACE-hard in [RS59]. Given an NFA $\mathcal{U}$, we construct a WFA $\mathcal{A}_\mathcal{U}$ such that $\mathcal{A}_\mathcal{U}$ is 0-resilient (that is, $\theta(\sigma, \sigma') = 0$ for all $\sigma, \sigma' \in \Sigma$) iff $\mathcal{U}$ is universal. Note that an automaton is 0-resilient iff no input word has an incentive to cheat even

if cheating is free. The idea behind the construction is that words not in $L(\mathcal{U})$ would induce words that have an incentive to cheat $\mathcal{A}_{\mathcal{U}}$. Thus, $\mathcal{U}$ is universal iff no word has an incentive to cheat $\mathcal{A}_{\mathcal{U}}$, so even the 0 penalties suffice to ensure resilience. Formally, let $\mathcal{U} = \langle \Sigma, Q, \Delta, Q_0, F \rangle$, where $F \subseteq Q$ is a set of final states, and let a be some letter in Σ. We assume that $|\Sigma| > 1$. We define $\mathcal{A}_{\mathcal{U}}$ to go with the letter a to a copy of $\mathcal{U}$ and to go with all letters $\Sigma \setminus \{a\}$ to an accepting sink (see Figure 3). Thus, $\mathcal{A}_{\mathcal{U}} = \langle \Sigma, Q \cup \{q_0, q_{qcc}\}, \Delta', \{q_0\}, c, \tau \rangle$, where $\Delta' = \Delta \cup (\{q_0\} \times \{a\} \times Q_0) \cup (\{q_0\} \times (\Sigma \setminus \{a\}) \times \{q_{acc}\}) \cup (\{q_{acc}\} \times \Sigma \times \{q_{acc}\})$.

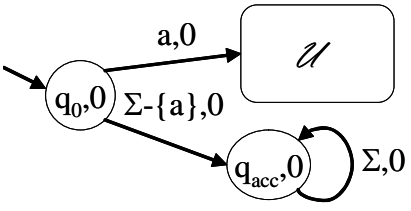


Fig. 3. The WFA $\mathcal{A}_{\mathcal{U}}$

Also, for all $\langle q, \sigma, q' \rangle \in \Delta'$, we have $c(\langle q, \sigma, q' \rangle) = 0$ and for all $q \in Q \cup \{q_0, q_{qcc}\}$ we have $\tau(q) = 0$. It is easy to see that $\mathcal{A}_{\mathcal{U}}$ accepts (with cost 0) all words of the form $a \cdot w$, for $w \in L(\mathcal{U})$, or of the form $\sigma \cdot w$, for $\sigma \neq a$ and $w \in \Sigma^*$. Accordingly, if $\mathcal{U}$ is universal, then $\mathcal{A}_{\mathcal{U}}$ accepts all words in Σ^* with cost 0, and is therefore 0-resilient. Also, if $\mathcal{U}$ is not universal, then there is $w \notin L(\mathcal{U})$ such that $\mathit{cost}(\mathcal{A}_{\mathcal{U}}, a \cdot w) = \infty$, while $\mathit{faked\_cost}(\mathcal{A}_{\mathcal{U}}, a \cdot w, b \cdot w) = \theta(a, b)$, for any $b \in \Sigma \setminus \{a\}$. Hence, $\mathcal{A}_{\mathcal{U}}$ is not 0-resilient, and we are done.

Many fundamental problems about WFAs are still open. Unlike standard (non-weighted) automata, not all weighted automata can be determinized [Moh97]. In fact, even the problem of deciding whether a given WFA has an equivalent DWFA is open, and so are problems that use determinization in their solution, like deciding whether $\mathcal{A} \preceq \mathcal{A}'$ for two WFAs $\mathcal{A}$ and $\mathcal{A}'$ [Kro94, CDH08]. We note that the problem of deciding whether $\mathcal{A} \preceq \mathcal{A}'$ is open even when $\mathcal{A}$ is a DWFA – it is the nondeterminism in $\mathcal{A}'$ that makes the problem challenging. Thus, even for the case $\mathcal{A}$ is deterministic, we cannot reduce the problem of deciding whether $\mathcal{A} \preceq \mathit{Cheat}(\mathcal{A}, \theta)$ to a problem whose solution is known. As we describe below, we are still able to present a polynomial solution to the problem.

3.2 A Polynomial Algorithm for DWFA

We turn to consider the case where $\mathcal{A}$ is deterministic. We show that in this case, the problem of deciding whether $\mathcal{A}$ is θ-resilient, for a given penalty function θ, can be solved in polynomial time. Let $\mathcal{A} = \langle \Sigma, Q, \Delta, c, q_0, \tau \rangle$ be a DWFA. Let $n = |Q|$. For a given penalty function θ, let $\mathcal{A}' = \langle \Sigma, Q, \Delta', c', q_0, \tau \rangle$ be $\mathit{Cheat}(\mathcal{A}, \theta)$. We describe an algorithm for deciding whether $\mathcal{A} \preceq \mathcal{A}'$. By Corollary 1, the latter holds iff $\mathcal{A}$ is θ-resilient to cheating.

Our algorithm is similar to the algorithm for deciding whether a given DWFA is equivalent to a WFA in which it is embodied [AKL09]. We define a sequence of functions $h_0, h_1, \ldots : Q \times Q \to \mathbb{R} \cup \{\infty, -\infty\}$, as follows.[2] Intuitively, $h_i(q, q')$ indicates how much a word of length at most i can gain if instead of a run of $\mathcal{A}$ that leads to q it takes a run of $\mathcal{A}'$ that leads to q'. This difference does not include the final costs of q, and q'. Note that there may not be words of length at most i along which q and q' are reachable, in which case $h_i(q, q')$ would be $-\infty$. Also, it may be that for all words w of length at most i, the cheapest run in $\mathcal{A}'$ that reads w and leads to q' costs more than the run of $\mathcal{A}$ that reads w and leads to q, in which case $h_i(q, q')$ is negative.

It is easy to see that if for some $i \in \mathbb{N}$ and $q, q' \in Q$, we have that $h_i(q, q') > \tau(q') - \tau(q)$, then there is a word of length at most i for which $cost(\mathcal{A}, w) > cost(\mathcal{A}', w)$, thus $\mathcal{A} \not\preceq \mathcal{A}'$. We show that h_i can be calculated efficiently, and that even though the sequence of functions may not reach a fixed-point, it is possible to determine whether $\mathcal{A} \preceq \mathcal{A}'$ after calculating h_i for $i = 0, \ldots, O(n^2)$. Intuitively, it follows from the fact that not reaching a fixed-point after $O(n^2)$ iterations points to cycles along which the gain of $\mathcal{A}'$ with respect to $\mathcal{A}$ is unbounded.

We initialize $h_0(q_0, q_0) = 0$ and $h_0(q, q') = -\infty$ for all other pairs. Indeed, (q_0, q_0) is the only pair of states to which an empty word might reach on $\mathcal{A}$ and $\mathcal{A}'$.

The calculation of h_{i+1}, for $i \geq 0$, uses a function $g_{i+1} : Q \times Q \times \Sigma \to \mathbb{R} \cup \{\infty, -\infty\}$. Intuitively, $g_{i+1}(q, q', \sigma)$ indicates how much a word of length at most $i+1$ that ends with the letter σ can gain if instead of a run of $\mathcal{A}$ that leads to q it takes a run of $\mathcal{A}'$ that leads to q'. Then,

$$g_{i+1}(q, q', \sigma) = \max_{p,p' : \Delta(p,\sigma,q) \ \wedge \ \Delta'(p',\sigma,q')} (h_i(p, p') + c(p, \sigma, q) - c'(p', \sigma, q')). \quad (1)$$

Thus, the calculation of $g_{i+1}(q, q', \sigma)$ considers all pairs $\langle p, p' \rangle \in Q$ from which q and q' can be reached, respectively, when a is read. Since $g_{i+1}(q, q', \sigma)$ is the gain obtained by running in $\mathcal{A}'$ instead of in $\mathcal{A}$, we add to $h_i(p, p')$ the cost of the transition $\langle p, \sigma, q \rangle$ in $\mathcal{A}$ and subtract the cost of the transition $\langle p', \sigma, q' \rangle$ in $\mathcal{A}'$. Now, for $i \geq 0$, we have

$$h_{i+1}(q, q') = \max\{h_i(q, q'), \max_{\sigma \in \Sigma} g_{i+1}(q, q', \sigma)\}. \quad (2)$$

For $i \geq 0$ and $q, q' \in Q$, we say that a word w *witnesses* $h_i(q, q')$ if $|w| \leq i$ and there is a run of $\mathcal{A}'$ on w that leads to q' and traversing its transitions costs $h_i(q, q')$ less than traversing the transitions of the run of $\mathcal{A}$ on w, which leads to q. Note that since the functions h_i ignore the final costs, the above refers to the cost of traversing the transitions along the runs, rather than the cost of the runs. Clearly, if $h_i(q, q')$ is finite, then it has at least one witness.

[2] In the definition of h_i we use addition and subtraction on the elements of $\mathbb{R} \cup \{\infty, -\infty\}$. For every finite $x \in \mathbb{R}$, we have $\infty - x = \infty$, and $x - \infty = -\infty$. Also $\infty - \infty = 0$.

We can now present the algorithm for deciding whether $\mathcal{A} \preceq A'$:

1. For $i = 0, \ldots, n^2$: Calculate h_i; if for some $q, q' \in Q$, we have $h_i(q, q') > \tau(q') - \tau(q)$, then return $(\mathcal{A} \not\preceq \mathcal{A}')$.
2. For $i = n^2+1, \ldots, 2n^2$: Calculate h_i; if for some $q, q' \in Q$, we have $h_{i-1}(q, q') < h_i(q, q')$, then return $(\mathcal{A} \not\preceq \mathcal{A}')$.
3. Return$(\mathcal{A} \preceq \mathcal{A}')$.

The correctness of the algorithm is proven in the full version. The function h_0 can be calculated in polynomial time, and so is the function h_{i+1}, given h_i. Hence, since we need only a polynomial number of iterations, we can conclude with the following.

Theorem 6. *Consider a DWFA $\mathcal{A}$ and a penalty function θ. The problem of deciding whether $\mathcal{A}$ is θ-resilient can be solved in polynomial time.*

4 Achieving Resilience with Minimum Resources

A system with no limit on penalties and with unbounded resources can prevent cheating by fixing a high penalty function. In practice, penalties may be limited by an external authority, and increasing the probability of detecting cheats requires resources. In this section we study the problem of minimizing the resources required in order to guarantee resilience.

We assume that the penalty function η is determined by an external authority and that $\mathcal{A}$ is $(\eta, \hat{1})$-resilient. Thus, the environment has no incentive to cheat if cheating is always detected.[3] Given a WFA $\mathcal{A}$, and a penalty function η, our goal is to find a detection-probability function p, such that $\mathcal{A}$ is (η, p)-resilient to cheating and the budget $B = \sum_{\sigma,\sigma' \in \Sigma} \eta(\sigma, \sigma') \cdot p(\sigma, \sigma')$ is minimal. The rationale behind our goal is that the system can control the probability of catching cheats. In practice, detection probability can be increased by investing in "guards", each responsible for a specific possible cheat. The budget we have is the total payment for the guards. The payment to the guard responsible for detecting σ being reported as σ' is independent of the actual number of times σ is being reported as σ'. On the other hand, the payment is proportional to the penalty $\eta(\sigma, \sigma')$ charged whenever the guard detects the cheat. Indeed, in practice, detecting a cheat with a high penalty tends to require high resources: knowing that his success leads to a high revenue, a guard would require high salary. We say that $\mathcal{A}$ *can achieve resilience with budget* B if there are η and p such that the budget of η and p is B, and $\mathcal{A}$ is (η, p)-resilient to cheating.

As explained in Section 2.2, we can consider an equivalent non-probabilistic setting in which all cheats are always detected and are charged according to the penalty function $\theta = \eta \circ p$. In the rest of this section we therefore consider the problem of deciding, given a WFA $\mathcal{A}$ and a budget $B \in \mathbb{R}^{\geq 0}$, whether $\mathcal{A}$ can achieve resilience with budget B, as well as the optimization problem of

[3] Note that this is a reasonable assumption as otherwise, the authority providing the penalty function encourages cheating.

finding the minimal budget with which $\mathcal{A}$ can achieve resilience. A solution for the above problems induces the expected-fee function θ. Having θ in hand, we use the given penalty function η to fix $p(\sigma,\sigma') = \frac{\theta(\sigma,\sigma')}{\eta(\sigma,\sigma')}$. In order to guaranteed that our solution is feasible, that is, the probability function is over the rage $[0,1]$, our algorithm only considers solutions in which for all $\sigma,\sigma' \in \Sigma$ we have $\eta(\sigma,\sigma') \geq \theta(\sigma,\sigma')$.

4.1 Hardness Proof for WFA

We first show that, as in the resilience testing problem, the nondeterministic setting is much more difficult.

Theorem 7. *Consider a WFA $\mathcal{A}$. Given a budget B, the problem of deciding whether there is a penalty function θ with budget B such that $\mathcal{A}$ is θ-resilient to cheating is PSPACE-hard.*

Proof. As in the proof of Theorem 5, we do a reduction from the universality problem for NFAs. Given an NFA $\mathcal{U}$, we construct a WFA $\mathcal{A}_\mathcal{U}$ such that there is a penalty function θ with budget 0 with which $\mathcal{A}_\mathcal{U}$ is θ-resilient to cheating iff $\mathcal{U}$ is universal.

The construction is similar to the one described in the proof of Theorem 5, except that now the transition from q_0 to q_{acc} is labelled by both all the letters in $\Sigma\setminus\{a\}$, with cost 0, and the letter a, with cost 1. It is easy to see that the cost in $\mathcal{A}_\mathcal{U}$ of words of the form $a\cdot w$ is 0 for $w \in L(\mathcal{A})$ and is 1 for $w \notin L(\mathcal{A})$. Also, for $\sigma \neq a$, the cost of words of the form $\sigma \cdot w$ is 0, regardless of the membership of w in $L(\mathcal{A})$. Accordingly, if $\mathcal{U}$ is universal, then $\mathcal{A}_\mathcal{U}$ accepts all words in Σ^* with cost 0, and is therefore 0-resilient, in which a budget 0 suffices to ensure resilience. Also, if $\mathcal{U}$ is not universal, then there is $w \notin L(\mathcal{U})$ such that $cost(\mathcal{A}_\mathcal{U}, a\cdot w) = 1$, while $faked\_cost(\mathcal{A}_\mathcal{U}, a\cdot w, b\cdot w) = \theta(a,b)$, for any $b \in \Sigma\setminus\{a\}$. Hence, in order to ensure θ-resilience, a penalty function θ must satisfy $\theta(a,b) \geq 1$, thus the budget required to θ is at least $|\Sigma| - 1$, and we are done.

4.2 A Polynomial Algorithm for DWFA

We turn to consider deterministic WFAs. Note that if we define an order $\leq$ between penalty functions, where $\theta_1 \leq \theta_2$ iff $\theta_1(\sigma,\sigma') \leq \theta_2(\sigma,\sigma')$ for all $\sigma,\sigma' \in \Sigma$, then the penalty functions that ensure resilience are not linearly ordered. This last observation hints that the problem of finding a minimal sufficient penalty with respect to which $\mathcal{A}$ is resilient cannot be solved in a straightforward way, as it cannot be based on a search in a linearly ordered domain. Still, as we show below, when $\mathcal{A}$ is a deterministic DFA, it is possible to describe the resilience requirements as a set of linear inequality constraints. Since the optimization objective can be also described as a linear function, it is possible to determine the minimal sufficient penalty function using linear programming (LP). LP is a mathematical tool suitable for determining an optimal solution for a linear objective function defined over a set of variables, while obeying a set of requirements represented as linear equations [Chv83].

We describe the problem as a linear programming optimization problem with a polynomial number of variables and constraints. Given a WFA $\mathcal{A}$ and a penalty function η, the algorithm returns a new penalty function θ such that:

1. $\sum_{\sigma,\sigma' \in \Sigma} \theta(\sigma, \sigma')$ is minimal.
2. For all $\sigma, \sigma' \in \Sigma$, we have $0 \leq \frac{\theta(\sigma,\sigma')}{\eta(\sigma,\sigma')} \leq 1$.
3. $\mathcal{A}$ is θ-resilient.

Note that the second property assures that $\theta = \eta \circ p$, for some probability function p satisfying $p(\sigma, \sigma') \in [0, 1]$.

The first property defines the objective function of the LP. The LP constraints assure the second and third properties. Specifically, for the third property, the LP constraints assure that the algorithm described in Section 3.2, for testing whether $\mathcal{A}$ is θ-resilient, would return $\mathcal{A} \preceq \mathit{Cheat}(\mathcal{A}, \theta)$. Accordingly, the variables we use are the following:

- For all $\sigma, \sigma' \in \Sigma$, the variable $\theta_{\sigma,\sigma'}$ maintains the penalty function $\theta(\sigma, \sigma')$.
- For $0 \leq i \leq 2n^2$ and $q, q' \in Q$, the variable $h_{i,q,q'}$ maintains $h_i(q, q')$.
- For $0 \leq i \leq 2n^2$, $q, q' \in Q$, and $\sigma \in \Sigma$, the variable $g_{i,q,q',\sigma}$ maintains $g_i(q, q', \sigma)$.

The objective function is $\min \sum_{\sigma,\sigma'} \theta_{\sigma,\sigma'}$. Since the penalty function is non-negative, we have $|\Sigma|^2$ constraints $\theta_{\sigma,\sigma'} \geq 0$ for all $\sigma, \sigma' \in \Sigma$. In addition, $\theta_{\sigma,\sigma} = 0$ for all $\sigma \in \Sigma$. Also, in order to guarantee that the detection-probability function is feasible, we have, for all $\sigma, \sigma' \in \Sigma$, the constraint $\theta_{\sigma,\sigma'} \leq \eta_{\sigma,\sigma'}$.

The additional constraints follow the structure of the algorithm presented in Section 3.2. For $k = 1, \ldots, n^2$, the k-th set of constraints assures that no word of length at most k should benefit from cheating. For $k = n^2 + 1, \ldots, 2n^2$, the k-th set of constraints assures that no cycle that can lead to unlimited gain exists. Each such set consists of a polynomial number of constraints and introduces a polynomial number of variables. Specifically, variables of type $h_{i,q,q'}$ bound the gain of words of length at most i, and variables of type $g_{i,q,q',\sigma}$ bound this gain for words of length at most i ending with σ. While the variables $h_{i,q,q'}, g_{i,q,q',\sigma}$ are defined for every $0 \leq i \leq 2n^2$, $q, q' \in Q$, and $\sigma \in \Sigma$, in practice, many of these variables are not constrained, as it might be that no word of length at most i can reach state q in $\mathcal{A}$ and q' in $\mathcal{A}'$.

We first describe the constraints considering words of length 1, and then the constraints for general k. Note that the first set of constraints can be viewed as a special case of the general set, however, since we know that q_0 is the only possible state preceding states reachable by a single letter, the presentation of this set is simpler. We also note that in order to clarify the intuition behind each constraint, the constraints are not necessarily presented in the canonical form of an LP (that is, with all variables in the left hand side and all constants in the right hand side).

In order to assure that words of length 1 will not cheat, we have a variable $h_{1,q,q'}$ for all $q, q' \in Q$, and a variable $g_{1,q,q',\sigma}$ for all $q, q' \in Q, \sigma \in \Sigma$. To reflect

Equation (1) in the algorithm described in Section 3.2, we have, for all $\sigma' \in \Sigma$ such that $\Delta(q_0, \sigma, q)$ and $\Delta(q_0, \sigma', q')$, the constraint $g_{1,q,q',\sigma} \geq c(q_0, \sigma, q) - c(q_0, \sigma', q') - \theta(\sigma, \sigma')$. To reflect Equation (2), we have, for all $q, q' \in Q$ and $\sigma \in \Sigma$ for which $g_{1,q,q',\sigma}$ is bounded, the constraint $h_{1,q,q'} \geq g_{1,q,q',\sigma}$. Since $h_0(q_0, q_0) = 0$ and the sequence of functions $h_0, h_1, \ldots$ is non-decreasing, we also have, for the state q_0, the constraint $h_1(q_0, q_0) \geq 0$. Finally, to reflect the comparison done in Step 1 of the resilience-testing algorithm, for all $q, q' \in Q$ we have the constraint $h_{1,q,q'} \leq \tau(q') - \tau(q)$.

In order to assure that words of length i do not cheat, we have a variable $h_{i,q,q'}$ for all $q, q' \in Q$, and a variable $g_{i,q,q',\sigma}$ for all $q, q' \in Q$ and $\sigma \in \Sigma$. To reflect Equation (1), we have, for all $p, p' \in Q$ and $\sigma' \in \Sigma$ such that $\Delta(p, \sigma, q)$ and $\Delta(p', \sigma', q')$, the constraint

$$g_{i,q,q',\sigma} \geq h_{i-1,p,p'} + c(p, \sigma, q) - c(p', \sigma', q') - \theta(\sigma, \sigma').$$

To reflect Equation (2), we have, for all $q, q' \in Q$ and $\sigma \in \Sigma$ for which $g_{i,q,q',\sigma}$ is bounded, the constraint $h_{i,q,q'} \geq g_{i,q,q',\sigma}$. Also, for all $q, q' \in Q$ we have the constraints $h_{i,q,q'} \geq h_{i-1,q,q'}$. Finally, for all $q, q' \in Q$ we have the constraint $h_{i,q,q'} \leq \tau(q') - \tau(q)$. This last type of constraints, considering the final costs, corresponds to the comparison done in Step 1 of the resilience-testing algorithm.

For $k = n^2 + 1 \ldots 2k^2$, the set of variables and the set of constraints are very similar to these sets for $k \leq n^2$. The only difference is the last type of constraints for every $q, q' \in Q$. Instead of $h_{i,q,q'} \leq \tau(q') - \tau(q)$, we have $h_{i,q,q'} \leq h_{i-1,q,q'}$. These constraints corresponds to the detection of gain increasing cycles, done in step 2 of the resilience testing algorithm.

The correctness of the following claim follows from the construction of the constraints.

Claim. The set of penalty functions in all feasible solutions to the LP is identical to the set of penalty functions for which the resilience algorithm provides a positive answer.

In particular, the feasible solution for which $\sum_{\sigma,\sigma'} \theta_{\sigma,\sigma'}$ is minimized, corresponds to a penalty function with minimal total budget. The total number of constraints and variables in our LP is polynomial in $|Q|$ and $|\Sigma|$. Therefore, it is possible to find an optimal solution for it [Kha79, Chv83] in polynomial time. This implies a polynomial algorithm for the minimum cost resilience problem of a DWFA. An example of a DWFA and its corresponding LP is given in the full version.

Acknowledgment. We thank Pnina and Yosef Bernholtz for many helpful discussions.

References

[AKL09] Aminof, B., Kupferman, O., Lampert, R.: Reasoning about online algorithms with weighted automata. In: Proc. 20th ACM-SIAM Symp. on Discrete Algorithms, pp. 835–844 (2009)

[CCH+05] Chakrabarti, A., Chatterjee, K., Henzinger, T.A., Kupferman, O., Majumdar, R.: Verifying quantitative properties using bound functions. In: Borrione, D., Paul, W. (eds.) CHARME 2005. LNCS, vol. 3725, pp. 50–64. Springer, Heidelberg (2005)

[CDH08] Chatterjee, K., Doyen, L., Henzinger, T.: Quantative languages. In: Proc. 17th Annual Conf. of the European Association for Computer Science Logic, pp. 385–400 (2008)

[Chv83] Chvatal, V.: Linear Programming. W.H. Freeman and Company, New York (1983)

[Dij59] Dijkstra, E.W.: A note on two problems in connexion with graphs. Numerische Mathematik 1, 269–271 (1959)

[DKe09] Droste, M., Kuich, W., Vogler, H. (eds.): Handbook of Weighted Automata. Springer, Heidelberg (2009)

[Eil74] Eilenberg, S.: Automata, Languages and Machines. Academic Press, San Diego (1974)

[FKL10] Fisman, D., Kupferman, O., Lustig, Y.: Rational synthesis. In: Proc. 16th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems. LNCS. Springer, Heidelberg (2010)

[Hen07] Henzinger, T.A.: Quantitative generalizations of languages. In: Development in Language Theory, pp. 20–22 (2007)

[HP85] Harel, D., Pnueli, A.: On the development of reactive systems. In: Apt, K. (ed.) Logics and Models of Concurrent Systems. NATO Advanced Summer Institutes, vol. F-13, pp. 477–498. Springer, Heidelberg (1985)

[Kha79] Khachiyan, L.G.: A polynomial algorithm in linear programming. Doklady Akademii Nauk SSSR 244, 1093–1096 (1979)

[Kro94] Krob, D.: The equality problem for rational series with multiplicities in the tropical emiring is undecidable. Journal of Algebra and Computation 4, 405–425 (1994)

[Moh97] Mohri, M.: Finite-state transducers in language and speech processing. Computational Linguistics 23(2), 269–311 (1997)

[NR99] Nisan, N., Ronen, A.: Algorithmic mechanism design. In: Proc. 31st ACM Symp. on Theory of Computing, pp. 129–140 (1999)

[NRTV07] Nisan, N., Roughgarden, T., Tardos, E., Vazirani, V.V.: Algorithmic Game Theory. Cambridge University Press, Cambridge (2007)

[PR89] Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: Proc. 16th ACM Symp. on Principles of Programming Languages, pp. 179–190 (1989)

[RS59] Rabin, M.O., Scott, D.: Finite automata and their decision problems. IBM Journal of Research and Development 3, 115–125 (1959)

[SS78] Salomaa, A., Soittola, M.: Automata: Theoretic Aspects of Formal Power Series. Springer, New York (1978)

[VW94] Vardi, M.Y., Wolper, P.: Reasoning about infinite computations. Information and Computation 115(1), 1–37 (1994)

Author Index