Nikolaj Bjørner
Andrei Voronkov (Eds.)

LNCS 7180 ARCoSS

Logic for Programming, Artificial Intelligence, and Reasoning

18th International Conference, LPAR-18
Mérida, Venezuela, March 2012
Proceedings

Springer

Lecture Notes in Computer Science 7180

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Advanced Research in Computing and Software Science
Subline of Lectures Notes in Computer Science

Nikolaj Bjørner Andrei Voronkov (Eds.)

Logic for Programming, Artificial Intelligence, and Reasoning

18th International Conference, LPAR-18
Mérida, Venezuela, March 11-15, 2012
Proceedings

Springer

Volume Editors

Nikolaj Bjørner
Microsoft Research
One Microsoft Way, Redmond, WA 98052-6399, USA
E-mail: nbjorner@microsoft.com

Andrei Voronkov
University of Manchester
School of computer Science
Kilburn Building, Oxford Road, Manchester, MP13 9PL, UK
E-mail: andrei.voronkov@manchester.ac.uk

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-28716-9 e-ISBN 978-3-642-28717-6
DOI 10.1007/978-3-642-28717-6
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2012932595

CR Subject Classification (1998): F.3, I.2, D.2, F.4.1, D.3, H.4

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

This volume contains the papers presented at LPAR-18: the 18th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning, held on March 11–15, 2012 in Merida, Venezuela.

Following the call for papers, LPAR-18 received 85 abstracts, materializing in 74 submissions. Each submission was reviewed by at least three of the 36 Program Committee members. The committee decided to accept 25 regular papers and 6 tool descriptions and experimental papers. The program also included talks by four distinguished invited speakers: Elvira Albert (Complutense University of Madrid), Kenneth McMillan (Microsoft Research), Aart Middeldorp (University of Innsbruck), and Boris Motik (University of Oxford); covering areas ranging from constraint programming and resource analysis of programs, software verification and interpolation, rewriting and matrix interpretations, and description logics.

Two workshops were co-located with LPAR-18. The 6th International Workshop on Analytic Proof Systems, APS-6, was organized by Matthias Baaz and Christian Fermüller, both from the University of Technology, Vienna. The 9th International Workshop on the Implementation of Logics was organized by Eugenia Ternovska (Simon Fraser University, Vancouver), Konstantin Korovin (University of Manchester), and Stephan Schulz (TU München). We were fortunate in having Laura Kovacs (Vienna University of Technology) acting as the LPAR workshop chair.

LPAR has a distinct track record as a series of high-quality conferences held in places where no other reasonable conference has gone before. The 18th conference in Merida takes LPAR to new heights, roughly 3500 meters. The PC Chairs are grateful for support from EasyChair and sponsorship from Microsoft Research. It also happens to be the case that one of the chairs is from Microsoft Research, the other is the author of EasyChair. The Conference Chair, Geoff Sutcliffe, stepped in for his fourth LPAR organization and secured the domain http://lpar-18.info for the conference. Finally, we thank the local organizers Blanca Abraham and José Aguilar for their support.

January 2012

Andrei Voronkov
Nikolaj Bjørner

Organization

Program Committee

José Aguilar	Universidad de Los Andes, Venezuela
Elvira Albert	Complutense University of Madrid, Spain
Franz Baader	TU Dresden, Germany
Gilles Barthe	IMDEA Software Institute, France
Peter Baumgartner	National ICT Australia
Armin Biere	Johannes Kepler University, Austria
Nikolaj Bjørner	Microsoft Research, USA
Thierry Coquand	Chalmers University, Sweden
Véronique Cortier	Loria, France
Luca de Alfaro	UCSC / Google, USA
Christian Fermüller	TU Vienna, Austria
John Harrison	Intel Corporation, USA
Manuel Hermengildo	IMDEA Software Institute, France
Barbara Jobstmann	CNRS/Verimag, France
Deepak Kapur	University of New Mexico, Mexico
Konstantin Korovin	Manchester University, UK
Laura Kovacs	TU Vienna, Austria
Carsten Lutz	Universität Bremen, Germany
Parthasarathy Madhusudan	University of Illinois at Urbana-Champaign, USA
Aart Middeldorp	University of Innsbruck, Austria
Dale Miller	INRIA Saclay - Île-de-France and LIX/ École Polytechnique, France
César Muñoz	National Aeronautics and Space Administration, USA
Albert Oliveras	Technical University of Catalonia, Spain
Lawrence Paulson	University of Cambridge, UK
Ruzica Piskac	Max Planck Institute for Software Systems, Germany
Francesca Rossi	University of Padova, Italy
Grigore Rosu	University of Illinois at Urbana-Champaign, USA
Torsten Schaub	University of Potsdam, Germany
Natarajan Shankar	SRI International, USA
Wolfgang Thomas	RWTH Aachen, Germany
Cesare Tinelli	The University of Iowa, USA
Pascal Van Hentenryck	Brown University, USA

Andrei Voronkov	University of Manchester, UK
Toby Walsh	NICTA and UNSW, Australia
Christoph Weidenbach	Max Planck Institute for Informatics, Germany
Frank Wolter	University of Liverpool, UK

Additional Reviewers

Accattoli, Beniamino
Alpuente, María
Andres, Benjamin
Arenas, Puri
Audemard, Gilles
Bana, Gergei
Booth, Richard
Boulmé, Sylvain
Brandner, Florian
Böhme, Sascha
Cerny, Pavol
Chaudhuri, Kaustuv
Ciobaca, Stefan
De Nivelle, Hans
Deters, Morgan
Doyen, Laurent
Ellison, Chucky
Everaere, Patricia
García Pérez, Alvaro
Garoche, Pierre-Loic
Gebser, Martin
Gelfond, Michael
Genaim, Samir
Geser, Alfons
Giesl, Jürgen
Giordano, Laura
Goodloe, Alwyn
Gutierrez, Raul
Gutiérrez, Basulto
Göller, Stefan
Haarslev, Volker
Haemmerlé, Rémy
Hagen, George
Heule, Marijn
Hoder, Krystof
Holloway, Michael
Hustadt, Ullrich
Järvisalo, Matti
Kristensen, Lars
Kutsia, Temur
König, Arne
Lamotte-Schubert, Manuel
Le Botlan, Didier
Leucker, Martin
Lopez-Garcia, Pedro
Lozes, Etienne
Löding, Christof
Meredith, Patrick
Moeller, Ralf
Montenegro, Manuel
Narkawicz, Anthony
Narodytska, Nina
Nguyen, Kim
Nguyen, Linh Anh
Nigham, Vivek
Olivetti, Nicola
Ostrowski, Max
Pacholski, Leszek
Parker, David
Pattinson, Dirk
Popescu, Andrei
Reynolds, Andrew
Riesco, Adrian
Rozier, Kristin Yvonne
Sabuncu, Orkunt
Schneider, Thomas
Serbanuta, Traian
Serebrenik, Alexander
Silva, Josep
Smyth, Ben
Stefanescu, Andrei
Sticksel, Christoph
Sürmeli, Jan
Tang, Ching Hoo
Thiemann, René
Thrane, Claus
Trunfio, Giuseppe A.
Veanes, Margus
Von Essen, Christian
Vyskocil, Jiri
Waldmann, Johannes
Waldmann, Uwe
Widmann, Florian
Wies, Thomas
Winkler, Sarah
Wintersteiger, Christoph M.
Wischnewski, Patrick
Zanardini, Damiano
Zankl, Harald
Zeilberger, Noam
Zuleger, Florian

Table of Contents

Automatic Inference of Resource Consumption Bounds

Elvira Albert[1], Puri Arenas[1], Samir Genaim[1], Miguel Gómez-Zamalloa[1], and Germán Puebla[2]

[1] DSIC, Complutense University of Madrid, Spain
[2] DLSIIS, Technical University of Madrid, Spain

Abstract. One of the main features of programs is the amount of resources which are needed in order to run them. Different resources can be taken into consideration, such as the number of execution steps, amount of memory allocated, number of calls to certain methods, etc. Unfortunately, manually determining the resource consumption of programs is difficult and error-prone. We provide an overview of a state of the art framework for automatically obtaining both upper and lower bounds on the resource consumption of programs. The bounds obtained are functions on the size of the input arguments to the program and are obtained statically, i.e., without running the program. Due to the approximations introduced, the framework can fail to obtain (non-trivial) bounds even if they exist. On the other hand, modulo implementation bugs, the bounds thus obtained are valid for any execution of the program. The framework has been implemented in the COSTA system and can provide useful bounds for realistic object-oriented and actor-based concurrent programs.

1 Introduction

One of the most important characteristics of a program is the amount of resources that its execution will require, i.e., its *resource consumption*. Resource analysis (a.k.a. cost analysis [37]) aims at *statically* bounding the cost of executing programs for any possible input data value. Typical examples of resources include execution time, memory watermark, amount of data transmitted over the net, etc. Resource usage information has many applications, both during program development and deployment. *Upper bounds* are useful because they provide *resource guarantees*, i.e., it is ensured that the execution of the program will never exceed the amount of resources inferred by the analysis. *Lower bounds* on the resource usage have applications in program parallelization, they can be used to decide if it is worth executing locally a task or requesting remote execution. Therefore, automated ways of estimating resource usage are quite useful and the general area of resource analysis has received [37,21,33] and is nowadays receiving [10,23,25] considerable attention. In this paper, we describe the main components underlying resource analysis of a today's imperative programming

N. Bjørner and A. Voronkov (Eds.): LPAR-18, LNCS 7180, pp. 1–11, 2012.

language, e.g., such techniques have been applied to analyze the resource consumption of sequential Java, Java bytecode [28], Featherweight X10 [27] and concurrent ABS programs [26].

The rest of the paper is organized as follows. Section 2 describes the process of, from the program, generating *cost relations* which define, by means of recurrence equations, the resource consumption of executing the program in terms of the input data sizes. Section 3 overviews a general approach to, from the cost relations, obtain upper and lower bounds which are not in recursive form. The compositionallity and incrementality of the analysis are described in Section 4. Standard cost analysis can be applied to infer any *cumulative* type of resource which always increases along the execution. We discuss in Section 5 the required extensions to estimate non-cumulative resources like the memory consumption in the presence of garbage collection. As distribution and concurrency are now mainstream, one of the most interesting extensions is to handle concurrency in cost analysis. This will be described in Section 6. The results of the analysis are correct only if the implementation does not contain bugs. In Section 7, we describe how the analysis results can be certified by existing program verification tools. Finally, in Section 8 we conclude and point out directions for future research.

2 Generation of Cost Relations

In the first phase, cost analysis takes as input a program, a selection of a cost model (among those available in the system), and yields a set of *recursive equations* which capture the cost of executing the program. For a general purpose programming language, the following steps are performed in order to generate the equations:

1 A *control flow graph* is constructed for each method in the original program by using standard techniques from *compiler theory* [1,2].

2 The control flow graph can then be represented by using some intermediate formalism, with the purpose of making the subsequent static analysis simpler. In [10], we propose that the control flow graph is represented as a set of procedures (defined by one or more rules) by using a *rule-based*, recursive *representation*.

3 Static analysis can be then performed on the rule-base representation in order to infer, for each rule, *size relations*, which define the size relationships among the input variables to the rule and the variables in the calls performed within the rule.

4 A parametric notion of *cost model* is used, which allows specifying the resource of interest (e.g., steps, memory). In particular, the cost model defines the cost assigned to each execution step and, by extension, to an entire execution trace.

5 From the rule-based representation, the size relations, and the selected cost model, a *cost relation system* is automatically generated. Cost relations are

```
static void m(List x, int i, int n){
  while (i<n){
    if (x.data) {g(i,n); i++;}
    else {g(0,i); n=n-1;}
    x=x.next;
  }}
```

$$(1)\ \langle C_m(i,n) = 3 \quad ,\varphi_1 = \{i \geq n\}\rangle$$
$$(2)\ \langle C_m(i,n) = 15 + C_g(i,n) + C_m(i',n) \quad ,\varphi_2 = \{i < n, i' = i+1\}\rangle$$
$$(3)\ \langle C_m(i,n) = 17 + C_g(0,i) + C_m(i,n') \quad ,\varphi_3 = \{i < n, n' = n-1\}\rangle$$

Fig. 1. Java method and Cost Relation

defined by means of recursive expressions which define the cost of executing a block in the control flow graph (or rule in the rule-based representation) in terms of the cost of executing the block itself plus the cost of its successor blocks.

All details about how to automatically obtain a cost relation from a program can be found in [8,10,22,32,37]. We illustrate the notion of cost relation systems by means of a simple example.

Example 1. Consider the Java method m shown in Figure 1 (left), which invokes the auxiliary method g, where x is a linked list of Boolean values implemented in the standard way. We have selected a cost model which counts the number of executed instructions. The cost relation associated to method m is shown in Figure 1 (right). The relations C_m and C_g capture, respectively, the costs of executing methods m and g. Intuitively, in cost relations, variables represent the sizes of the corresponding data structures in the program and in the case of integer variables they represent their integer value. Equation 1 is a base case and captures the case where the loop body is not executed. It can be observed that we have two recursive equations (Equations 2 and 3) which capture the respective costs of the then and else branches within the while loop. The constraints in the equations (namely φ_1, φ_2 and φ_3) contain the applicability conditions for the equations and also the size constraints computed in the previous step of the analysis (see Section 2). As the list x has been abstracted to its length, the values of x.data are not visible in the cost relation and the two equations have the same (incomplete) guard, which results in a *non-deterministic* cost relation. Also, variables which do not affect the cost (e.g., x) do not appear in the cost relation [9].

3 Inference of Closed-Form Bounds

The second main phase of cost analysis consists in generating *closed-form* bounds from the cost relations, i.e., cost expressions which are not in recursive form. Let us first see how we can infer an upper bound for m from the equations in Figure 1. We shall start by computing upper bounds for the cost relations which do not depend on any other cost relations, referred to as standalone cost relations, and continue by replacing the computed upper bounds on the equations

which call such relations. For instance, assuming that the upper bound for g is $C_g^+(a,b)$=4+5∗nat(b−a), where $\mathsf{nat}(v) = max(\{v,0\})$, the cost relation in Figure 1 becomes standalone:

$$\begin{array}{lll}
(1)\ \langle C_m(i,n) = 3 & , & \varphi_1 = \{i \geq n\}\rangle \\
(2)\ \langle C_m(i,n) = 15 + \mathsf{nat}(n-i) + C_m(i',n) & , & \varphi_2 = \{i < n, i' = i+1\}\rangle \\
(3)\ \langle C_m(i,n) = 17 + \mathsf{nat}(i) + C_m(i,n') & , & \varphi_3 = \{i < n, n' = n-1\}\rangle
\end{array}$$

The use of nat in cost expressions is required in order to avoid incorrectly evaluating upper bounds to negative values (see [7]), as for instance, when $b < a$ in $C_g^+(a,b)$. The problem is thus now reduced to automatically inferring bounds from standalone relations. In general, given a standalone cost relation made up of nb base cases of the form $\langle C(\bar{x})=base_j, \varphi_j\rangle$, $1 \leq j \leq nb$ and nr recursive equations of the form, $\langle C(\bar{x})=rec_j + \sum_{i=1}^{k_j} C(\bar{y}_i), \varphi_j\rangle$, $1 \leq j \leq nr$, [7] proposes to compute an upper bound for $C(\bar{x})$, denoted $C(\bar{x})^+$, as follows:

$$(*) \quad C(\bar{x})^+ = \mathsf{l_b} * worst(\{base_1, \ldots, base_{nb}\}) + \mathsf{l_r} * worst(\{rec_1, \ldots, rec_{nr}\})$$

where $\mathsf{l_b}$ and $\mathsf{l_r}$ are, respectively, upper bounds of the number of visits to the base cases and recursive equations and $worst(\{Set\})$ denotes the worst-case (the maximum) value that the expressions in Set can take. The first challenge is thus to have an automatic method to compute $\mathsf{l_b}$ and $\mathsf{l_r}$. In [7], we have proposed the use of *ranking functions* to automatically find $\mathsf{l_b}$ and $\mathsf{l_r}$. Intuitively, a ranking function [30] is a function on the variables of a relation which (1) is positive and (2) decreases in each iteration. For our example, a ranking function is $f_{C_m}(i,n) = n - i$ since the constraints of the corresponding equations imply the previous two conditions, i.e., $\varphi_2 \models f(i,n) > f(i',n) \wedge f(i,n) > 0$ and $\varphi_3 \models f(i,n) > f(i',n) \wedge f(i,n) > 0$.

The next challenge is to have an automatic method to compute the maximum value $worst(\{Set\})$. Following [6], in order to obtain $worst(\{Set\})$, first we need to infer *invariants* between the equation's variables and their initial values. For example, the cost relation $C_m(i,n)$ admits as invariant for the recursive equations the formula $\mathcal{I}$ defined as $\mathcal{I}((i_0,n_0),(i,n)) \equiv i \geq i_0 \wedge n \leq n_0 \wedge i < n$, which captures that the values of i (resp. n) are greater (resp. smaller) or equal than the initial value and that i is smaller than n at all iterations. Once we have the invariant, we can *maximize* the expressions w.r.t. these values and take the maximal: $worst(\{rec_1, \ldots, rec_{nr}\}) = \max(maximize(\mathcal{I}, \{rec_1, \ldots, rec_{nr}\}))$ (see [6] for the technical details of the maximization procedure). For instance, for our cost relation, we compute:

$$worst(\{rec_1, rec_2\}) = \max(maximize(\mathcal{I}, \{\mathsf{nat}(n-i), \mathsf{nat}(i)\}))$$

which results in $worst(\{rec_1, rec_2\}) = \max(\{\mathsf{nat}(n_0 - i_0), \mathsf{nat}(n_0 - 1)\})$. The same procedure can be applied to the expressions in the base cases. However, it is unnecessary in our example, because the base case is a constant. By applying

the same reasoning to the standalone cost relation above, we obtain the following upper bound (on the number of instructions):

$$C_m^+ = 6 + \mathsf{nat}(n-i) * \max(\{21 + 5 * \mathsf{nat}(n-1), 19 + 5 * \mathsf{nat}(n-i)\})$$

The framework could be adapted to infer closed-form lower bounds by using minimization instead of maximization (i.e., using min instead of max when defining *best*, the counterpart of *worst*). However, taking always the best case cost of all iterations would lead to a too pessimistic lower bound, indeed, the obtained lower bound would be in most cases zero. In [16], we propose a refined method in which, instead of assuming the best-case cost for all iterations, we infer tighter bounds on each of them in an automatic way and then approximate the summation of the sequence.

4 Modularity and Incrementality

Typically, cost analysis performs a global analysis of the program which requires that all reachable code is available. Such traditional analysis scheme in which all code is analyzed from scratch, and no previous analysis information is available, is unsatisfactory in many situations. For a practical uptake of cost analysis, it is thus required to reach some degree of compositionallity which allows decomposing the analysis of large programs into the analysis of smaller parts.

In [31], a *modular* approach for the particular case of termination analysis is presented, which allows reasoning on a method at a time. This approach is generalized in [13], where an *incremental* resource analysis scheme is presented. The aim of incremental global analysis is, given a program, its analysis results and a series of changes to the program, to obtain the new analysis results as efficiently as possible and, ideally, without having to re-analyze fragments of code which are not affected by the changes. Incremental analysis can significantly reduce both the time and the memory requirements of analysis.

5 Memory Consumption Analysis for Garbage-Collected Languages

Predicting the memory required to run a program is crucial in many contexts like in embedded applications with stringent space requirements or in real-time systems which must respond to events or signals within a predefined amount of time. When the considered resource is *cumulative*, the approach to cost analysis described so far is parametric w.r.t. the notion of *cost model*, which defines the type of resource we are measuring and, which gives a cost unit to each instruction. It can therefore be directly applied for inferring bounds on the *total* memory usage of programs by devising a cost model which assigns the corresponding cost only to those instructions that consume memory [14].

However, in presence of *garbage collection*, memory is not a cumulative resource but rather it can increase or decrease at any point of the execution.

Thus, the above approach, though still correct, can produce too pessimistic estimations. *Peak memory analysis* [36,19,20], also known as *live memory analysis*, aims at approximating the maximum memory usage during a program's execution, which provides a much tighter estimation. Whereas analyzing the total memory consumption needs to observe the consumption at the *final* state only, peak memory analysis has to reason on the memory consumption at *all program states* along the execution, hence making the analysis much more complicated. In [15], we present a peak memory analysis which is parametric w.r.t the *lifetime* of objects and which can therefore be instantiated with different garbage collection strategies. This requires a non-standard type of recurrence equations.

Example 2. Let us consider method m of Figure. 1, and let us assume that method g only creates an array of 10 integers. Our memory consumption analysis obtains an upper bound of $40{*}\mathsf{nat}(n{-}i)$ bytes for the total memory usage of method m (assuming that the size of an integer is 4 bytes). Let us now consider a *scope-based* garbage collection, i.e., one that reclaims the local memory created in a method call when it finishes. In this case, our memory consumption analysis obtains a constant upper bound of 40 bytes for the peak memory usage of method m. This intuitively corresponds to the maximum between the peak usage of method g (40 bytes) and what *escapes* (i.e. what the garbage collection cannot collect) from g plus the consumption after the call to g (which is 0 bytes).

6 Concurrency in Cost Analysis

Distribution and concurrency are now mainstream, as the availability of multiprocessors radically influences software. This brings renewed interest in developing techniques that help in understanding, analyzing, and verifying the behavior of concurrent programs. This includes the resource consumption behavior.

In concurrent and distributed settings, the meaning of a resource is not limited to traditional measures such as memory consumption or number of executed instructions, but it rather extends to other measures such as the tasks spawned [11], the number of requests to remote servers, etc. In addition, the consumption is not anymore associated to a single execution entity, but rather is distributed over the different components of the system (servers, CPUs, etc). This opens up new interesting applications for resource guarantees, for example, upper bounds can be used to predict that one component may be overloaded, while other siblings are idle most of the time; and lower bounds can be used to decide if it is worth executing locally a task or requesting remote execution.

In [4], we extended the underlying techniques used in the analysis for a concurrency model based on the notion of concurrently running (groups of) objects, in the spirit of the actor-based and active-objects approaches [34,35]. These models aim at taking advantage of the inherent concurrency implicit in the notion of object in order to provide programmers with high-level concurrency constructs that help in producing concurrent applications more modularly and in a less error-prone way. The main challenges we had to deal with are: (1) developing a

size analysis that is able to infer sound size relations taking into account all possible interleavings between the different tasks. The main problem is that when a suspended task is resumed, it cannot assume that the global (shared) state has not been changed since the task has been suspended, but rather it should take into account that it has been possibly modified by other tasks; and (2) incorporating the idea of distributing the cost over the system components in the cost relations.

For (1), we have developed a size analysis that is based on the idea of tracking the global state when it is possible, and using class invariants at points in which interleaving might occur, in order to describe how the global state has been modified. In many cases we are able to automatically infer these invariants, and in some others the user has to provide them. For (2), we have introduced a new kind of cost relations which extend those standard ones with explicit cost centers that represent the system components on which the cost should be distributed. Example of such centers are classes, objects, and groups of objects.

7 Certified Resource Bounds

Resource guarantees, like any other program property which is automatically inferred by static analysis tools, are generally not considered to be completely trustworthy, unless the correctness of the tools or the results have been formally verified. In the case of static analyzers, verifying the correctness of the tools is a daunting task, among other things, because of the sophisticated algorithms used for the analysis and their evolution over time. Thus, in COSTA we take an alternative approach [18,12] in which we verify the results after each run using KeY, a state-of-the-art theorem prover for Java programs. This approach, which apart from being simpler, has the advantage that the proof generated by the theorem prover can then be translated to independently checkable *certificates* in the proof-carrying code style [29].

The certification component of COSTA is based on verifying that all intermediate results produced by COSTA are correct. Then, the correctness of composing them into bounds is straightforward. The intermediate results need to be verified include: (1) *ranking functions*, which are used to bound the number of iterations of each loop; (2) *loop invariants*, which provide an insight on the values that each variable can take, and relations between them; (3) *size relations*; which describe how the size of the data change when moving from one part of the program to another; and (4) *heap properties*, such as depth of data-structures and acyclicity, which are essential for inferring resource guarantees for heap manipulating programs.

8 Conclusions and Future Work

We have described the main techniques used in cost analysis of a today's programming language. Our approach is based on the traditional cost analysis framework which generates recurrence relations from programs and then solves

them to obtain upper and/or lower bounds. There exist other approaches to cost analysis (e.g., [23,24]) which are not based on recurrence relations. It is hence not possible to formally compare the resulting upper bounds in the general case (more details are provided in [16]).

COSTA (available at `http://costa.ls.fi.upm.es`) is a state-of-the-art cost and termination analyzer which implements our approach. The system was originally designed to obtain upper bounds from *bytecode* programs. Analyzing bytecode has a much wider application area than analyzing Java source code since the latter is often not available. The COSTA system can be used online through a web interface and also from its Eclipse plugin. The user can provide assertions which state the expected resource consumption of selected methods by using JML notation. COSTA then tries to verify (or falsify) the assertions [5]. The assertions can also be written in asymptotic form [3]. This facilitates human reasoning, as asymptotic cost expressions are much simpler than the non-asymptotic ones.

Recently, the system has been extended to obtain upper bounds from concurrent ABS programs [26]. The resulting extension is called COSTABS [17]. ABS is an Abstract Behavioral Specification language for distributed object-oriented systems. COSTABS is integrated in the ABS development tools (available at `http://www.hats-project.eu`), which can be used within the Eclipse development environment. Moreover, it can be used through a web-interface[1], and can be downloaded and used through a command-line. In [11], we have applied similar, but simpler, techniques to an async-finish concurrency model.

As regards certifying the upper bounds, we have succeed to make the COSTA and KeY system cooperate as follows: COSTA outputs the intermediate results of the analyzed program by means of extended JML annotations, and then KeY verifies the resulting proof obligations in its program logic and produces proof certificates that can be saved and reloaded. Realizing the above cooperation between COSTA and KeY, and integrating it in the Eclipse development environment, has required a number of non-trivial extensions of both systems.

Our current research is mainly focused on improving the accuracy and efficiency of the analysis. Being more accurate is especially relevant in the context of concurrency as, due to potential tasks interleavings, shared data are currently lost when the processor is released. We are working on automatically generating class invariants to improve the abstraction of share data such that we gain precision when the cost depends on the sizes of such data.

Acknowledgments. This work was funded in part by the Information & Communication Technologies program of the EC, Future and Emerging Technologies (FET), under the ICT-231620 *HATS* project, by the Spanish Ministry of Science and Innovation (MICINN) under the TIN-2008-05624 *DOVES* project, the UCM-BSCH-GR35/10-A-910502 *GPD* Research Group and by the Madrid Regional Government under the S2009TIC-1465 *PROMETIDOS-CM* project.

[1] `http://costa.ls.fi.upm.es/costabs`

References

1. Aho, A.V., Hopcroft, J.E., Ullman, J.D.: The Design and Analysis of Computer Algorithms. Addison-Wesley (1974)
2. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers – Principles, Techniques and Tools. Addison-Wesley (1986)
3. Albert, E., Alonso, D., Arenas, P., Genaim, S., Puebla, G.: Asymptotic Resource Usage Bounds. In: Hu, Z. (ed.) APLAS 2009. LNCS, vol. 5904, pp. 294–310. Springer, Heidelberg (2009)
4. Albert, E., Arenas, P., Genaim, S., Gómez-Zamalloa, M., Puebla, G.: Cost Analysis of Concurrent OO Programs. In: Yang, H. (ed.) APLAS 2011. LNCS, vol. 7078, pp. 238–254. Springer, Heidelberg (2011)
5. Albert, E., Arenas, P., Genaim, S., Herraiz, I., Puebla, G.: Comparing Cost Functions in Resource Analysis. In: van Eekelen, M., Shkaravska, O. (eds.) FOPARA 2009. LNCS, vol. 6324, pp. 1–17. Springer, Heidelberg (2010)
6. 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)
7. Albert, E., Arenas, P., Genaim, S., Puebla, G.: Closed-Form Upper Bounds in Static Cost Analysis. Journal of Automated Reasoning 46(2), 161–203 (2011)
8. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost Analysis of Java Bytecode. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 157–172. Springer, Heidelberg (2007)
9. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Removing Useless Variables in Cost Analysis of Java Bytecode. In: ACM Symposium on Applied Computing (SAC) - Software Verification Track (SV 2008), Fortaleza, Brasil, pp. 368–375. ACM Press, New York (2008)
10. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost Analysis of Object-Oriented Bytecode Programs. Theoretical Computer Science 413(1), 142–159 (2012)
11. Albert, E., Arenas, P., Genaim, S., Zanardini, D.: Task-Level Analysis for a Language with Async-Finish parallelism. In: Vitek, J., De Sutter, B. (eds.) Proceedings of the ACM SIGPLAN/SIGBED 2011 Conference on Languages, Compilers, and Tools for Embedded Systems, LCTES 2011, Chicago, IL, USA, April 11-14, pp. 21–30. ACM (2011)
12. Albert, E., Bubel, R., Genaim, S., Hähnle, R., Román-Díez, G.: Verified Resource Guarantees for Heap Manipulating Programs. In: Proceedings of the 15th International Conference on Fundamental Approaches to Software Engineering, FASE 2012, Tallinn, Estonia. Springer, Heidelberg (to appear, 2012)
13. Albert, E., Correas, J., Puebla, G., Román-Díez, G.: Incremental Resource Usage Analysis. In: Proceedings of the 2012 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2012, Philadelphia, Pennsylvania, USA, January 23-24. ACM Press (to appear, 2012)
14. Albert, E., Genaim, S., Gómez-Zamalloa, M.: Heap Space Analysis of Java Bytecode. In: 6th International Symposium on Memory Management (ISMM 2007), pp. 105–116. ACM Press (2007)
15. Albert, E., Genaim, S., Gómez-Zamalloa, M.: Parametric Inference of Memory Requirements for Garbage Collected Languages. In: 9th International Symposium on Memory Management (ISMM 2010), pp. 121–130. ACM Press, New York (2010)

16. Albert, E., Genaim, S., Masud, A.N.: More Precise Yet Widely Applicable Cost Analysis. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 38–53. Springer, Heidelberg (2011)
17. Albert, E., Arenas, P., Genaim, S., Gómez-Zamalloa, M., Puebla, G.: COSTABS: A Cost and Termination Analyzer for ABS. In: Proceedings of the 2012 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2012, Philadelphia, Pennsylvania, USA, January 23-24, ACM Press (to appear, 2012)
18. Albert, E., Bubel, R., Genaim, S., Hähnle, R., Puebla, G., Román-Díez, G.: Verified resource guarantees using costa and key. In: Khoo, S.-C., Siek, J.G. (eds.) PEPM, pp. 73–76. ACM (2011)
19. Braberman, V., Fernández, F., Garbervetsky, D., Yovine, S.: Parametric Prediction of Heap Memory Requirements. In: ISMM. ACM Press (2008)
20. Chin, W.-N., Nguyen, H.H., Popeea, C., Qin, S.: Analysing Memory Resource Bounds for Low-Level Programs. In: ISMM. ACM Press (2008)
21. Debray, S.K., Lin, N.W.: Cost Analysis of Logic Programs. ACM Transactions on Programming Languages and Systems 15(5), 826–875 (1993)
22. Debray, S.K., Lin, N.W.: Cost analysis of logic programs. ACM TOPLAS 15(5), 826–875 (1993)
23. Gulwani, S., Mehra, K.K., Chilimbi, T.M.: SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In: The 36th Symposium on Principles of Programming Languages (POPL 2009), pp. 127–139. ACM (2009)
24. Hoffmann, J., Hofmann, M.: Amortized Resource Analysis with Polynomial Potential. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 287–306. Springer, Heidelberg (2010)
25. Hofmann, M., Hoffmann, J., Aehlig, K.: Multivariate Amortized Resource Analysis. In: The 38th Symposium on Principles of Programming Languages (POPL 2011), pp. 357–370. ACM (2011)
26. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A Core Language for Abstract Behavioral Specification. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 142–164. Springer, Heidelberg (2011)
27. Lee, J.K., Palsberg, J.: Featherweight X10: A Core Calculus for Async-Finish Parallelism. In: Principles and Practice of Parallel Programming (PPoPP 2010), pp. 25–36. ACM, New York (2010)
28. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification. Addison-Wesley (1996)
29. Necula, G.: Proof-Carrying Code. In: ACM Symposium on Principles of programming languages (POPL 1997). ACM Press (1997)
30. Podelski, A., Rybalchenko, A.: A Complete Method for the Synthesis of Linear Ranking Functions. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 239–251. Springer, Heidelberg (2004)
31. Ramírez-Deantes, D., Correas, J., Puebla, G.: Modular Termination Analysis of Java Bytecode and Its Application to phoneME Core Libraries. In: Barbosa, L.S. (ed.) FACS 2010. LNCS, vol. 6921, pp. 218–236. Springer, Heidelberg (2010)
32. Sands, D.: Complexity Analysis for a Lazy Higher-Order Language. In: Jones, N.D. (ed.) ESOP 1990. LNCS, vol. 432, pp. 361–376. Springer, Heidelberg (1990)
33. Sands, D.: A Naïve Time Analysis and its Theory of Cost Equivalence. Journal of Logic and Computation 5(4) (1995)

34. Schäfer, J., Poetzsch-Heffter, A.: JCoBox: Generalizing Active Objects to Concurrent Components. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 275–299. Springer, Heidelberg (2010)
35. Srinivasan, S., Mycroft, A.: Kilim: Isolation-Typed Actors for Java. In: Ryan, M. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 104–128. Springer, Heidelberg (2008)
36. Unnikrishnan, L., Stoller, S.D., Liu, Y.A.: Optimized Live Heap Bound Analysis. In: Zuck, L.D., Attie, P.C., Cortesi, A., Mukhopadhyay, S. (eds.) VMCAI 2003. LNCS, vol. 2575, pp. 70–85. Springer, Heidelberg (2002)
37. Wegbreit, B.: Mechanical Program Analysis. Communications of the ACM 18(9) (1975)

Matrix Interpretations for Polynomial Derivational Complexity of Rewrite Systems

Aart Middeldorp

Institute of Computer Science, University of Innsbruck, Austria

Rewrite systems form an attractive model of computation. In the past decades numerous methods have been developed to prove rewrite systems terminating. Spurred by the International Termination Competition, the emphasis in recent years is on powerful methods that can be automated.

Termination is a prerequisite, but to ensure that normal forms can be effectively computed, one needs termination methods from which a polynomial upper bound on the lengths of computations can be inferred. Until recently, very few results of this kind were known: If termination of a rewrite system can be established by a strongly linear interpretation [3] or by the match-bound technique [2], there is a linear upper bound on the derivational complexity. The recent matrix method [1] radically changed the picture. Using results from linear algebra (joint spectral radius theory) and weighted automata (degrees of ambiguity), conditions on matrix interpretation can be formulated to ensure a polynomial upper bound on the derivational complexity [4–7]. These conditions can be translated into finite-domain constraint systems and solved by state-of-the-art SAT/SMT solvers.

In the talk, which is based on joint work with Georg Moser, Friedrich Neurauter, Johannes Waldmann and Harald Zankl, we summarize the known results and report on ongoing research.

References

1. Endrullis, J., Waldmann, J., Zantema, H.: Matrix interpretations for proving termination of term rewriting. JAR 40(2-3), 195–220 (2008)
2. Geser, A., Hofbauer, D., Waldmann, J., Zantema, H.: On tree automata that certify termination of left-linear term rewriting systems. I&C 205(4), 512–534 (2007)
3. Hofbauer, D., Lautemann, C.: Termination Proofs and the Length of Derivations (Preliminary Version). In: Dershowitz, N. (ed.) RTA 1989. LNCS, vol. 355, pp. 167–177. Springer, Heidelberg (1989)
4. Middeldorp, A., Moser, G., Neurauter, F., Waldmann, J., Zankl, H.: Joint Spectral Radius Theory for Automated Complexity Analysis of Rewrite Systems. In: Winkler, F. (ed.) CAI 2011. LNCS, vol. 6742, pp. 1–20. Springer, Heidelberg (2011)
5. Moser, G., Schnabl, A., Waldmann, J.: Complexity Analysis of Term Rewriting Based on Matrix and Context Dependent Interpretations. In: Hariharan, R., Mukund, M., Vinay, V. (eds.) FSTTCS. LIPIcs, vol. 2, pp. 304–315 (2008)
6. Neurauter, F., Zankl, H., Middeldorp, A.: Revisiting Matrix Interpretations for Polynomial Derivational Complexity of Term Rewriting. In: Fermüller, C.G., Voronkov, A. (eds.) LPAR-17. LNCS, vol. 6397, pp. 550–564. Springer, Heidelberg (2010)
7. Waldmann, J.: Polynomially Bounded Matrix Interpretations. In: Lynch, C. (ed.) RTA. LIPIcs, vol. 6, pp. 357–372 (2010)

N. Bjørner and A. Voronkov (Eds.): LPAR-18, LNCS 7180, p. 12, 2012.

Parameterized Complexity and Fixed-Parameter Tractability of Description Logic Reasoning

Boris Motik

Department of Computer Science, University of Oxford
Wolfson Building, Parks Road, OX1 3QD, Oxford

An important goal of research in description logics (DLs) and related logic-based KR formalisms is to identify the worst-case complexity of reasoning. Such results, however, measure the complexity of a logic *as a whole*. For example, reasoning in the basic DL $\mathcal{ALCI}$ is ExpTime-complete, which means that $\mathcal{ALCI}$ constructors can be used in a way so that exponential time is strictly required for solving a reasoning problem. It is, however, well known that, given two $\mathcal{ALCI}$ knowledge bases of roughly the same size, reasoning with one knowledge base may be much more difficult than with the other, depending on the interaction of the axioms in the KBs. Thus, existing worst-case complexity results provide only a very coarse measure of reasoning complexity, and they do not tell us much about the "hardness" of each individual knowledge base.

Parameterized complexity [2] provides us with a framework for a more fine-grained analysis of the difficulty of reasoning. The general idea is to measure the "hardness" of a problem instance of size n using a nonnegative integer *parameter* k, and the goal is to solve the problem in time that becomes polynomial in n whenever k is fixed. A particular goal is to identify *fixed parameter tractable* (FPT) problems, which can be solved in time $f(k) \cdot n^c$, where c is a constant and f is an arbitrary computable function that depends *only* on k.

Each problem is clearly in FPT if the parameter is the problem's size, so a useful parameterization should allow increasing the size arbitrarily while keeping the parameter bounded. Various problems in AI were successfully parameterized using the graph-theoretic notions of *tree decompositions* and *treewidth* [3–5], and many FPT results have been obtained using the Courcelle's Theorem [1]. Applying these ideas to formalisms such as description and modal logics seems difficult: due to existential and universal quantifiers, solving a reasoning problem may require exploring very large structures.

In my talk I will present an overview of parameterized complexity, fixed-parameter tractability, and treewidth, and I will briefly discuss how these notions can be used to obtain FPT results for formalisms such as propositional logic and answer set programming. Furthermore, I will discuss the difficulties in applying these ideas to logics with existential quantifiers, such as DLs. I will then present a particular parameterization for DL knowledge bases. This result is based on a novel notion of a *decomposition*—a structure inspired by tree decompositions, but extended in a way that captures the effects of quantifiers. I will also discuss

N. Bjørner and A. Voronkov (Eds.): LPAR-18, LNCS 7180, pp. 13–14, 2012.

a fundamental tradeoff between decomposition *width* and *length*—the two parameters that characterize the difficulty of DL reasoning. Finally, I will present what I believe to be the first result FPT result for DL reasoning.

References

1. Courcelle, B.: The monadic second-order logic of graphs. I. recognizable sets of finite graphs. Information and Computation 85, 12–75 (1990)
2. Downey, R.G., Fellows, M.R.: Parameterized Complexity. Springer, Heidelberg (1999)
3. Gottlob, G., Pichler, R., Wei, F.: Bounded Treewidth as a Key to Tractability of Knowledge Representation and Reasoning. In: Proc. AAAI, pp. 250–256 (2006)
4. Gottlob, G., Scarcello, F., Sideri, M.: Fixed-parameter complexity in AI and nonmonotonic reasoning. Artificial Intelligence 138(1-2), 55–86 (2002)
5. Szeider, S.: On Fixed-Parameter Tractable Parameterizations of SAT. In: Proc. SAT, pp. 188–202 (2003)

Enfragmo: A System for Modelling and Solving Search Problems with Logic

Amir Aavani, Xiongnan (Newman) Wu, Shahab Tasharrofi, Eugenia Ternovska, and David Mitchell

Simon Fraser University
{aaa78,xwa33,sta44,ter,mitchell}@cs.sfu.ca

Abstract. In this paper, we present the Enfragmo system for specifying and solving combinatorial search problems. It supports natural specification of problems by providing users with a rich language, based on an extension of first order logic. Enfragmo takes as input a problem specification and a problem instance and produces a propositional CNF formula representing solutions to the instance, which is sent to a SAT solver. Because the specification language is high level, Enfragmo provides combinatorial problem solving capability to users without expertise in use of SAT solvers or algorithms for solving combinatorial problems. Here, we describe the specification language and implementation of Enfragmo, and give experimental evidence that its performance is comparable to that of related systems.

1 Introduction

Computationally hard search and optimization problems are ubiquitous in science, engineering and business. Examples include drug design, protein folding, phylogeny reconstruction, hardware and software design, test generation and verification, planning, timetabling, scheduling and so on. In rare cases, practical application-specific software exists, but most often development of successful solution methods requires specialists to apply technology such as mathematical programming of constraint logic programming systems, or develop custom-refined implementations of general algorithms, such as branch and bound, simulated annealing, or reduction to SAT.

One goal of development of the Enfragmo system [1] is to provide another practical technology for solving combinatorial search problems, but one which would require considerably less specialized expertise on the part of the user, thus making technology for solving these problems accessible to a wider variety of users. In this approach, the user gives a precise specification of their search (or optimization) problem in a high-level declarative modelling language. A solver then takes this specification, together with an instance of the problem, and produces a solution to the problem (if there is one).

A natural formalization of search problems in general is as model expansion (MX) [2], which is the logical task of expanding a given structure by new relations. Formally, users axiomatize their problems, formalized as model expansion, in some extension of classical logic. A problem instance, in this formalization, is a finite structure, and solutions to the instance are expansions of this structure that satisfy the specification

N. Bjørner and A. Voronkov (Eds.): LPAR-18, LNCS 7180, pp. 15–22, 2012.

formula. At present, our focus is on problems in the complexity class NP. For this case, the specification language is based on classical first-order logic (FO). Fagin's theorem [3] states that the problems which can be axiomatized in the existential fragment of second order logic ($\exists$SO) are exactly those in NP, and thus the problems which can be axiomatized as FO MX are exactly the NP search problems.

Enframgo's operation is based on grounding, which is the task of producing a variable-free first-order formula representing the expansions of the instance structure which satisfy the specification formula – in other words, the solutions for the instance. The ground formula is mapped to a propositional CNF formula, which is sent to a SAT solver. For any fixed FO formula, grounding can be carried out in polynomial time, so grounding provides a universal polytime reduction to SAT for problems in NP. (Developing grounding to other languages, for example to use SMT solvers, is a promising direction which we have begun exploring.) An important advantage in solving through grounding and transformation to SAT, or other standard ground language, is that the performance of ground solvers is constantly being improved, and we can always select from the best solvers available.

Many interesting real-world problems cannot be conveniently expressed in pure FO MX, in particular if their natural descriptions involve arithmetic or inductive properties. Examples of the former include Knapsack and other problems involving weights or costs, while examples of the latter include the Traveling Salesman problem and other problems involving reachability. To address these issues, Enfragmo's specification language includes a limited use of inductive definitions, and extends classical first order logic with arithmetic and aggregate operators. The authors of [2] emphasized the importance of having inductive definitions in a specification language. In the case of arithmetic, care must be exercised to avoid increasing the expressive power of the language beyond NP, which would prevent polytime grounding to SAT. A theoretical investigation of the issues involved appears in [4,5,6]. Extension of the basic grounding algorithm of Enfragmo to arithmetic terms, including aggregate operators, is described in [7].

2 Specification Language

The specification language of the Enfragmo system is based on multi-sorted classical first-order logic extended with inductive definitions, arithmetic functions, and aggregate operators. We will illustrate the language with two examples, and give brief discussions of some major features. The examples use the actual ASCII input representation. The mapping from logical symbols to ASCII symbols used is given in Table 1. For the full description of the input language, please refer to the Enfragmo manual, which is available from [1].

An Enfragmo specification consists of four main sections, delineated by keywords. The `GIVEN:` section defines the types and vocabulary used in the specification. The `FIND:` section identifies the vocabulary symbols for which the solver must find interpretations, that is, the functions and relations which will constitute a solution. Interpretations of the remaining vocabulary symbols are given by the problem instance. The third part consists of one or more `PHASE:` sections, each of which contains an optional

Table 1. ASCII Equivalents for Logical Symbols

Logical Symbol	$\forall$	$\exists$	$\wedge$	$\vee$	$\neg$	$\rightarrow$	$\leftrightarrow$
ASCII Representation	!	?	&	|	~	=>	<=>

```
GIVEN:
   TYPES: Vtx Clr;
   PREDICATES: Edge(Vtx,Vtx), Colour(Vtx,Clr);
FIND: Colour;
   // expansion predicate(s) are listed under FIND
   //(instance predicates are those that are not expansion)
PHASE:
 SATISFYING:
   // every vertex has at least one colour
   !v:Vtx : ?c:Clr : Colour(v,c);
   // no vertex has more than one colour
   !v:Vtx c:Clr : Colour(v,c) => ~?c2:Clr < c : Colour(v,c2);
   // no two vertices of the same colour are adjacent
   !u:Vtx v:Vtx c:Clr : Colour(u,c) & Colour(v,c)=> ~Edge(u,v);
PRINT: Colour; // solution can be printed
```

Fig. 1. Enfragmo specification of K-colouring

`FIXPOINT:` part, which provided an inductive definition, followed by a `SATISFYING:` part, which consists of a set of sentences in the extended first order logic. If there are multiple `PHASE:` sections, the define a sequence of expansions. One way such a sequence can be used is to carry out a kind of pre-processing or post-processing, which may support more convenient axiomatizations or more efficient solving. An example is provided in the section on inductive definitions below. Finally, the `PRINT:` section identifies relations that are to be displayed, if a solution is found.

Example 1 (Graph K-Colouring). Graph colouring is a classic and well-studied NP-hard search problem. The task is to colour vertices of a given graph using colours from a given set of K colours, so that no two adjacent vertices have the same colour. To axiomatize this problem, we introduce two sorts, vertices and colours. The axiomatization says that there is a binary relation Colour which must be a proper colouring of the vertices. The corresponding Enfragmo specification is given in Figure 1.

Arithmetic and Aggregates. Enfragmo specifications have two kinds of types: integer types and enumerated types. Terms of integer types may use the arithmetic functions $+$, $-$, $*$, and $ABS(\cdot)$, which have their standard meaning for the integers. Arithmetic terms also include the aggregate operators maximum, minimum, sum, and cardinality. In the following, if $\phi(\bar{x})$ is a formula with free variables $\bar{x}$, then $\phi^{\mathcal{B}}[\bar{a}]$ denotes the truth value of ϕ in structure $\mathcal{B}$ when the variables $\bar{x}$ denote the domain elements $\bar{z}$, and similarly for terms $t(\bar{x})$. The aggregate terms are defined, as follows, with respect to a structure $\mathcal{B}$ in which the formula containing the term is true.

$Max_{\bar{x}}\{t(\bar{x},\bar{y}) : \phi(\bar{x},\bar{y}); d_M\}$ denotes, for any instantiation $\bar{b}$ for $\bar{y}$, the maximum value obtained by $t^{\mathcal{B}}[\bar{a},\bar{b}]$ over instantiations $\bar{a}$ for $\bar{x}$ for which $\phi^{\mathcal{B}}[\bar{a},\bar{b}]$ is true, or d_M (the default value) if there are none.
$Min_{\bar{x}}\{t(\bar{x},\bar{y}) : \phi(\bar{x},\bar{y}); d_m\}$ is defined dually to Max.
$Sum_{\bar{x}}\{t(\bar{x},\bar{y}) : \phi(\bar{x},\bar{y})\}$, denotes, for any instantiation $\bar{b}$ for $\bar{y}$, the sum of all values $t^{\mathcal{B}}[\bar{a},\bar{b}]$ over instantiations $\bar{a}$ for $\bar{x}$ for which $\phi^{\mathcal{B}}[\bar{a},\bar{b}]$ is true.
$Count_{\bar{x}}\{\phi(\bar{x},\bar{y})\}$ denotes, for any instantiation $\bar{b}$ for $\bar{y}$, the number of tuples $\bar{a}$ for which $\phi^{\mathcal{B}}[\bar{a},\bar{b}]$ is true.

Example 2 illustrates use of arithmetic terms, including sum and count aggregates.

Example 2 (A Knapsack Problem Variant). Consider the following variation of the knapsack problem: We are given a set of items (loads) $L = \{l_1, \cdots, l_n\}$, each with an integer weight $W(l_i)$, and m knapsacks $K = \{k_1, \cdots, k_m\}$. The task is to put the n items into the m knapsacks, while satisfying the following constraints. 1) Certain items must go into preassigned knapsacks, as specified by the binary instance predicate P; 2) H of the m knapsacks are high capacity, and can hold items with total weight Cap_H, while the remainder have capacity Cap_L; 3) No knapsack may contain two items with weights that differ by more than D. Each of Cap_H, Cap_L and D is an instance function with arity zero, i.e. a given constant. An Enfragmo specification for this problem is given in Figure 2. Q is the mapping of items to knapsacks that must be constructed.

Inductive Definitions. Enfragmo supports a limited use of inductive definitions. In the current implementation, the open predicates [2] in a definition must be instance predicates, or have been constructed explicitly in a previous `Phase:` section. Even this limited form has proved to be very useful in practice. These definitions can be used to efficiently compute useful information, such as a bound or partial solution, which can be used later to help solve a problem more efficiently. For example, Graph Colouring can be solved more efficiently by having inductive definitions in an initial group of phases compute a maximal clique in the graph, and pre-assign distinct colours to the vertices in that clique. (A further improvement might be to construct a maximal clique containing a vertex of maximum degree.) Then a final phase can construct a colouring of the graph restricted by the pre-computed colouring of the large clique. Use of these inductive definitions can also support writing of natural axiomatizations, by allowing the introduction of defined terms without incurring a performance penalty.

3 Implementation

Enfragmo's input is a problem specification, stated in the language described in Section 2, together with a description of an instance. (The syntax for specifying instances is described in the manual. Eventually, instances may be retrieved by queries to a database or other source.) The phases in the specification are solved one-by-one, in the order written. For each phase, any predicates defined by an inductive definition are computed, and then the satisfying phase is solved by grounding. This in turn involves three stages: 1) grounding each formula with respect to the instance to produce a ground FO formula

```
GIVEN:
   TYPES: Item Knaps;
   INTTYPES: Weight ItemCount;
   PREDICATES: P(Item, Knaps) Q(Item, Knaps);
   FUNCTIONS:
     W(Item): Weight
     Cap_H(): Weight
     Cap_L(): Weight
     D(): Weight
     H(): ItemCount;
FIND: Q;
PHASE:
 SATISFYING:
   // Q is a function mapping items to knapsacks
   !l:Item : ?k:Knaps : Q(l, k);
   !l:Item k1:Knaps k2:Knaps : Q(l, k1) & Q(l, k2) => k1 = k2;
   // Q agrees with the pre-assignment P
   !l:Item k:Knaps : P(l, k) => Q(l, k);
   // The total weight in each knapsack is at most Cap_H
   !k:Knaps: SUM{l:Item; W(l); Q(l, k)} <= Cap_H();
   // At most H knapsacks have total weight greater than Cap_L
   COUNT{k:Knaps; SUM{l:Item; W(l); Q(l, k)} > Cap_L()} <= H();
   // Items in a knapsack differ in weight by at most D
   !k:Knaps l1:Item l2:Item : Q(l1, k) & Q(l2, k)
                              => ABS(W(l1) - W(l2)) <= D();
PRINT: Q;
```

Fig. 2. Enfragmo specification for Knapsack variant

representing the solutions; 2) the ground formula is transformed to a propositional CNF formula; 3) a SAT solver is called on the CNF formula; 4) if the SAT solver reports a satisfying assignment, it is mapped back to a description of a solution in the vocabulary of the specification.

Grounding. Enfragmo computes a grounding of a formula bottom up, in a process analogous to bottom up evaluation of a database query using the relational algebra. Here, an extension of the relational algebra is used, in which an formula is associated with each tuple. A tuple contains domain elements, and the associated formula is (equivalent to) a ground instance of a sub-formula of the specification formula, with variables instantiated by constants denoting the domain elements in the tuple. An "answer" to a sub-formula of the specification formula is an extended table representing all instantiations of the sub-formula. Similarly, and answer to a terms is a set of triples, each consisting of an instantiation of the arguments, a value the term may denote, and a formula. Details can be found in [7] and [8]. The answer for a sentence consists of an empty tuple associated with a formula that is a grounding of the sentence with respect to the instance.

Efficiency of this grounding method requires using suitable data structures. For efficiency, all formulas in computed answers are represented in a dag which is constructed as the operations of the algebra are applied. Next we briefly describe some aspects of the implementation of tables representing answers.

In the tables representing answers to formulas, it is natural have non-existence of a tuple correspond to associating the formula False with the tuple. However, negating a sparse table with this convention produces a very dense table in which many tuples are associated with the formula True. To help keep tables sparse, we employ two kinds of tables, one in which absence of a tuple corresponds to False, and one where it corresponds to True. Details on design and performance of True/False tables are given in [8]. It is often the case that, in an answer for a sub-formula, the instantiated formulas are independent of the instantiations of some of the free variables. In this case, the table explicitly records only the partial instantiations that are needed. This method is described in [9] and [8] as tables with "hidden variables".

A simple term is a term whose denotation can be computed (with respect to an instance), just using the assignments to its free variables. For example, $W(l)$ in Example 2 is a simple term. The formula for each tuple in an answer to a simple term is either True or False. A term which is not simple is called complex. The Count aggregate used in Example 2 is a complex term, because its value depends on the expansion predicate Q (the solution). To represent the answer to a complex term occurring as an argument to sub-formula ϕ, we have two data-structures: 1) A hash-map which maps each value o which term t may denote to a table which is an answer to the formula $t(\bar{x}) = o$, and 2) A table which can be viewed as being the answer to the formula $\phi(\bar{x}, y) : t(\bar{x}) = y$. Methods for efficiently constructing the answers to complex terms, such as terms containing nested count and sum aggregates, are examined in [7] and [10].

CNF Transformation. The set of answers for the sentences of a specification are then transformed to a propositional CNF formula. As usual, this is done using a refinement of Tseitin's polytime transformation to CNF [11]. Refinements include re-writing the formula into negation normal form, so negations occur only on atoms; flattening nested conjunctions and disjunctions; and in some cases merging of identical sub-formulas.

4 Experimental Evaluation

In this section, we compare the performance of Enfragmo to other grounding-based systems. A set of NP-hard of problems were chosen from [12]. We excluded problems for which all instances in the collection are easy. We also excluded problems where the sum aggregate is central, as the current implementation of sum in Enfragmo is preliminary and does not perform well. (We are currently studying better methods for sum. For example, see [13]). The other solvers are Clingo (v 3.0.3) [14], DLV (v 2010-10-14) [15], and IDP (v 2.20) [16]. For each system, we used specifications provided by the system authors, obtained from [12]. The experiments were run on an Intel Xeon L5420 quad-core 2.5 GHz processor, with a timeout of 600 seconds. All specifications, instances, and scripts used for the experiments can be downloaded from [1]. The results

are given in Table 2. The entry n/t indicates that n instances were solved, each within the 600 second timeout, in a total time of t seconds. The time t includes the time for all the runs that timed out.

Table 2. Performance comparison of Enfragmo and other systems. The entry n/t means that n instances were solved in total time of t seconds. The 600-second timeouts are included in the times. The best result for each problem is in bold.

Problem	# of Instances	Clingo	DLV	IDP	Enfragmo
GraphColoring	29	9/12400	8/13398	9/12199	**27/6965**
HamiltonianPath	29	**29/1.6**	20/6856	29/2.1	29/308
SchurNumbers	29	29/889	18/8273	28/1452	**29/643**
BlockedNQueens	29	**29/165**	28/9870	29/896	29/1278
ConnectedDominatingSet	20	**20/969**	13/6190	17/3258	19/2038
DisjunctiveScheduling	10	10/1174	5/3581	10/1008	**10/421**
Total	146	126/15601	92/48170	122/18818	**143/11656**

Table 2 shows that Enfragmo was able to solve almost all the instances in the collection, and performed the best on three of the six problems. Enfragmo also performed the best by the aggregate measures of total number of instances solved and total time spent.

5 Conclusion

We presented the Enfragmo system for modelling and solving combinatorial search problems. It provides users with a convenient way to specify and solve computationally hard problems, in particular search problems whose decision versions are in the complexity class NP. The performance of the Enfragmo system is comparable to that of related systems.

References

1. http://www.cs.sfu.ca/research/groups/mxp/
2. Mitchell, D., Ternovska, E.: A framework for representing and solving NP search problems. In: Proc. AAAI, pp. 430–435 (2005)
3. Fagin, R.: Generalized first-order spectra and polynomial-time recognizable sets. Complexity of Computation, 43–74 (1974)
4. Ternovska, E., Mitchell, D.: Declarative programming of search problems with built-in arithmetic. In: Proc. of IJCAI, pp. 942–947 (2009)
5. Tasharrofi, S., Ternovska, E.: Built-in arithmetic in knowledge representation languages. In: NonMon at 30 (Thirty Years of Nonmonotonic Reasoning) (October 2010)

6. Tasharrofi, S., Ternovska, E.: PBINT, A Logic for Modelling Search Problems Involving Arithmetic. In: Fermüller, C.G., Voronkov, A. (eds.) LPAR-17. LNCS, vol. 6397, pp. 610–624. Springer, Heidelberg (2010)
7. Aavani, A., Wu, X(N.), Ternovska, E., Mitchell, D.: Grounding Formulas with Complex Terms. In: Butz, C., Lingras, P. (eds.) Canadian AI 2011. LNCS, vol. 6657, pp. 13–25. Springer, Heidelberg (2011)
8. Aavani, A., Tasharrofi, S., Unel, G., Ternovska, E., Mitchell, D.: Speed-Up Techniques for Negation in Grounding. In: Clarke, E.M., Voronkov, A. (eds.) LPAR-16 2010. LNCS, vol. 6355, pp. 13–26. Springer, Heidelberg (2010)
9. Mohebali, R.: A method for solving NP search problems based on model expansion and grounding. Master's thesis, Simon Fraser University (2006)
10. Aavani, A., Wu, X., Mitchell, D., Ternovska, E.: Grounding Cardinality Constraints. In: LPAR-16 Short Paper (2010)
11. Tseitin, G.S.: On the complexity of derivations in the propositional calculus. Studies in Mathematics and Mathematical Logic, 115–125 (1968)
12. Denecker, M., Vennekens, J., Bond, S., Gebser, M., Truszczyński, M.: The Second Answer Set Programming Competition. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR 2009. LNCS, vol. 5753, pp. 637–654. Springer, Heidelberg (2009)
13. Aavani, A.: Translating Pseudo-Boolean Constraints into CNF. In: Sakallah, K.A., Simon, L. (eds.) SAT 2011. LNCS, vol. 6695, pp. 357–359. Springer, Heidelberg (2011)
14. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Schneider, M.: Potassco: The Potsdam answer set solving collection. AI Commun. 24(2), 105–124 (2011)
15. Dell'Armi, T., Faber, W., Ielpa, G., Koch, C., Leone, N., Perri, S., Pfeifer, G.: System Description: DLV. In: Eiter, T., Faber, W., Truszczyński, M. (eds.) LPNMR 2001. LNCS (LNAI), vol. 2173, pp. 424–428. Springer, Heidelberg (2001)
16. Wittocx, J., Marién, M., Denecker, M.: The IDP system: A model expansion system for an extension of classical logic. In: Proceedings of the 2nd Workshop on Logic and Search, pp. 153–165 (2008)

The Permutative λ-Calculus

Beniamino Accattoli[1] and Delia Kesner[2]

[1] INRIA and LIX, École Polytechnique
[2] PPS, CNRS and Université Paris-Diderot

Abstract. We introduce the permutative λ-calculus, an extension of λ-calculus with three equations and one reduction rule for permuting constructors, generalising many calculi in the literature, in particular Regnier's sigma-equivalence and Moggi's assoc-equivalence. We prove confluence modulo the equations and preservation of beta-strong normalisation (PSN) by means of an auxiliary substitution calculus. The proof of confluence relies on M-developments, a new notion of development for λ-terms.

1 Introduction

Background. The standard operational semantics of λ-calculus is given by β-reduction. However, this unique notion of reduction is often extended with some other rewriting rules allowing to permute constructors. This arises in different contexts and comes with many different motivations. A typical example is the postponement of *erasing* steps, which is obtained by introducing one particular such permutation rule [5]. Four other notable motivations for introducing permutations are: making redexes more visible [10], analysing the relation between λ-terms and Proof-Nets [17], proving the completeness of CPS-translation for the call-by-value λ-calculus [18], translating Moggi's monadic metalanguage into λ-calculus [21]. The rewriting theory of these permutation rules is often tricky, in particular when proving strong normalisation or preservation of strong normalisation (PSN) [12,4,14,6,7]. This is indeed the major and usually difficult question arising in all these extensions: to prove that if t is a β-strongly-normalising λ-term then t is also strongly-normalising with respect to the extended reduction relation.

The Permutative λ-Calculus. The permutative λ-calculus Λ_{p} introduced in this paper extends λ-calculus with three *equations* and one rewriting rule for permuting constructors. It sensibly generalises all previous extended λ-calculi by taking — when possible — the permutations as *equivalences*, and not as *reductions*. This is a key point of our approach. We show that the permutative λ-calculus preserves β-strong normalisation and is Church-Rosser modulo the equivalences, the strongest possible form of confluence for a reduction relation modulo an equivalence. Whenever an orientation of the equations (or a subset of them) yields a terminating reduction $\rightsquigarrow$ then the system where the equations

N. Bjørner and A. Voronkov (Eds.): LPAR-18, LNCS 7180, pp. 23–36, 2012.

are replaced by $\rightsquigarrow$ enjoys PSN. Thus, our result subsumes all PSN results of the kind in the literature.

The Proof Technique. We study the permutative λ-calculus through an auxiliary and new calculus with **explicit substitutions (ES)** called $\lambda\mathtt{sub}$. In this calculus β-reduction is split into two subsystems: $\rightarrow_{\mathtt{dB}}$ which creates a substituted term $t[x/u]$, *i.e.* a term t affected by a *delayed/explicit* substitution $[x/u]$, and $\rightarrow_{\mathtt{sub}}$ which executes the ES $[x/u]$ — getting $t\{x/u\}$ — and hence completes β-reduction. This simple calculus is then enriched with various equivalences — thus getting the equational $\lambda\mathtt{sub}$-calculus — obtained by what might be called an *extension by continuity*: if t and u are equivalent λ-terms in $\Lambda_{\hat{\mathtt{p}}}$ and they $\rightarrow_{\mathtt{dB}}$-reduce to t' and u', respectively, then t' and u' are equivalent in the equational $\lambda\mathtt{sub}$. This requires to consider equivalences on terms with ES and not only on λ-terms.

PSN. We prove PSN for the permutative λ-calculus by reducing this problem to PSN for the *equational* $\lambda\mathtt{sub}$-calculus, which in turn reduces to an existing result for the structural λ-calculus [2].

Confluence. Confluence of the permutative λ-calculus turns out to be delicate, and our proof is one of the main contributions of the paper. Indeed, confluence of $\Lambda_{\hat{\mathtt{p}}}$ does not follow from confluence of λ-calculus. The usual Tait–Martin Löf technique does not work, since the equations may create/hide redexes. While confluence of many reduction systems can usually be proved by means of developments [9], this notion does not suffice in the case of $\Lambda_{\hat{\mathtt{p}}}$, again because the equations create redexes. Its stronger variant, known as superdevelopments [13] or L-developments [1] — which also reduces some created redexes — does not work either. We then introduce a new form of development called M-development, show its good properties with respect to $\Lambda_{\hat{\mathtt{p}}}$ and then derive confluence for $\Lambda_{\hat{\mathtt{p}}}$. A key point is that M-developments are defined and studied through the equational $\lambda\mathtt{sub}$-calculus, where the splitting of β-reduction in terms of dB and sub becomes crucial to allow a fine analysis of redex creation. A nice fact is that our proof technique is modular, in the sense that one can choose to arbitrarily orient all or only some of the equations as rewriting rules while keeping the proof essentially unchanged. Moreover, our proof does not rely on confluence of λ-calculus.

Proof-Nets. Our work is the final product of a long-term study of the relation between ES and Linear Logic Proof-Nets. Here we present the implications of our study on λ-calculus, a language without ES, which is of a more general interest. No knowledge of Proof-Nets is assumed in this paper. However, in Sec. 4 the reader accustomed with Proof-Nets will find hints to the graphical intuitions for the main concepts. In particular, the equations of $\Lambda_{\hat{\mathtt{p}}}$ have a natural justification in terms of Proof-Nets.

Roadmap. Sec. 2 introduces the permutative λ-calculus. Sec. 3 explains the difficulties to prove confluence using the notion of development. Sec. 4 introduces the equational $\lambda\mathtt{sub}$-calculus and defines M-developments. Sec. 5 proves Church-Rosser of the reduction relation $\rightarrow_{\beta}$ modulo the equations and Sec. 6 extends the result to the whole calculus. Sec. 7 proves PSN and Sec. 8 concludes the paper.

$$
\begin{array}{lll}
(\lambda x.t)\ u & \mapsto_{\beta} & t\{x/u\} \\
t\ ((\lambda x.v)\ u) & \mapsto_{\hat{\mathtt{u}}} & (\lambda x.t\ v)\ u \qquad \text{if } x \notin \mathtt{fv}(t) \ \&\ x \notin \mathtt{fv}(v) \\
(\lambda x.\lambda y.t)\ u & \sim_{\hat{\sigma}_1} & \lambda y.((\lambda x.t)\ u) \qquad \text{if } y \notin \mathtt{fv}(u) \\
(\lambda x.t\ v)\ u & \sim_{\hat{\sigma}_2} & (\lambda x.t)\ u\ v \qquad \text{if } x \notin \mathtt{fv}(v) \\
(\lambda x.t\ v)\ u & \sim_{\widehat{\mathtt{box}}} & t\ ((\lambda x.v)\ u) \qquad \text{if } x \notin \mathtt{fv}(t) \ \&\ x \in \mathtt{fv}(v)
\end{array}
$$

Fig. 1. The permutative λ-calculus $\Lambda_{\hat{\mathtt{P}}}$

2 The Permutative λ-Calculus

The permutative λ-calculus $\Lambda_{\hat{\mathtt{P}}}$ is given by the set of λ-terms, written Λ, and a set of equations and reduction rules. As usual [3], the term x is called a **variable**, $\lambda x.t$ an **abstraction** and $t\ u$ an **application**. **Free** and **bound** variables of λ-terms are defined as usual and respectively written $\mathtt{fv}(t)$ and $\mathtt{bv}(t)$. The equivalence relation generated by the renaming of bound variables, written $\equiv_\alpha$ or simply $=$, is called **α-conversion**. The meta-level **substitution** operation is given, as usual, on α-equivalence classes; the notation $t\{x/u\}$ means that all the free occurrences of the variable x in the term t are substituted by u by avoiding capture of free variables. **Contexts** are defined as usual an denoted by C, we write $C[\![t]\!]$ for the context C where its unique hole has been replaced by the term t.

The rewriting rules and equations of the **permutative λ-calculus** $\Lambda_{\hat{\mathtt{P}}}$ are given in Fig. 1. The two equations $\hat{\sigma}_1$ and $\hat{\sigma}_2$ are exactly Regnier's σ-equivalence [17]. The equation $\widehat{\mathtt{box}}$ and the rule $\hat{\mathtt{u}}$, called **box** and **void unboxing** respectively, are instances of a more general equation called $\mathtt{badbox}$ obtained from $\widehat{\mathtt{box}}$ by removing the side condition "$x \in \mathtt{fv}(v)$". The equation $\mathtt{badbox}$ does not belong to $\Lambda_{\hat{\mathtt{P}}}$ because it is unsound: it breaks PSN as shown in Sec. 6. In order to simplify the presentation of our results we first treat the equations of $\Lambda_{\hat{\mathtt{P}}}$ (*i.e.* $\hat{\sigma}_1$, $\hat{\sigma}_2$ and $\widehat{\mathtt{box}}$), and consider the void unboxing rule $\mapsto_{\hat{\mathtt{u}}}$ only later, in Sec. 6. The *assoc* rule of [16,14,21] is a particular case of $\widehat{\mathtt{box}} \cup \hat{\mathtt{u}}$.

A **β-redex** is any term of the form $(\lambda x.t)\ u$. We define $\hat{\mathtt{P}}$ as the set of equations $\{\hat{\sigma}_1, \hat{\sigma}_2, \widehat{\mathtt{box}}\}$. The **reduction relation** $\to_\beta$ (resp. $\to_{\hat{\mathtt{u}}}$) is generated by the contextual closure of the rewriting rule $\mapsto_\beta$ (resp. $\mapsto_{\hat{\mathtt{u}}}$). We write $\to_{\{\beta,\hat{\mathtt{u}}\}}$ for $\to_\beta \cup \to_{\hat{\mathtt{u}}}$. The **permutative equivalence relation** $\equiv_{\hat{\mathtt{P}}}$ is generated by the contextual and reflexive-transitive closure of α-conversion and all the equations in $\hat{\mathtt{P}}$.

Given a reduction (resp. equivalence) relation $\mathcal{R}$ (resp. $\mathcal{E}$), the **reduction relation modulo** $\to_{\mathcal{R}/\mathcal{E}}$ is defined as $\mathcal{R}$-reduction on $\mathcal{E}$-equivalence classes, *i.e.* $t \to_{\mathcal{R}/\mathcal{E}} t'$ iff $\exists t_0, t_1$ s.t. $t \equiv_\mathcal{E} t_0 \to_\mathcal{R} t_1 \equiv_\mathcal{E} t'$. In this paper we give particular attention to the reduction relations $\to_{\beta/\hat{\mathtt{P}}}$ and $\to_{\{\beta,\hat{\mathtt{u}}\}/\hat{\mathtt{P}}}$.

Given any reduction relation $\mathcal{R}$, we use $\to^+_\mathcal{R}$ (resp. $\to^*_\mathcal{R}$) for the transitive (resp. reflexive-transitive) closure of $\mathcal{R}$. The notation $\leftrightarrow_\mathcal{R}$ is used for $\to_\mathcal{R} \cup {}_\mathcal{R}\!\leftarrow$ and $\to^k_\mathcal{R}$ for k compositions of $\to_\mathcal{R}$ with itself. A term t is in **$\mathcal{R}$-normal form**, written $t \in \mathcal{R}$**-nf**, if there is no t' such that $t \to_\mathcal{R} t'$. A term t has an **$\mathcal{R}$-normal form** iff there exists $t' \in \mathcal{R}$-nf such that $t \to^*_\mathcal{R} t'$. When t has a **unique** $\mathcal{R}$-normal form, this one is denoted by $\mathcal{R}(t)$. A reduction system $\mathcal{R}$ is **confluent** iff $t \to^*_\mathcal{R} u$ and $t \to^*_\mathcal{R} v$ implies there exists t' s.t. $u \to^*_\mathcal{R} t'$ and $v \to^*_\mathcal{R} t'$.

3 Towards Confluence of $\to_{\beta/\hat{\mathtt{P}}}$

Well-known notions [3,19] in λ-calculus equipped with β-reduction are those of residual, created redex, (complete) development, etc. Informally, a λ-term t is either a normal form or it contains some redexes. However, if one reduces all the redexes in t it is not always the case that the obtained term is a normal form: redexes can be dynamically created along a reduction. A **development** of a term t is a reduction sequence starting at t in which only *residuals* of redexes that already exist in t are contracted along the sequence. Developments always terminate [19]; moreover all complete (*i.e.* maximal) developments terminate on the same term [9].

From now on we only consider complete developments, and to simplify the text, we omit the adjective *complete*. Analogously for the other notions of developments to be introduced in a while. A known method to show confluence of reduction relations is based on developments, particularly using the so-called Z-property [20]. **A reduction relation $\mathcal{R}$ satisfies the Z-property** iff there exists a map $^\sharp$ s.t.

$$\text{for all } t, \text{ for all } u,\ \ t \to_{\mathcal{R}} u \text{ implies } u \to_{\mathcal{R}}^* t^\sharp \text{ and } t^\sharp \to_{\mathcal{R}}^* u^\sharp$$

The requirement $u \to_{\mathcal{R}}^* t^\sharp$ expresses the fact that $t^\sharp$ is obtained by reducing at least all redexes in t, hence it abstracts away the role of developments. Note that if $\mathcal{R}$ satisfies the Z-property, and $t = t^\sharp$ for every $\mathcal{R}$-normal form, then $t \to_{\mathcal{R}}^* t^\sharp$ holds for every term t.

Theorem 1. *[20] If $\mathcal{R}$ satisfies the Z-property, then $\mathcal{R}$ is confluent.*

Confluence of β-reduction in λ-calculus can be proved by defining the map $^\sharp$ to be the function which computes the (complete) development of a term. To use this same technique to prove confluence of the relation $\to_{\beta/\hat{\mathtt{P}}}$, one first needs to generalise the Z-property to reduction modulo. **The reduction relation $\mathcal{R}$ satisfies the Z-property modulo the equivalence relation** $\mathtt{E}$ if there exists a map $^\sharp$ s.t. for all t, for all u,

1. $t \to_{\mathcal{R}} u$ implies $u \to_{\mathcal{R}}^* t^\sharp$ and $t^\sharp \to_{\mathcal{R}}^* u^\sharp$, and
2. $t\ \mathtt{E}\ u$ implies $t^\sharp = u^\sharp$.

It is easy to show that if $\mathcal{R}$ satisfies the Z-property modulo $\mathtt{E}$, then the reduction relation $\to_{\mathcal{R}/\mathtt{E}}$ is confluent. Actually, it implies that $\mathcal{R}$ is Church-Rosser modulo $\mathtt{E}$, which is the strongest possible notion of confluence in the realm of reduction modulo.

Lemma 1 (Z-property modulo $\Rightarrow$ Church-Rosser modulo). *If $\mathcal{R}$ satisfies the Z-property modulo $\mathtt{E}$, then $\mathcal{R}$ is Church-Rosser modulo $\mathtt{E}$, i.e. $\forall t, \forall u, \exists t_1, \exists u_1$ s.t. $t\ (\leftrightarrow_{\mathcal{R}} \cup\ \mathtt{E})^*\ u$ implies $t \to_{\mathcal{R}}^*\ t_1\ \mathtt{E}\ u_1\ {}_{\mathcal{R}}^*\!\leftarrow\ u$.*

Proof. Let $^\sharp$ be the map satisfying the Z-property for $\mathcal{R}$ modulo $\mathtt{E}$. Define $t^{\sharp\sharp} := t$ if t is an $\mathcal{R}$-nf, $t^{\sharp\sharp} := t^\sharp$ otherwise. Trivially, also $^{\sharp\sharp}$ satisfies the Z-property for $\mathcal{R}$

modulo E. Moreover, $t \to^*_{\mathcal{R}} t^{\sharp\sharp}$ for every term t. Now, by the Z-property, $t \leftrightarrow_{\mathcal{R}} u$ implies $t^{\sharp\sharp} \leftrightarrow^*_{\mathcal{R}} u^{\sharp\sharp}$ and t E u implies $t^{\sharp\sharp} = u^{\sharp\sharp}$. Thus, $t\ (\leftrightarrow_{\mathcal{R}} \cup \mathtt{E})^*\ u$ implies $t^{\sharp\sharp} \leftrightarrow^*_{\mathcal{R}} u^{\sharp\sharp}$. Since $\mathcal{R}$ is confluent (Th. 1), then $\exists\ v$ s.t. $t^{\sharp\sharp} \to^*_{\mathcal{R}} v\ {}^*_{\mathcal{R}}\!\leftarrow u^{\sharp\sharp}$. We then conclude $t \to^*_{\mathcal{R}} t^{\sharp\sharp} \to^*_{\mathcal{R}} v\ {}^*_{\mathcal{R}}\!\leftarrow u^{\sharp\sharp}\ {}^*_{\mathcal{R}}\!\leftarrow u$.

Church-Rosser modulo has two important corollaries.

Corollary 1. *[19] Let $\mathcal{R}$ be Church-Rosser modulo* E*. Then:*

1. **Uniqueness of Normal Forms***: if $t\ (\leftrightarrow_{\mathcal{R}} \cup \mathtt{E})^*\ u$ and $t \to^*_{\mathcal{R}} t_1$ and $u \to^*_{\mathcal{R}} u_1$ and t_1, u_1 are $\mathcal{R}$-nf, then t_1 E u_1.*
2. **Confluence of the reduction modulo***: if $t \to^*_{\mathcal{R}/\mathtt{E}} u_i\ (i = 1, 2)$, then $\exists\ t'$ s.t. $u_i \to^*_{\mathcal{R}/\mathtt{E}} t'\ (i = 1, 2)$.*

The first natural attempt to prove confluence for the reduction relation $\to_{\beta/\hat{\mathtt{P}}}$ is then to use Lem. 1 by choosing $^\sharp$ as the development function. Unfortunately, this idea does not work: for instance $t = (\lambda x.\lambda y.y)\ z\ w \equiv_{\hat{\sigma}_1} u = (\lambda y.((\lambda x.y)\ z))\ w$ but $(\lambda y.y)\ w$, the development of t, is different from w, the development of u. The reason is that $\hat{\sigma}_1$ *creates redexes.*

In λ-calculus creation of redexes can be classified in three types [15]:

(Type 1) $((\lambda x.\lambda y.t)\ u)\ v \to_\beta (\lambda y.t\{x/u\})\ v$.
(Type 2) $(\lambda x.x)\ (\lambda y.t)\ u \to_\beta (\lambda y.t)\ u$.
(Type 3) $(\lambda x.C[x\ v])\ (\lambda y.u) \to_\beta C\{x/\lambda y.u\}[(\lambda y.u)\ v\{x/\lambda y.u\}]$

Developments do not work to show confluence of $\to_{\beta/\hat{\mathtt{P}}}$ because $\hat{\sigma}_1$ *anticipates/postpones* creations of Type 1, making these creations visible/hidden in the starting term. However, another well-known notion of development, called superdevelopment [13] or **L-development** [1], exists. A (complete) superdevelopment [13] of a term t is a reduction sequence starting at t in which only *residuals* of redexes that already exist in t and created redexes of Type 1 and 2 are allowed to be contracted along the sequence.

Unfortunately, superdevelopments do not work either: for instance $t = (\lambda x.(x\ y))\ I \equiv_{\hat{\sigma}_2} (\lambda x.x)\ I\ y = u$, where I is the identity function, but their superdevelopments are different. Now the reason is more subtle: $\hat{\sigma}_2$ does not anticipate creations of Type 2, but it turns future creations of Type 2 (e.g. in u) into creations of Type 3 (in t), or viceversa. The solution is to weaken the notion of L-development to that of M-development. A (complete) **M-development** of a term t is a reduction sequence starting at t in which only *residuals* of redexes that already exist in t and created redexes of Type 1 — but not of Type 2 — are allowed to be contracted along the sequence. We are going to define the result t° of a (complete) M-development by using an auxiliary calculus, λsub, having explicit substitutions. On one hand this seems to be necessary, because apparently there is no way to describe such a term by induction on t inside $\Lambda_{\hat{\mathtt{P}}}$, as it is the case for (L-)developments. On the other hand the use of λsub will not be costly: we shall obtain a concise proof of confluence for $\to_{\beta/\hat{\mathtt{P}}}$.

$$\begin{array}{ll} (\lambda x.t)\mathtt{L}\ u & \mapsto_{\mathtt{dB}}\ t[x/u]\mathtt{L} \\ t[x/u] & \mapsto_{\mathtt{sub}}\ t\{x/u\} \end{array}$$

Fig. 2. The λsub-calculus

4 The Auxiliary λsub-Calculus

This section introduces the λsub-calculus which is used as an auxiliary tool to show confluence of the permutative λ-calculus. The set $\mathcal{T}$ of terms of the λsub-calculus is given by **variables** x, **abstractions** $\lambda x.t$, **applications** $t\ u$ and **substituted terms** $t[x/u]$. The object $[x/t]$, which is not a term, is called an **explicit substitution (ES)**. We consider **free** and **bound** variables of terms with ES as usual [11]. The meta-level **substitution** operation and the α-conversion operation are extended from Λ to $\mathcal{T}$ as expected. We use $\mathtt{L}$ to denote a possibly empty list of ES $[y_1/t_1]\dots[y_m/t_m]$. We write C to denote a **context** in λsub. The rewriting rules of the λsub-calculus are given in Fig. 2.

One feature of λsub is that rule dB acts **at a distance**, as in Proof-Nets [8]. Indeed, the list L of ES introduces some distance between the function $\lambda x.t$ and its argument u in a term of the form $(\lambda x.t)\mathtt{L}\ u$. Rule $\to_{\mathtt{dB}}$ (resp. $\to_{\mathtt{sub}}$) corresponds exactly to the multiplicative (resp. exponential) cut-elimination rule of Pure Proof-Nets. Another feature of λsub is that it splits $\to_\beta$, which does not always terminate, into two terminating and confluent reduction systems dB and sub (property which follows from [1]), through which $\to_\beta$ can be finely studied.

Lemma 2. *The reduction system* dB *(resp.* sub*) is confluent and terminating, thus* dB *(resp.* sub*)-normal forms always exist and are unique.*

From now on, we write $\mathtt{dB}(t)$ (resp. $\mathtt{sub}(t)$) for the unique dB-normal form (resp. sub-normal form) of the term t.

Lemma 3. *The* sub*-function enjoys the following equalities.*

$$\begin{array}{llll} \mathtt{sub}(x) & = x & \mathtt{sub}(t[x/u]) & = \mathtt{sub}(t)\{x/\mathtt{sub}(u)\} \\ \mathtt{sub}(\lambda x.t) & = \lambda x.\mathtt{sub}(t) & \mathtt{sub}(t\ u) & = \mathtt{sub}(t)\ \mathtt{sub}(u) \end{array}$$

The following property is of course expected, it is shown by induction on the reduction relation by using the previous characterisation.

Lemma 4 (Projection on $\to_\beta$). *If* $t_0 \to_{\lambda\mathtt{sub}} t_1$*, then* $\mathtt{sub}(t_0) \to_\beta^* \mathtt{sub}(t_1)$.

We now define the **M-development** of a term $t \in \mathcal{T}$ as a special normal form in λsub:

$$t^\circ := \mathtt{sub}(\mathtt{dB}(t))$$

The M-development of t thus reduces all its multiple applications, *i.e.* applications of functions to several arguments. Consider a simple example of creation of Type 1 which applies a function to two arguments: $((\lambda x.\lambda y.y)\ z)\ z' \to_\beta (\lambda y.y)\ z'$. The reduction to dB-normal form:

$$((\lambda x.\lambda y.y)\ z)\ z' \to_{\mathtt{dB}} (\lambda y.y)[x/z]\ z' \to_{\mathtt{dB}} y[y/z][x/z']$$

$$
\begin{array}{lll}
(\lambda x.t)\mathtt{L}\ u & \mapsto_{\mathtt{dB}} & t[x/u]\mathtt{L} \\
t[x/u] & \mapsto_{\mathtt{sub}} & t\{x/u\}
\end{array}
$$

$$
\begin{array}{lll}
(\lambda x.\lambda y.t)\ u & \sim_{\hat{\sigma}_1} & \lambda y.((\lambda x.t)\ u) \quad \text{if } y \notin \mathtt{fv}(u) \\
(\lambda x.t\ v)\ u & \sim_{\hat{\sigma}_2} & (\lambda x.t)\ u\ v \quad \text{if } x \notin \mathtt{fv}(v) \\
t\ ((\lambda x.v)\ u) & \sim_{\widehat{\mathtt{box}}} & (\lambda x.t\ v)\ u \quad \text{if } x \notin \mathtt{fv}(t)\ \&\ x \in \mathtt{fv}(v)
\end{array}
$$

$$
\begin{array}{lll}
t[x/s][y/v] & \sim_{\mathtt{CS}} & t[y/v][x/s] \quad \text{if } x \notin \mathtt{fv}(v)\ \&\ y \notin \mathtt{fv}(s) \\
\lambda y.(t[x/s]) & \sim_{\sigma_1} & (\lambda y.t)[x/s] \quad \text{if } y \notin \mathtt{fv}(s) \\
t[x/s]\ v & \sim_{\sigma_2} & (t\ v)[x/s] \quad \text{if } x \notin \mathtt{fv}(v) \\
(t\ v)[x/u] & \sim_{\mathtt{box}_1} & t\ v[x/u] \quad \text{if } x \notin \mathtt{fv}(t)\ \&\ x \in \mathtt{fv}(v) \\
t[y/v][x/u] & \sim_{\mathtt{box}_2} & t[y/v[x/u]] \quad \text{if } x \notin \mathtt{fv}(t)\ \&\ x \in \mathtt{fv}(v)
\end{array}
$$

Fig. 3. The calculus $\lambda\mathtt{sub}/\mathit{\Pi}$

reduces in particular a created dB-redex, then, reduction to sub-normal form $y[y/z][x/z] \to^*_{\mathtt{sub}} z$ completes the M-development. Note that the second dB-step is possible only because the rule acts at a *distance*.

Note that the definition of **M-development** uses the reduction rules of $\lambda\mathtt{sub}$, which are external to $\Lambda_{\hat{\mathtt{P}}}$, since they are defined on $\mathcal{T}$ and not on Λ. However, this definition makes sense also when one looks only at $\Lambda_{\hat{\mathtt{P}}}$, as stated by next Lemma, which can be shown by induction using Lem. 4.

Lemma 5. *Let t be a λ-term. Then t° is a λ-term and $t \to^*_\beta t^\circ$.*

The Z-property for the permutative λ-calculus can be proved by means of M-developments. One of the properties to be verified is that $t_0 \equiv_{\hat{\mathtt{P}}} t_1$ implies $t_0^\circ = t_1^\circ$, which is quite tricky. To simplify this proof, and also to later show PSN for $\Lambda_{\hat{\mathtt{P}}}$, we need to extend $\lambda\mathtt{sub}$ with some equations, resulting in the equational calculus $\lambda\mathtt{sub}/\mathit{\Pi}$ in Fig. 3.

The equations are divided in two groups $\hat{\mathtt{P}} = \{\hat{\sigma}_1, \hat{\sigma}_2, \widehat{\mathtt{box}}\}$ and $\mathtt{P} = \{\mathtt{CS}, \sigma_1, \sigma_2, \mathtt{box}_1, \mathtt{box}_2\}$. We write $\equiv_{\hat{\mathtt{P}}}$, $\equiv_{\mathtt{P}}$ and $\equiv_{\mathit{\Pi}}$ for the contextual, reflexive-transitive closure of α-conversion and all the equations in $\hat{\mathtt{P}}$, $\mathtt{P}$ and $\hat{\mathtt{P}} \cup \mathtt{P}$, respectively. We use $\to_{\lambda\mathtt{sub}/\mathit{\Pi}}$ for reduction $\to_{\lambda\mathtt{sub}}$ modulo $\equiv_{\mathit{\Pi}}$.

The first group $\hat{\mathtt{P}}$ of equations is the same of $\Lambda_{\hat{\mathtt{P}}}$, but now also on terms with ES. The second group $\mathtt{P}$ is obtained by projecting the equations of $\hat{\mathtt{P}}$ acting on λ-terms into terms with ES by means of $\to_{\mathtt{dB}}$:

$$
\begin{array}{ccc@{\qquad}ccc}
(\lambda x.\lambda y.t)\ u & \equiv_{\hat{\sigma}_1} & \lambda y.((\lambda x.t)\ u) & ((\lambda x.t)\ u)\ v & \equiv_{\hat{\sigma}_2} & (\lambda x.(t\ v))\ u \\
\downarrow_{\mathtt{dB}} & & \downarrow_{\mathtt{dB}} & \downarrow_{\mathtt{dB}} & & \downarrow_{\mathtt{dB}} \\
(\lambda y.t)[x/u] & \equiv_{\sigma_1} & \lambda y.(t[x/u]) & (t[x/u])\ v & \equiv_{\sigma_2} & (t\ v)[x/u]
\end{array}
\qquad (A)
$$

Analogously, $\equiv_{\mathtt{box}_1}$ and $\equiv_{\mathtt{box}_2}$ are obtained by projecting $\equiv_{\widehat{\mathtt{box}}}$. Thus,

$$
\begin{array}{ccc@{\qquad}ccc}
t\ ((\lambda x.v)\ u) & \equiv_{\widehat{\mathtt{box}}} & ((\lambda x.t\ v)\ u) & (\lambda y.t)\ ((\lambda x.v)\ u) & \equiv_{\widehat{\mathtt{box}}} & ((\lambda x.(\lambda y.t)\ v)\ u) \\
\downarrow_{\mathtt{dB}} & & \downarrow_{\mathtt{dB}} & \downarrow_{*\mathtt{dB}} & & \downarrow_{*\mathtt{dB}} \\
t\ v[x/u] & \equiv_{\mathtt{box}_1} & (t\ v)[x/u] & t[y/v[x/u]] & \equiv_{\mathtt{box}_2} & t[y/v][x/u]
\end{array}
$$

Obtaining $\equiv_{\mathtt{CS}}$ is more subtle, indeed $t[y/v][x/u] \equiv_{\mathtt{CS}} t[x/u][y/v]$ can be understood as the dB-projection of $(\lambda x.((\lambda y.t)v))u \equiv_{\hat{\sigma}_1,\hat{\sigma}_2} (\lambda y.((\lambda x.t)u))v$. The

equivalence relation generated by the equations $\{\mathtt{CS}, \sigma_1, \sigma_2\}$ on the set $\mathcal{T}$ can be understood by means of the translation from terms with ES to Pure Proof-Nets. Equations $\{\mathtt{box}_1, \mathtt{box}_2\}$ are obtained by taking the box-box commutative rule of Proof-Nets as an equation rather than a rule (which is a novelty of our study). The equations of $\hat{\mathtt{P}}$, which are exactly those of $\Lambda_{\hat{\mathtt{P}}}$, are obtained by lifting those of $\mathtt{P}$ from terms with ES to λ-terms, which is the reason for writing $\widehat{\ }$.

5 The Z-property Modulo by Means of $\mathtt{M}$-developments

We prove here that $\to_\beta$ satisfies the Z-property modulo $\equiv_{\hat{\mathtt{P}}}$ by means of the notion of $\mathtt{M}$-developments introduced in Sec. 4. The new equivalence $\equiv_{\mathtt{P}}$ on terms with ES allows to prove that $t_0 \equiv_{\hat{\mathtt{P}}} t_1$ implies $t_0^\circ = t_1^\circ$ by *continuously* extending $\equiv_{\hat{\mathtt{P}}}$ through reduction. This notion of continuity is a strong form of the so-called **local coherence** (see [19], pp. 769-770).

Lemma 6. *Let $t, u, u_1, u_2 \in \mathcal{T}$.*

1. **$\mathtt{dB}$-Continuity of $\equiv_\Pi$**: *if $t \equiv_\Pi u_1$ and $t \to_{\mathtt{dB}} u_2$ then exists v s. t. $u_1 \to_{\mathtt{dB}} v$ and $u_2 \equiv_\Pi v$.*
2. **Projecting $\equiv_\Pi$ by $\mathtt{dB}$-nf**: *if $t \equiv_\Pi u$ then $\mathtt{dB}(t) \equiv_{\mathtt{P}} \mathtt{dB}(u)$.*
3. **Projecting $\equiv_{\mathtt{P}}$ by $\mathtt{sub}$-nf**: *if $t \equiv_{\mathtt{P}} u$ then $\mathtt{sub}(t) = \mathtt{sub}(u)$.*
4. **Projecting $\equiv_\Pi$ by $\mathtt{M}$-developments**: *if $t \equiv_\Pi u$ then $t^\circ = u^\circ$.*

Proof. 1. By induction on $\equiv_\Pi$. The base case is as in the equations labelled (A) on Page 29. For instance: if $t = (\lambda x.(s\ w))\ r \sim_{\widehat{\mathtt{box}}} s\ ((\lambda x.w)\ r) = u_1$ with $x \notin \mathtt{fv}(s)$ and $x \in w$, and if $t \to_{\mathtt{dB}} (s\ w)[x/r] = u_2$ then $u_1 \to_{\mathtt{dB}} s\ w[x/r] = v$ and $u_2 \sim_{\mathtt{box}_1} v$. The inductive cases are straightforward.
2. By induction on the length of $t \to_{\mathtt{dB}}^* \mathtt{dB}(t)$ using Point 1. one gets $\mathtt{dB}(t) \equiv_\Pi \mathtt{dB}(u)$. We conclude since $\equiv_\Pi$ coincides with $\equiv_{\mathtt{P}}$ on $\mathtt{dB}$-nfs.
3. By induction on $\equiv_{\mathtt{P}}$ using the characterisation in Lem. 3.
4. By composing the two previous points.

Now we need to show the Z-property for $\to_\beta$ with respect to $\mathtt{M}$-developments. This is done by analysing the commutation of $\to_{\mathtt{dB}}$ and $\to_{\mathtt{sub}}$. We first prove the result for $\to_{\lambda\mathtt{sub}}$, thus obtaining the Z-property for $\to_\beta$ modulo $\hat{\mathtt{P}}$ as a corollary.

Lemma 7 (Commutation of $\to_{\mathtt{sub}}^*$ and $\to_{\mathtt{dB}}^*$). *Let $t, u_1, u_2 \in \mathcal{T}$. If $t \to_{\mathtt{sub}}^k u_1$ and $t \to_{\mathtt{dB}}^h u_2$ then there exists v s.t. $u_2 \to_{\mathtt{sub}}^k v$ and $u_1 \to_{\mathtt{dB}}^* v$.*

Proof. By induction on the pair $\langle k, h\rangle$ ordered lexicographically, using local commutation (case $k = h = 1$), which is proved by induction on t.

Lemma 8 (Z-property for $\to_{\lambda\mathtt{sub}}$). *Let $t, u \in \mathcal{T}$. Then:*

1. *If $t \to_{\lambda\mathtt{sub}} u$ then $u \to_{\lambda\mathtt{sub}}^* t^\circ$.*
2. *If $t \to_{\lambda\mathtt{sub}} u$ then $t^\circ \to_{\lambda\mathtt{sub}}^* u^\circ$.*

Proof. 1. If $t \to_{\mathtt{dB}} u$ then $t^\circ = u^\circ$ (thus $u \to^*_{\lambda\mathtt{sub}} u^\circ = t^\circ$ holds). If $t \to_{\mathtt{sub}} u$ then by Lem. 7 $\exists\ v$ s.t. $\mathtt{dB}(t) \to^*_{\mathtt{sub}} v$ and $u \to^*_{\mathtt{dB}} v$. We have $\mathtt{sub}(\mathtt{dB}(t)) = \mathtt{sub}(v)$ and so $u \to^*_{\mathtt{dB}} v \to^*_{\mathtt{sub}} \mathtt{sub}(\mathtt{dB}(t)) = t^\circ$.
2. If $t \to_{\mathtt{dB}} u$ then $t^\circ = u^\circ$. If $t \to_{\mathtt{sub}} u$ then by Lem. 7 exists v s.t. $u \to^*_{\mathtt{dB}} v$ and $\mathtt{dB}(t) \to^*_{\mathtt{sub}} v$. By definition $v \to^*_{\mathtt{dB}} \mathtt{dB}(u)$ and $v \to^*_{\mathtt{sub}} t^\circ$. By Lem. 7 exists w s.t. $\mathtt{dB}(u) \to^*_{\mathtt{sub}} w$ and $t^\circ \to^*_{\mathtt{dB}} w$. By definition $w \to^*_{\mathtt{sub}} \mathtt{sub}(\mathtt{dB}(u)) = u^\circ$, therefore $t^\circ \to^*_{\lambda\mathtt{sub}} u^\circ$.

Corollary 2 (Z-property for $\to_\beta$). *Let $t, u \in \Lambda$. Then:*

1. *If $t \to_\beta u$ then $u \to^*_\beta t^\circ$.*
2. *If $t \to_\beta u$ then $t^\circ \to^*_\beta u^\circ$.*

Proof. 1. If $t \to_\beta u$ then $t \to_{\mathtt{dB}} u_1 \to_{\mathtt{sub}} u$. We have $t^\circ = u_1^\circ$ and by Lem. 8.1 $u \to^*_{\lambda\mathtt{sub}} u_1^\circ$, hence $u \to^*_{\lambda\mathtt{sub}} t^\circ$. By Lem. 4 we get $\mathtt{sub}(u) \to^*_\beta \mathtt{sub}(t^\circ)$ and we conclude since both $\mathtt{sub}(u) = u$ and $\mathtt{sub}(t^\circ) = t^\circ$ hold, given that both u and t° are λ-terms.
2. If $t \to_\beta u$, then $t \to_{\mathtt{dB}} u_1 \to_{\mathtt{sub}} u$, Lem. 8.2 gives $t^\circ \to^*_{\lambda\mathtt{sub}} u_1^\circ \to^*_{\lambda\mathtt{sub}} u^\circ$. As in the previous point we conclude $t^\circ \to^*_\beta u^\circ$ by Lem. 4.

Thus we get:

Theorem 2. *1. The relation $\to_{\lambda\mathtt{sub}}$ is Church-Rosser modulo $\equiv_\Pi$.*
2. The relation $\to_\beta$ is Church-Rosser modulo $\equiv_{\hat{\mathtt{P}}}$.

Proof. Lem. 8 and Lem. 6.3 prove the Z-property for $\to_{\lambda\mathtt{sub}}$ modulo $\equiv_\Pi$. Cor. 2 and Lem. 6.4 prove the Z-property for $\to_\beta$ modulo $\hat{\mathtt{P}}$. Church-Rosser modulo follows in both cases from Lem. 1.

Note that our proof of confluence for $\Lambda_{\hat{\mathtt{P}}}$ and $\lambda\mathtt{sub}$ modulo $\equiv_\Pi$ *does not* use confluence of λ-calculus.

6 Adding the Unboxing Rule

In this section we add the void unboxing rule in order to lift our confluence result from $\to_{\beta/\hat{\mathtt{P}}}$ to $\to_{\{\beta,\hat{\mathtt{u}}\}/\hat{\mathtt{P}}}$.

The application construct $t\ u$ is *linear* in t and *non-linear* in u, in the sense that u can be duplicated/erased (for instance if $t = \lambda x.x\ x$) while t cannot. The translation of λ-calculus into Linear Logic makes this point explicit: u is placed in a !-box—the construction allowing non-linearity—while t is not. The natural permutation (note the absence of the side condition "$x \in \mathtt{fv}(v)$"):

$$t\ ((\lambda x.v)\ u) \sim_{\mathtt{badbox}} (\lambda x.t\ v)\ u \qquad \text{if } x \notin \mathtt{fv}(t)$$

is delicate, because it permutes a redex in/out of a non-linear sub-term, and thus affects reduction lengths. Indeed, we now show that $\to_\beta$ plus the equations $\{\hat{\sigma}_2, \mathtt{badbox}\}$ does *not* preserve β-strong normalisation, *i.e.* there exists $t \in \mathcal{SN}_\beta$

$$B[\![t[x/u]]\!] \mapsto_{\mathtt{u}} B[\![t]\!][x/u] \quad \text{if } B \text{ does not capture variables in } \mathtt{fv}(u)$$

Fig. 4. The unboxing rule

s.t. $t \notin \mathcal{SN}_{\beta/\{\hat{\sigma}_2,\mathtt{badbox}\}}$ as the following example shows. Let $t = (\lambda x.u)\ u$, where $u = (\lambda z.z\ z)\ y$. Then,

$$\begin{array}{lllll} t = (\lambda x.u)\ u & = & (\lambda x.((\lambda z.z\ z)\ y))\ u & \equiv_{\mathtt{badbox}} \\ (\lambda z.z\ z)((\lambda x.y)\ u) & \to_\beta & ((\lambda x.y)\ u)\ ((\lambda x.y)\ u) & \equiv_{\hat{\sigma}_2} \\ (\lambda x.y\ ((\lambda x.y)\ u))\ u & \equiv_{\mathtt{badbox}} & (\lambda x.(\lambda x.y\ y)\ u)\ u & \equiv_{\mathtt{badbox}} \\ (\lambda x.y\ y)\ ((\lambda x.u)\ u) & = & (\lambda x.y\ y)\ t & \end{array}$$

The term t reduces to a term containing t so that $t \notin \mathcal{SN}_{\beta/\{\hat{\sigma}_2,\mathtt{badbox}\}}$. Note that in the counter-example the equation `badbox` is used with respect to a λ-abstraction binding a variable which does not occur in the body.

Thus we split the previous equation $\sim_{\mathtt{badbox}}$ in two cases: the case "$x \in \mathtt{fv}(v)$" goes to the equation $\widehat{\mathtt{box}}$, while the case "$x \notin \mathtt{fv}(v)$" is captured by the void unboxing rewriting rule $\mapsto_{\hat{\mathtt{u}}}$ in Fig. 11, which is just an orientation from left to right of the dangerous equation $\sim_{\mathtt{badbox}}$. The idea behind a reduction step $t\ ((\lambda x.v)\ u) \to_{\hat{\mathtt{u}}} (\lambda x.t\ v)\ u$ is that both sides of the rule β-reduce to $t\ v$, or, equivalently, the permuted redex simply erases u, which therefore can be considered as *garbage*. The interest in permuting garbage is to get it out of the arguments, so that it does not get duplicated. Indeed, consider the case where t (in the rule) is $\lambda y.y\ y$: a β-reduction step from the left-hand side duplicates u, while this is not the case for the right-hand side.

Void unboxing is a rewriting rule but it behaves exactly as the other equations with respect to $\mathtt{M}$-developments, *i.e.* $t_0 \to_{\hat{\mathtt{u}}} t_1$ implies $t_0^\circ = t_1^\circ$. To show this property we proceed as for the other equations, *i.e.* we extend $\lambda\mathtt{sub}/\mathit{\Pi}$ with a rule $\mapsto_{\mathtt{u}}$ reflecting $\mapsto_{\hat{\mathtt{u}}}$ on terms with ES. To specify the rewriting rule $\mapsto_{\mathtt{u}}$, we first need to define a special notion of context. A **boxed context** B is given by the following grammar:

$$B ::= t\ C \mid t[x/C] \mid B\ t \mid B[x/t] \mid \lambda y.B$$

where C denotes a context in $\lambda\mathtt{sub}$. The name *boxed context* is justified by the Proof-Net representation of λ-terms (with explicit substitutions): every argument of an application or content of an ES is denoted by a !-box, hence the hole of a boxed context necessarily occurs inside a !-box. The unboxing rule for terms with ES is the context closure of the rule in Fig. 4 (for technical reasons $\mapsto_{\mathtt{u}}$ is more general than the projection of $\mapsto_{\hat{\mathtt{u}}}$ by $\mathtt{dB}$-steps). Next lemma relates unboxing and $\mathtt{M}$-developments.

Lemma 9. *Let $t, u_1, u_2 \in \mathcal{T}$.*

1. **Commutation of $\to_{\{\mathtt{u},\hat{\mathtt{u}}\}}$ and $\to_{\mathtt{dB}}^*$:** *if $t \to_{\{\mathtt{u},\hat{\mathtt{u}}\}} u_2$ and $t \to_{\mathtt{dB}}^k u_1$ then there exists v s.t. $u_2 \to_{\mathtt{dB}}^k v$ and $u_1 \to_{\{\mathtt{u},\hat{\mathtt{u}}\}} v$.*
2. **Projecting $\to_{\{\mathtt{u},\hat{\mathtt{u}}\}}$ by $\mathtt{dB}$-nf:** *If $t \to_{\{\mathtt{u},\hat{\mathtt{u}}\}} u$, then $\mathtt{dB}(t) \to_{\mathtt{u}} \mathtt{dB}(u)$.*

3. **Projecting $\to_{\{\mathtt{u},\hat{\mathtt{u}}\}}$ by $\mathtt{M}$-developments***: If $t \to_{\{\mathtt{u},\hat{\mathtt{u}}\}} u$, then $t^\circ = u^\circ$.*

Proof. 1. By induction on k. The case $k = 1$ is by induction on $t \to_{\{\mathtt{u},\hat{\mathtt{u}}\}} u_2$. The only interesting subcases are the root ones for $t \to_{\hat{\mathtt{u}}} u_2$ and $t \to_{\mathtt{u}} u_2$, given by the following diagrams:

$$\begin{array}{ccc} t = s\ ((\lambda x.w)\ r) & \to_{\hat{\mathtt{u}}} & (\lambda x.s\ w)\ r = u_2 \\ \downarrow_{\mathtt{dB}} & & \downarrow_{\mathtt{dB}} \\ s\ w[x/u] & \to_{\mathtt{u}} & (s\ w)[x/r] \end{array} \qquad \begin{array}{ccc} t = (\lambda y.s)\mathtt{L}\ w[x/u] & \to_{\mathtt{u}} & ((\lambda y.s)\mathtt{L}\ w)[x/u] = u_2 \\ \downarrow_{\mathtt{dB}} & & \downarrow_{\mathtt{dB}} \\ s[y/w[x/u]]\mathtt{L} & \to_{\mathtt{u}} & s[y/w]\mathtt{L}[x/u] \end{array}$$

The other subcases and inductive cases are all straightforward.
2. By the previous point there exists v s.t. $\mathtt{dB}(t) \to_{\mathtt{u},\hat{\mathtt{u}}} v$ and $u \to^*_{\mathtt{dB}} v$. But $\mathtt{dB}$-normal forms cannot $\to_{\hat{\mathtt{u}}}$-reduce, so $\mathtt{dB}(t) \to_{\mathtt{u}} v$. Moreover, $\to_{\mathtt{u}}$ cannot create $\mathtt{dB}$-redexes, so $v = \mathtt{dB}(v) = \mathtt{dB}(u)$.
3. From 2. we get $\mathtt{dB}(t) \to_{\mathtt{u}} \mathtt{dB}(u)$. Thus $t^\circ = \mathtt{sub}(\mathtt{dB}(t)) = \mathtt{sub}(\mathtt{dB}(u)) = u^\circ$ since $\mathtt{u}$-reduction only moves one void substitution.

Thus we can extend our confluence results to void unboxing.

Corollary 3 (Z-property for unboxing). *Let $t, t_0 \in \Lambda$ and $u, u_0 \in \mathcal{T}$.*

1. *If $t \to_{\{\lambda\mathtt{sub},\hat{\mathtt{u}},\mathtt{u}\}} t_0$ then $t_0 \to^*_{\{\lambda\mathtt{sub},\hat{\mathtt{u}},\mathtt{u}\}} t^\circ$ and $t^\circ \to^*_{\{\lambda\mathtt{sub},\hat{\mathtt{u}},\mathtt{u}\}} t_0^\circ$.*
2. *If $u \to_{\{\beta,\hat{\mathtt{u}}\}} u_0$ then $u_0 \to^*_{\{\beta,\hat{\mathtt{u}}\}} u^\circ$ and $u^\circ \to^*_{\{\beta,\hat{\mathtt{u}}\}} u_0^\circ$.*

Proof. 1. For $\to_{\lambda\mathtt{sub}}$ use Lem. 8. Suppose $t \to_{\{\hat{\mathtt{u}},\mathtt{u}\}} t_0$. Then $t_0 \to^*_{\lambda\mathtt{sub}} t_0^\circ$ by definition and $t_0^\circ = t^\circ$ by Lem. 9.3, which allow us to conclude.
2. For $\to_\beta$ use Cor. 2. For $\to_{\hat{\mathtt{u}}}$ use $u_0 \to^*_\beta u_0^\circ$ (Lem. 5) and Lem. 9.3.

As before we get:

Theorem 3. *1. The relation $\to_{\{\lambda\mathtt{sub},\hat{\mathtt{u}},\mathtt{u}\}}$ is Church-Rosser modulo $\equiv_{\Pi}$.*
2. The relation $\to_{\{\beta,\hat{\mathtt{u}}\}}$ is Church-Rosser modulo $\equiv_{\hat{\mathtt{P}}}$.

The extension of Church-Rosser modulo to unboxing relies on the fact that $t \to_{\hat{\mathtt{u}}} u$ implies $t^\circ = u^\circ$. Let $\to_{\hat{\mathtt{P}}}$ be the reduction system obtained by an arbitrary orientation of the equations in the set $\hat{\mathtt{P}}$. The reduction $\to_{\hat{\mathtt{P}}}$ enjoys the same property above for $\to_{\hat{\mathtt{u}}}$. Hence, we easily get the Z-property for $\to_{\{\beta,\hat{\mathtt{u}},\hat{\mathtt{P}}\}}$, and thus confluence holds. Note that even a stronger fact holds: it is possible to orient only some of the equations in $\hat{\mathtt{P}}$ keeping the other(s) as equations, and Church-Rosser modulo still holds.

7 Preservation of β-Strong Normalisation

In this section we show that $\Lambda_{\hat{\mathtt{P}}}$ enjoys PSN. As before, we shall actually prove PSN for $\{\lambda\mathtt{sub},\mathtt{u},\hat{\mathtt{u}}\}/\Pi$ and then deduce PSN for $\Lambda_{\hat{\mathtt{P}}}$. The proof simply consists in reducing the problem to the following result from [2].

Theorem 4. *The calculus $\{\lambda\mathtt{j},\mathtt{u}\}/\mathtt{P}$ enjoys PSN.*

Since $\lambda\mathtt{sub}$ can be seen as a sub-calculus of $\lambda\mathtt{j}$ (because $\to_{\lambda\mathtt{sub}} \subseteq \to_{\lambda\mathtt{j}}$, see [1]), from Th. 4 we immediately get the following corollary:

Theorem 5. *The calculus* $\{\lambda\mathtt{sub}, \mathtt{u}\}/\mathtt{P}$ *enjoys PSN.*

In order to infer PSN for $\{\lambda\mathtt{sub}, \mathtt{u}, \hat{\mathtt{u}}\}/\Pi$ we need to show that $\equiv_\Pi$ and $\to_{\hat{\mathtt{u}}}$ preserve strong normalisation. The idea is to project reductions of $\{\lambda\mathtt{sub}, \mathtt{u}, \hat{\mathtt{u}}\}/\Pi$ over $\mathtt{dB}$-normal forms, since $\equiv_\Pi$ (resp. $\to_{\hat{\mathtt{u}}}$) collapses on $\equiv_\mathtt{P}$ (resp. $\to_\mathtt{u}$), and then show that this projection preserves strong normalisation. But this is trivial: $\to_\mathtt{dB}$ cannot erase any redex except the one it reduces. For $\to_{\hat{\mathtt{u}}}$ this is given by Lem. 9.2, while for $\to_\mathtt{sub}$ it is given by the the following lemma, where we get $\to^+_{\lambda\mathtt{sub}}$ and not just $\to^*_{\lambda\mathtt{sub}}$.

Lemma 10. *If* $t \to_\mathtt{sub} u$ *then* $\mathtt{dB}(t) \to^+_{\lambda\mathtt{sub}} \mathtt{dB}(u)$.

Proof. Lem. 7 applied to the hypothesis and $t \to^*_\mathtt{dB} \mathtt{dB}(t)$ gives v s.t. $u \to^*_\mathtt{dB} v$ and $\mathtt{dB}(t) \to_\mathtt{sub} v$. We conclude since $v \to^*_\mathtt{dB} \mathtt{dB}(v) = \mathtt{dB}(u)$ and so $\mathtt{dB}(t) \to_\mathtt{sub} v \to^*_\mathtt{dB} \mathtt{dB}(u)$.

Corollary 4. *The* $\{\lambda\mathtt{sub}, \mathtt{u}, \hat{\mathtt{u}}\}/\Pi$*-calculus enjoys PSN.*

Proof. Let $t \in \mathcal{SN}_\beta$ and suppose $t \notin \mathcal{SN}_{\{\lambda\mathtt{sub},\mathtt{u},\hat{\mathtt{u}}\}/\Pi}$. Then, there is an infinite $\{\lambda\mathtt{sub}, \mathtt{u}, \hat{\mathtt{u}}\}/\Pi$-reduction starting at t, and since $\mathtt{dB}$ modulo Π is a trivial well-founded relation, this reduction has necessarily the following form: $t \to^*_{\mathtt{dB}/\Pi} t_1 \to^+_{\{\mathtt{sub},\mathtt{u},\hat{\mathtt{u}}\}/\Pi} t_2 \to^*_{\mathtt{dB}/\Pi} t_3 \to^+_{\{\mathtt{sub},\mathtt{u},\hat{\mathtt{u}}\}/\Pi} t_4 \dots$. By Lem. 6.2, Lem. 10 and Lem. 9.2, we can transform this infinite reduction into an infinite $\{\lambda\mathtt{sub}, \mathtt{u}\}/\mathtt{P}$-reduction starting at t. Since $t \in \mathcal{SN}_{\{\lambda\mathtt{sub},\mathtt{u}\}/\mathtt{P}}$ by Cor. 5, then also $\mathtt{dB}(t) \in \mathcal{SN}_{\{\lambda\mathtt{sub},\mathtt{u}\}/\mathtt{P}}$, so we get a contradiction.

The permutative λ-calculus can be (strictly) simulated into the reduction relation $\{\lambda\mathtt{sub}, \mathtt{u}, \hat{\mathtt{u}}\}/\Pi$ and thus it enjoys PSN.

Corollary 5 (PSN for $\{\beta, \hat{\mathtt{u}}\}/\hat{\mathtt{P}}$). *The permutative* λ*-calculus enjoys PSN, i.e. if* $t \in \mathcal{SN}_\beta$*, then* $t \in \mathcal{SN}_{\{\beta,\hat{\mathtt{u}}\}/\hat{\mathtt{P}}}$.

This last corollary is a generalisation of René David's results [4], where $\hat{\sigma}_1$ is taken from left to right while $\{\hat{\sigma}_2, \widehat{\mathtt{box}}\}$ are taken from right to left.

More generally, consider any orientation of (a subset) of our equations that yields a terminating reduction $\rightsquigarrow$. Then, the system where these equations are replaced by $\rightsquigarrow$ turns out to enjoy PSN. Thus, our result strictly subsumes previous results in the literature [4,7].

To appreciate the power of Cor. 5 note that whenever t is typable with respect to a system $\mathcal{S}$ guaranteing β-strong normalisation (for instance, simple types, intersection types, second-order types) then t is strongly normalising (SN) in $\Lambda_{\hat{\mathtt{P}}}$.

Theorem 6 (SN). *Typability implies strong normalisation.*

8 Conclusions and Future Work

This paper proposes the permutative λ-calculus as a natural generalization of existing λ-calculi for reasoning about permutation of constructors. In all these frameworks permutations are reduction rules, while in $\Lambda_{\hat{\mathtt{P}}}$ they are treated as equations, which is more general and also requires more sophisticated rewriting techniques. The one we use for confluence, based on the new notion of M-developments, is simple and yet powerful: we believe it is interesting by itself. We think that M-developments can also be used for proving meta-confluence of $\Lambda_{\hat{\mathtt{P}}}$.

It would be also interesting to understand if it is possible to state some abstract conditions on *equational* extensions of λ-calculus implying the good behaviour of equations. Indeed, [7] gives sufficient conditions on *reduction systems* extending λ-calculus to guarantee that they enjoy PSN. However, the method in [7] does not seem to be naturally applicable to equational extensions.

References

1. Accattoli, B., Kesner, D.: The Structural λ-Calculus. In: Dawar, A., Veith, H. (eds.) CSL 2010. LNCS, vol. 6247, pp. 381–395. Springer, Heidelberg (2010)
2. Accattoli, B., Kesner, D.: Preservation of strong normalisation modulo permutations for the structural calculus. Submitted to LMCS (2011), https://sites.google.com/site/beniaminoaccattoli/PSN-modulo.pdf
3. Barendregt, H.: The Lambda Calculus: Its Syntax and Semantics, Revised edition. North-Holland (1984)
4. David, R.: A short proof that adding some permutation rules to preserves SN. TCS 412(11), 1022–1026 (2011)
5. de Groote, P.: The Conservation Theorem Revisited. In: Bezem, M., Groote, J.F. (eds.) TLCA 1993. LNCS, vol. 664, pp. 163–178. Springer, Heidelberg (1993)
6. Espírito Santo, J.: Delayed Substitutions. In: Baader, F. (ed.) RTA 2007. LNCS, vol. 4533, pp. 169–183. Springer, Heidelberg (2007)
7. Espírito Santo, J.: A note on preservation of strong normalisation in the λ-calculus. TCS 412(11), 1027–1032 (2011)
8. Girard, J.-Y.: Linear logic. TCS 50 (1987)
9. Hindley, J.R.: Reductions of residuals are finite. Transactions of the American Mathematical Society 240, 345–361 (1978)
10. Kamareddine, F.: Postponement, conservation and preservation of strong normalization for generalized reduction. JLC 10(5), 721–738 (2000)
11. Kesner, D.: A theory of explicit substitutions with safe and full composition. LMCS 5(3:1), 1–29 (2009)
12. Kfoury, A.J., Wells, J.B.: New notions of reduction and non-semantic proofs of beta-strong normalization in typed lambda-calculi. In: LICS, pp. 311–321. IEEE Computer Society Press (1995)
13. Klop, J.-W., van Oostrom, V., van Raamsdonk, F.: Combinatory reduction systems: introduction and survey. TCS 121(1/2), 279–308 (1993)
14. Lengrand, S.: Termination of lambda-calculus with the extra call-by-value rule known as assoc. CoRR, abs/0806.4859 (2008)

15. Lévy, J.-J.: Réductions correctes et optimales dans le lambda-calcul. PhD thesis, Univ. Paris VII, France (1978)
16. Moggi, E.: Computational lambda-calculus and monads. In: LICS, pp. 14–23. IEEE Computer Society Press (1989)
17. Regnier, L.: Une équivalence sur les lambda-termes. TCS 2(126), 281–292 (1994)
18. Sabry, A., Felleisen, M.: Reasoning about programs in continuation-passing style. In: LFP, pp. 288–298. ACM, New York (1992)
19. Terese: Term Rewriting Systems. Cambridge Tracts in Theoretical Computer Science, vol. 55. Cambridge University Press (2003)
20. van Oostrom, V.: Z. Slides, http://www.phil.uu.nl/~oostrom/publication/rewriting.html
21. Espírito Santo, J., Matthes, R., Pinto, L.: Monadic Translation of Intuitionistic Sequent Calculus. In: Berardi, S., Damiani, F., de'Liguoro, U. (eds.) TYPES 2008. LNCS, vol. 5497, pp. 100–116. Springer, Heidelberg (2009)

Automated and Human Proofs in General Mathematics: An Initial Comparison

Jesse Alama[1], Daniel Kühlwein[2], and Josef Urban[2,⋆]

[1] New University of Lisbon
[2] Radboud University Nijmegen

Abstract. First-order translations of large mathematical repositories allow discovery of new proofs by automated reasoning systems. Large amounts of available mathematical knowledge can be re-used by combined AI/ATP systems, possibly in unexpected ways. But automated systems can be also more easily misled by irrelevant knowledge in this setting, and finding deeper proofs is typically more difficult. Both large-theory AI/ATP methods, and translation and data-mining techniques of large formal corpora, have significantly developed recently, providing enough data for an initial comparison of the proofs written by mathematicians and the proofs found automatically. This paper describes such an initial experiment and comparison conducted over the 50000 mathematical theorems from the Mizar Mathematical Library.

1 Introduction: Automated Theorem Proving in Mathematics

Computers are becoming an indispensable part of many areas of mathematics [6]. As their capabilities develop, human mathematicians are faced with the task of steering, comprehending, and evaluating the ideas produced by computers, similar to the players of chess in recent decades. A notable milestone is the automatically found proof of the Robbins conjecture by EQP [8] and its postprocessing into a human-comprehensible proof by ILF [3] and Mathematica [5]. Especially in small equational algebraic theories (e.g., quasigroup theory), a number of nontrivial proofs have been already found automatically [11], and their evaluation, understanding, and automated post-processing is an open problem [18].

This motivates our interest in making ATP useful also for general mathematics, as done by trained mainstream mathematicians, using standard set-theoretical foundations and a large body of general mathematical knowledge. In the recent years, large general mathematical corpora like the Mizar Mathematical Library (MML) and the Isabelle/HOL library are being made available to automated reasoning and AI methods [14,10], leading to the development of automated reasoning techniques working in large theories with many previous theorems, definitions, and proofs that can be reused [16,7,9,17]. A recent evaluation (and tuning) of ATP systems on the MML [15]

⋆ J. Alama was funded by the ESF research project *Dialogical Foundations of Semantics*, FCT LogICCC/0001/2007. D. Kühlwein and J. Urban were funded by the NWO projects *Learning2Reason* and *MathWiki*. We thank P. Rudnicki for providing computing support for the work described here, and the anonymous LPAR referees for valuable comments.

N. Bjørner and A. Voronkov (Eds.): LPAR-18, LNCS 7180, pp. 37–45, 2012.

has shown that the Vampire/SInE system (winner of all CASC LTB competitions) can already re-prove 39% of the MML's 50000 theorems when the necessary premises are precisely selected from the human[1] proofs, and about 14% of the theorems when the ATP is allowed to use the whole available library, leading on average to 40000 premises in such ATP problems. In [1] it is further shown on a subset of 2078 MML problems that re-using (generalizing and learning) the knowledge accumulated in previous proofs can further significantly (currently 30–40%, depending on the learning method) improve the performance of combined AI/ATP systems in large-theory mathematics.

This performance, and the recently developed exact proof analysis for the MML [2], allowed an experiment with finding automatically all proofs in the MML by a combination of learning and ATP methods. This is described in Section 2. The 9141 ATP proofs found automatically were then compared using several metrics to the human proofs. This is described in Section 3 and in Section 4.

2 Finding Proofs in the MML with AI/ATP Support

To create a sufficient body of ATP proofs from the MML, we have conducted a large AI/ATP experiment that makes use of several recently developed techniques and significant computational resources. The basic idea of the experiment is to lift to the whole MML (approximately 50000 theorems and more than 100000 premises) the setting used in [1] for large-theory automated proving of 2078 related MML problems. The setting consists of the following three consecutive steps:

- mining the exact minimal proof dependencies from all human-written MML proofs;
- learning premise selection from the minimal proof dependencies;
- using an ATP (Vampire) to prove new conjectures from the best selected premises.

2.1 Mining the Minimal Dependencies from All Human-Written MML Proofs

This process is in detail described in [2], where it is conducted for the 100 initial articles from the MML. The difficulty (and interestingness) consists in (partially brute-force) minimization of the many proof premises (like typing information) that advanced ITPs like Mizar use implicitly, in order to save their users most of the obvious and tedious steps. Conducting this data-mining requires nontrivial transformation of all MML items (theorems, definitions, etc.) into separate micro-articles, followed by the minimization of the micro-articles' dependencies. This takes several days for all of MML, however the precise information thus obtained gives rise to an unparalleled corpus of data about human-written proofs in the largest available formal body of mathematics. One advantage of this minimization approach over just collecting dependencies from the standard

[1] Mizar proofs are initially human-written, but they are formal and machine-understandable. That allows their automated machine processing and refactoring, which can make them "less human". Yet, we believe that their classification as "human" is mostly correct for our purposes, and that MML/MPTP is probably the most suitable resource today for attempting this initial comparison of ATP and human proofs.

ITP proof objects is that the latter often include many products of "wasteful" (non-minimized) ITP techniques, like congruence closure over all available ground equalities, and exhaustive application of typing hierarchies. In the final account, the days of computation devoted to minimization pay off, by providing more precise advice for proving new conjectures over the whole MML. An approximate estimate of the computational resources taken by this job is about ten days of full (parallel) CPU load (12 hyperthreading Xeon 2.67 GHz cores, 24 GB RAM) of the Mizar server at the University of Alberta. The resulting minimized dependencies for all MML items can be viewed online at a web page[2] presenting them.

2.2 Learning Premise Selection from Proof Dependencies

To learn premise selection from proof dependencies, one characterizes all MMLformulas by suitable features, and feeds them (together with the detailed proof information) to a machine learning system that is trained to advise premises for later conjectures. Formula symbols have been used previously for this task in [14]. Thanks to sufficient hardware being available, we have for the first time included also term features generated by the MaLARea framework, added to it in 2008 [16] for experiments with smaller subsets of MML. Thus, for each MML formula we include as its characterization also all the subterms and subformulas contained in the formula, which makes the learning and prediction more precise. To our surprise, the EPROVER-based [13] utility that consistently numbers all shared terms in all formulas, written for this purpose in 2008 by the last author, scaled without problems to the whole MML, and this feature-generation phase took only minutes. This created over one million learning features that are used to characterize all the mathematics in MMLfor learning. We have also briefly explored using validity in finite models (again, introduced in MaLARea in 2008, building on Pudlák's previous work [12]) as a more semantic way of characterizing formulas. However, this has turned out to be very time-consuming, most likely caused by the LADR-based `clausefilter` utility struggling with evaluating in the models some complicated (containing many quantifiers) mathematical formulas from the MML. Clearly, further optimizations are needed for extracting such semantic characterizations for all of MML. Even without such features, the machine learning was already pushed to the limit. The more advanced kernel-based multi-output ranker developed in [1] turned out to be too slow and memory-exhaustive to handle over one million features and over hundred thousand training examples without further improvements. The SNoW system used in naive Bayes mode took several gigabytes of RAM to train on the data, and on average about a second (ca a day of computation for all of MML) to produce a premise prediction for each MML problem (based always on incremental training[3] on all previous MML proofs characterized by their exact dependencies, and the formula features). It seems that pushing such AI methods to handle, for example, the mathematics from the whole arXiv.org, Elsevier, or Springer corpora (if ever (semi-)formalized), will require some nontrivial scaling up, going for technologies developed by Google and similar

[2] `http://mizar.cs.ualberta.ca/mizar-items`

[3] In the incremental learning mode, the evaluation and training are done at the same time for each example, hence there was no extra time taken by training.

companies for dealing with very large learning tasks. The results of this run are SNoW premise predictions for all of MML, available online as the raw SNoW output at our web page[4], and also postprocessed into corresponding ATP problems (see below).

2.3 Using ATPs to Prove the Conjectures from the Selected Premises

As the MML grows from axioms of set theory to advanced mathematics, it gives rise to a chronological ordering of its theorems. When a new theorem C is conjectured, all the previous theorems and definitions are available as premises, and all the previous proofs are used to learn which of these premises are relevant for C. The SNoW system provides a ranking of all premises, and the best premises are given to an ATP which attempts a proof of C.

There are many ways how to organize several ATP systems to attack C, and how to try with different numbers of the best-ranked premises and with different time limits. Several such policies are implemented in MaLARea, which can run for months on the whole MML while still producing new proofs. Because our main interest here was to get a sufficient body of automatically found proofs for their further evaluation, we have fixed the ATP system to be Vampire (version 1.8), and we have always used 200 best premises and a time limit of 20 seconds. A 12-core 2.67 GHz Xeon server at University of Alberta was used for (parallelized) proving, which took about a day in real time. This has produced 9141 automatically found proofs that we further analyze. The overall success rate is over 18% of theorems proved which is so far the best result on the whole MML, but we have not really focused yet on getting this as high as possible. For example, running Vampire in parallel with both 40 and 200 best recommended premises has been shown to significantly improve the success rate, and a preliminary experiment with the Z3 solver has provided another two thousand proofs from the problems with 200 best premises. Unfortunately, Z3 does not (yet) print the names of premises used in the proofs, so its proofs would not be directly usable for the analysis that is conducted here. When using a large number of premises, an ATP proof can often contain unnecessary premises. To get more closely the set of premises that an ATP actually needs, we always re-run the ATP only with the premises that were used in the first run. For Vampire, such minimization is quite significant. The resulting ATP problems are also available online[5] for further experiments, as well as all the 9141 proofs found.[6]

3 Proof Metrics

We thus have, for 9141 Mizar theorems ϕ, the set of premises that were used in the (minimized) ATP proof of ϕ. Each ATP proof was found completely independently of its Mizar proof, i.e., no information (e.g., about the premises used) from the Mizar proof was transferred to the ATP proof.[7] This gives us a notion of dependency for

[4] http://mizar.cs.ualberta.ca/~mptp/proofcomp/snow_predictions.tar.gz
[5] http://mizar.cs.ualberta.ca/~mptp/proofcomp/advised200f1.tar.gz
[6] http://mizar.cs.ualberta.ca/~mptp/proofcomp/proved200f1min.tar.gz
[7] The ATP proofs are however always based on the same state of previous theory and proof knowledge. This could be further relaxed in future experiments.

Mizar theorems, derived from an ATP. From the Mizar proof dependency analysis we also know precisely what Mizar items are needed for a given Mizar (human-written) proof to be successful.

Definition 1. *For a Mizar theorem ϕ, let $P_{\mathrm{H}}(\phi)$ be the minimal set of premises needed for the success of the human proof of ϕ. Let $P_{\mathrm{A}}(\phi)$ be the set of premises used by an ATP to prove ϕ.*

This gives rise to the notions of "immediate dependence" and "indirect dependence" of one Mizar item a upon another Mizar item b:

Definition 2. *For Mizar items a and b, $a <_1 b$ means that a immediately depends on b ($b \in P_{\mathrm{H}}(a)$). Let $<$ be the transitive closure of $<_1$, $\leq$ its reflexive version, and let $P^*_{\mathrm{H}}(a) := \{b : b < a\}$. For a set S of items, let $P^*_{\mathrm{H}}(S) := \{b : \exists a \in S : b \leq a\}$.*

While theoretically, there are multiple versions of $<_1$ and $<$ induced by different (ATP, Mizar) proofs, unless we explicitly state otherwise, these relations will always refer to the dependencies derived from the Mizar proofs. The pragmatic reason is that we do not have an ATP proof for all Mizar items,[8] and hence we do not have the full dependency graph induced by ATP proofs. Also, the way ATP proofs were produced was by always relying on the previous Mizar theorems and dependency data, therefore it makes sense to also use the Mizar data for the transitive closure.

We now define two metrics (and their transitive, human, and ATP versions) D (*Dependencies*) and L (*Length*) measuring the complexity of proofs of Mizar theorems. The human alternative of D just (recursively) counts all proof dependencies, based on the full Mizar dependency graph. The ATP alternative uses the ATP proof, and its transitive version recurses using the (full) Mizar dependency graph.

Definition 3. *For each Mizar item a, let $D_{\mathrm{H}}(a) := |P_{\mathrm{H}}(a)|$ and $D^*_{\mathrm{H}}(a) := |P^*_{\mathrm{H}}(a)|$. For a set S of items, let $D^*_{\mathrm{H}}(S) := |P^*_{\mathrm{H}}(S)|$. For each ATP-proved theorem a, let $D_{\mathrm{A}}(a) := |P_{\mathrm{A}}(a)|$, and $D^*_{\mathrm{A}}(a) := D^*_{\mathrm{H}}(P_{\mathrm{A}}(a))$.*

The second metric L adds weighting by (recursive) proof complexity, which is for the Mizar proofs computed using the assumption that the Mizar weak refutational checker enforces a relatively uniform degree of derivational complexity on all Mizar proof steps, which roughly correspond to proof lines in Mizar formalizations. For the ATP version, we make a similar assumption that the complexity of ATP proof steps is roughly uniform.[9] For the comparison with human proofs, we first need to define a conversion ratio $c_{A/H}$ between the number of ATP inference lines and the corresponding number of Mizar proof lines. This is pragmatically estimated as the average of such ratios for all the proofs where the ATP used the same premises as the Mizar proof. The actual value computed (based on 1223 proofs where $P_{\mathrm{A}}(a) = P_{\mathrm{H}}(a)$) is $c_{A/H} = 81.99$. Formally:

[8] We have limited the ATP experiment to Mizar *theorems*, so even with perfect ATP success rate we would still miss for example all ATP dependencies of Mizar definitions, that often require proofs of existence, uniqueness, etc.

[9] The precision of such metrics could be further improved, for example by expanding Vampire proofs into the (really uniform) detailed proof objects developed for Otter/Prover9/IVY.

Definition 4. *For a Mizar-proved item a, let $L_H(a)$ be the number of Mizar lines of code used to prove a (direct Mizar proof length). For each ATP-proved item a, let $L^0_A(a)$ be the number of steps in the ATP proof. Let $E_{H=A} := \{a : P_A(a) = P_H(a)\}$ (items whose ATP and Mizar proofs use the same premises). Let $c_{A/H} := 1/|E_{H=A}| * \sum_{a \in E_{H=A}} L^0_A(a)/L_H(a)$ (the length conversion ratio). Finally, we define the normalized ATP proof length as $L_A(a) := L^0_A(a)/c_{A/H}$. For a set of items S, let again $L_H(S) := \sum_{a \in S} L_H(a)$.*

Definition 5. *For a Mizar theorem a we define $L^*_H(a) := L_H(a) + L_H(P^*_H(a))$. If we have an ATP proof, we define $L^*_A(a) := L_A(a) + L_H(P^*_H(P_A(a)))$.*

The reason for using $L_H()$ and $P^*_H()$ in the recursive part of $L^*_A()$ is again the fact that we only have the complete line count information for the Mizar proofs. Note that both in D^* and in L^* we always count any lemma on the transitive proof path exactly once. We believe that this approach captures the mathematician's intuition of *proof complexity* as the *set* of "the proofs that need to be understood" rather than as their multiset. This could be further explored by various cognitive experiments.

4 Evaluation

The metrics developed above were used for an initial comparison of the Mizar and ATP proofs. The detailed evaluation data corresponding to this section are available online.[10] First we analyze the data based on the relation between $P_H()$ and $P_A()$. For each Mizar theorem ϕ that can be proved by an ATP, we have either $P_H(\phi) = P_A(\phi)$, $P_H(\phi) \subset P_A(\phi)$, $P_A(\phi) \subset P_H(\phi)$, or neither set is included in the other. Let us say that two sets A and B are *orthogonal* if neither $A \subseteq B$, nor $B \subseteq A$. The statistics is given in Table 1.

Table 1. Premise statistics for the categories

Category	Cases	Max A	Min A	Avg A	Max H	Min H	Avg H	Max $\lvert H - A\rvert$	Avg $\lvert H - A\rvert$
$P_H(a) = P_A(a)$	1223	7	0	2.18	7	1	2.11	0	0
$P_A(a) \subset P_H(a)$	1980	12	0	2.20	59	1	5.58	58	3.40
$P_H(a) \subset P_A(a)$	386	89	1	6.24	10	1	2.41	83	3.88
Orthogonal	5552	63	1	5.22	58	1	6.33	60	3.86

While the *orthogonal* category is largest as was expected, it is surprising to see more than 10% of the proofs to be the same according to the D metric. It is even more surprising to see that 1980 ATP proofs (21.66%) are shorter according to the D metric. An initial analysis suggested (at least) the following explanations:

– The ATP is naturally oriented towards as short proofs as possible. Getting involved proofs with many premises is hard, and it may well be the main reason of ATP failure outside the 9141 proved theorems.

[10] http://mizar.cs.ualberta.ca/~mptp/proofcomp/metrics_evaluation.xls

- In many cases, a human formalizer can overlook the fact that the same or very similar theorem is already in the library.[11] An example is the theorem LOPBAN_3:24[12] which required a 20-line proof in Mizar, but the ATP found an earlier more general theorem BHSP_4:3 that (using additional typing information) provides an almost immediate proof.
- ATPs work in untyped first-order logic, and they are not constrained by the Mizar's (and other ITPs') requirement that all types should be inhabited. For example, Mizar proof checking of GOEDELCP:1[13] fails if two type non-emptiness declarations are removed, because the formula is no longer well-typed. The ATP proof however does not need any of them.

An interesting case is when the ATP finds an inventive way how to re-use previous lemmas. Sometimes enough knowledge about advanced concepts is already developed that can be used for their quite simple ("algebraic") manipulation, abstracting from their definitions. An example is COMSEQ_3:40[14], proving the relation between the limit of a complex sequence and its real and imaginary parts. The human proof expands the definitions (finding a suitable n for a given ϵ). The ATP just notices that this kind of groundwork was already done in a "similar" case COMSEQ_3:39[15], and notices the "similarity" (algebraic simplification) provided by COMPLEX1:28[16]. Such manipulations can be used (if noticed!) to avoid the "hard thinking" about the epsilons in the definitions.

4.1 Comparing Weights

For a Mizar theorem ϕ, a large difference between $L^*_{\mathrm{H}}(\phi)$ and $L^*_{\mathrm{A}}(\phi)$ is a sign that the ATP of ϕ is importantly different from the human Mizar proof of ϕ. The Table 2 shows that with exception of the $P_{\mathrm{H}}(a) = P_{\mathrm{A}}(a)$ case, which we used to define $c_{A/H}$, the ATP proofs have on average higher recursive complexity L^* than the corresponding human proofs. Again, we have found several explanations:

- Some cases are due to the failure in minimization of the ATP proofs. For example, the ATP proof of FUNCT_7:20[17] reports 40 premises and 178715 ATP (non-normalized) proof steps, largely coming from recent addition of BDDs to Vampire.

[11] From this point of view, this analysis is conducted at the right time, because the ATP service is starting to be used by authors, and such simple repetitions will be prevented by it.

[12] http://mizar.cs.ualberta.ca/~MPTP/cgi-bin/browserefs.cgi?refs=t24_lopban_3 The theorem says that the partial-sums operator on normed space sequences commutes with multiplication by a scalar.

[13] http://mizar.cs.ualberta.ca/~MPTP/cgi-bin/browserefs.cgi?refs=t1_goedelcp

[14] http://mizar.cs.ualberta.ca/~MPTP/cgi-bin/browserefs.cgi?refs=t40_comseq_3

[15] http://mizar.cs.ualberta.ca/~MPTP/cgi-bin/browserefs.cgi?refs=t39_comseq_3

[16] http://mizar.cs.ualberta.ca/~MPTP/cgi-bin/browserefs.cgi?refs=t28_complex1

[17] http://mizar.cs.ualberta.ca/~MPTP/cgi-bin/browserefs.cgi?refs=t20_funct_7

Table 2. Recursive line count/proof step statistics (L^*) for the categories

Category	Cases	Max A	Min A	Avg A	Max H	Min H	Avg H	Max $\lvert H - A\rvert$	Avg $\lvert H - A\rvert$
$P_{\mathrm{H}}(a) = P_{\mathrm{A}}(a)$	1223	140176	7	7390.77	140438	1	7385.06	6210	0
$P_{\mathrm{A}}(a) \subset P_{\mathrm{H}}(a)$	1980	40653	26	7373.31	32652	1	6167.73	40626	1220.52
$P_{\mathrm{H}}(a) \subset P_{\mathrm{A}}(a)$	386	162308	9	14155.3	162532	3	14768.3	35536	632.329
Orthogonal	5552	139935	13	9893.04	140172	3	9828.81	75114	910.744

– Most of the cases again seem to be due to the ATPs tendency to get a short proof by advanced lemmas, rather than getting into longer proofs by expanding the definitions. The lemmas typically recursively use the basic definitions anyway, and their line complexity is then a net contribution to the ATP proof's recursive complexity.

5 Conclusion

While ATPs in general large-theory formal mathematics are becoming clearly useful, an initial proof analysis using quite straightforward metrics has not yet found any highly surprising ATP proofs. Clearly, the general large-theory mathematical setting is still quite far from producing automated proofs of the order of complexity that some specialized algebraic theories enjoy. On the other hand, the ATPs have found a surprising number of proofs that are shorter than the mathematicians' version, and the ATP's ideas are very often relevant. Unlike humans, the combined AI/ATP stack learns new lemmas and new proofs immediately, and this results in their more extensive use and significantly higher value of L^*. An ATP working in unsorted FOL can sometimes find proofs that, in some sense, get to the "mathematical heart" of a theorem without first going through the syntactic hoops of ensuring that terms have suitable sorts. The tools produced for our experiments can produce information that is useful for maintainers of large formal libraries. We found cases where an ATP was able to find a significantly shorter proof—sometimes employing only one premise—compared to a human proof. At times, such highly efficient ATP proofs were due to duplication in the library or failure to use a generalization to prove a special case. Finally, our work comparing different proofs could provide a practical "test bed" for theoretical criteria of proof identity [4].

References

1. Alama, J., Kühlwein, D., Tsivtsivadze, E., Urban, J., Heskes, T.: Premise selection for mathematics by corpus analysis and kernel methods. CoRR, abs/1108.3446
2. Alama, J., Mamane, L., Urban, J.: Dependencies in formal mathematics. CoRR, abs/1109.3687 (2011), http://arxiv.org/abs/1109.3687
3. Dahn, I.: Robbins algebras are Boolean: A revision of McCune's computer-generated solution of Robbins problem. Journal of Algebra 208, 526–532 (1998)
4. Dosen, K.: Identity of proofs based on normalization and generality. Bulletin of Symbolic Logic 9, 477–503 (2003)
5. Fitelson, B.: Using Mathematica to understand the computer proof of the Robbins Conjecture. Mathematica In Education and Research 7(1) (1998)

6. Hales, T.: Mathematics in the age of the Turing Machine. Lecture Notes in Logic in Commemoration of the Centennial of the Birth of Alan Turing (to appear, 2012)
7. Hoder, K., Voronkov, A.: Sine Qua Non for Large Theory Reasoning. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS, vol. 6803, pp. 299–314. Springer, Heidelberg (2011)
8. McCune, W.W.: Solution of the Robbins Problem. Journal of Automated Reasoning 19(3), 263–276 (1997)
9. Meng, J., Paulson, L.C.: Lightweight relevance filtering for machine-generated resolution problems. J. Applied Logic 7(1), 41–57 (2009)
10. Paulson, L.C., Blanchette, J.: Three years of experience with Sledgehammer, a practical link between automated and interactive theorem provers. In: 8th IWIL (2010)
11. Phillips, J.D., Stanovský, D.: Automated Theorem Proving in Loop Theory. In: Sutcliffe, G., Colton, S., Schulz, S. (eds.) ESARM. CEUR Workshop Proceedings, vol. 378, pp. 42–53. CEUR-WS.org (2008)
12. Pudlák, P.: Semantic selection of premisses for automated theorem proving. In: Sutcliffe, G., Urban, J., Schulz, S. (eds.) ESARLT. CEUR Workshop Proceedings, vol. 257. CEUR-WS.org (2007)
13. Schulz, S.: E – a brainiac theorem prover. J. of AI Communications 15(2-3), 111–126 (2002)
14. Urban, J.: MPTP 0.2: Design, implementation, and initial experiments. J. Autom. Reasoning 37(1-2), 21–43 (2006)
15. Urban, J., Hoder, K., Voronkov, A.: Evaluation of Automated Theorem Proving on the Mizar Mathematical Library. In: Fukuda, K., van der Hoeven, J., Joswig, M., Takayama, N. (eds.) ICMS 2010. LNCS, vol. 6327, pp. 155–166. Springer, Heidelberg (2010)
16. Urban, J., Sutcliffe, G., Pudlák, P., Vyskočil, J.: MaLARea SG1–Machine learner for automated reasoning with semantic guidance. In: IJCAR, pp. 441–456 (2008)
17. Urban, J., Vyskočil, J., Štěpánek, P.: MaLeCoP Machine Learning Connection Prover. In: Brünnler, K., Metcalfe, G. (eds.) TABLEAUX 2011. LNCS, vol. 6793, pp. 263–277. Springer, Heidelberg (2011)
18. Vyskočil, J., Stanovský, D., Urban, J.: Automated Proof Compression by Invention of New Definitions. In: Clarke, E.M., Voronkov, A. (eds.) LPAR-16 2010. LNCS, vol. 6355, pp. 447–462. Springer, Heidelberg (2010)

Lazy Abstraction with Interpolants for Arrays

Francesco Alberti[1], Roberto Bruttomesso[2], Silvio Ghilardi[2], Silvio Ranise[3], and Natasha Sharygina[1]

[1] Università della Svizzera Italiana, Lugano, Switzerland
[2] Università degli Studi di Milano, Milan, Italy
[3] FBK-Irst, Trento, Italy

Abstract. Lazy abstraction with interpolants has been shown to be a powerful technique for verifying imperative programs. In presence of arrays, however, the method shows an intrinsic limitation, due to the fact that successful invariants usually contain universally quantified variables, which are not present in the program specification. In this work we present an extension of the interpolation-based lazy abstraction in which arrays of unknown length can be handled in a natural manner. In particular, we exploit the Model Checking Modulo Theories framework, to derive a backward reachability version of lazy abstraction that embeds array reasoning. The approach is generic, in that it is valid for both parameterized systems and imperative programs. We show by means of experiments that our approach can synthesize and prove universally quantified properties over arrays in a completely automatic fashion.

1 Introduction

The automatic verification of software is a long standing scientific challenge. A promising line of research is that in which Model Checking techniques are employed to automatically traverse the state-space of a program, and check it with respect to a user-specified property. Since the problem is undecidable, complete and fully automatic techniques cannot exist and the programmer must provide additional annotations describing, for instance, loop invariants. It is well-known that the task of providing such annotations is far from trivial. In order to significantly alleviate the annotation burden, it is crucial to employ abstraction techniques. For example, Predicate Abstraction [13], the CEGAR approach [4], or Lazy Abstraction [16] have been shown successful and are nowadays employed in many state-of-the-art software verification tools. In particular, Lazy Abstraction is capable of tuning the abstraction by using different degrees of precision for different parts of the program by keeping track of both the control-flow graph, which describes *how* the program locations are traversed, and the data-flow, which describes *what* holds at a program location. The control-flow is represented explicitly, while the data-flow is symbolically encoded with quantifier-free first-order formulæ and it is subjected to abstraction. The procedure is therefore based on a CEGAR loop in which the control-flow graph is iteratively unwinded, and the data in the newly explored locations is overapproximated. When reaching an error location, if the path is spurious—i.e. the quantifier-free formula

N. Bjørner and A. Voronkov (Eds.): LPAR-18, LNCS 7180, pp. 46–61, 2012.

representing the manipulations of the data along the path is unsatisfiable, the abstraction along the path is refined. In state-of-the-art methods, this is done by means of interpolants [15,21]. The procedure terminates when a non-spurious path is found, or when reaching an inductive invariant.

When arrays come into the picture the situation is complicated by at least two problems. First, the need of handling quantified formulæ (as opposed to just quantifier-free) to take care of meaningful array properties; e.g., a typical post-condition of a sorting algorithm is the following universally quantified formula:

$$\forall i, j.\ (0 \leq i < j \leq a.length) \Rightarrow a[i] \leq a[j],$$

expressing the fact that the array a is sorted, where $a.length$ represents the *symbolic* size of a. Second, the difficulty of computing quantifier-free interpolants. In [18], it is shown that quantifiers must occur in interpolants of quantifier-free formulæ for the "standard" theory of arrays.

This paper contributes a new verification approach that addresses the above problems. It redefines the lazy abstraction method based on interpolation (which is known to be one of the most effective approaches in program verification) and makes it possible to reason about arrays of unknown length. For that, it exploits the framework behind the Model Checking Modulo Theory approach (MCMT) [10,11]. The reasoning about arrays and the corresponding quantified formula is performed by means of the symbolic backward reachability algorithm now extended with the interpolation-based abstraction refinement techniques. This combination is able to generate the quantified predicates required for the synthesis of the quantified inductive invariants needed to establish the validity of the program assertions. Notably, our abstraction-based approach can be applied to enhance the verification of array-based systems (a wide class of infinite-state systems currently handled by MCMT). We implemented the new approach and verified various common-use programs over arrays.

The paper is organized as follows. Section 2 recalls basic notions about array-based systems as used in MCMT and demonstrates how sequential programs can be specified using this model. Section 3 introduces the new lazy abstraction approach and discusses its completeness and termination. Experiments are presented in Section 4. We conclude in Section 5. Proofs of claims made within the paper are worked out in the online available extended version [1].

Related Work. The work described in this paper can be considered as part of the broad line of research in model-checking for infinite state systems that makes use of abstraction-refinement techniques to cope with the infinite search space [4,13,16]. A challenging task in this setting is to find the right predicates to ensure convergence; these predicates may be extracted, e.g., from the proof of unsatisfiability of an infeasible abstract path [15,21]. When arrays come into the picture the situation is complicated by the need of using quantifiers to express meaningful properties (such as "sortedness"). Earlier work in predicate abstraction approached this issue by using *Skolem constants* [8], *indexed predicates* [24], or *range predicates* [17]. Approaches based on templates [25] may

synthesize more expressive formulæ, but they require manual specification of templates and predicates. To the best of our knowledge, the closest related work to ours is that of [23], where a backward reachability procedure for universally quantified assertions over arrays is described. As in our approach the procedure visits backward the set of unsafe states to find an intersection with the initial ones, by performing the coarsest possible abstraction first. However in [23] the computed abstraction is then refined with predicates obtained by simulating the "pre" operator on a spurious trace, and by performing classical predicate abstraction (requiring injection of, in the worst case, exponentially many bound constraints between indexes), whereas in our approach we achieve refinement by means of interpolants.

Proving properties over arrays has also been extensively studied in the context of abstract domains other than predicate abstraction. The approaches of [5,6,12,14], for example, follow a line of research in which arrays are divided into *segments*, based on the access and write operations in the programs. Several techniques are employed to avoid the combinatorial explosion, e.g., by means of the introduction of a suitable widening operator. These approaches have been shown to be useful even at an industrial-scale level [5] to automatically infer a wide range of properties. The goal of our approach is to automatically verify that the program satisfy expressive properties.

A further promising direction of research relies on saturation-based theorem-provers [19,20] to generate invariants over arrays. These approaches may in principle produce more expressive invariants than ours, but they require, on the other hand, to instruct the prover with axioms for handling arithmetic. In our setting, instead, we use SMT techniques to take care of the necessary arithmetic operations.

Backward reachability of array-based systems, implemented in the tool MCMT, has been successfully used for checking the safety of several classes of distributed algorithms [10]. The work in this paper shows that MCMT combined with Lazy Abstraction can also be used for verifying expressive properties of sequential programs manipulating arrays. We also characterize when our method behaves as a decision procedure for establishing the safety of classes of array-based systems that cover existing results (e.g., [7]).

2 Background Notions on MCMT

We assume the usual syntactic and semantic notions of many-sorted first-order logic with equality. We use lower-case latin letters $x, a, i, e, \ldots$ for free variables; for tuples of free variables we use underlined letters $\underline{x}, \underline{a}, \underline{i}, \underline{e}, \ldots$ or bold face letters like $\mathbf{a}, \mathbf{v}, \ldots$. With $E(\underline{x})$ we denote that the syntactic expression (term, formula, tuple of terms or of formulæ) E contains at most the free variables taken from the tuple $\underline{x}$. According to [22], a theory T is a pair $(\Sigma, \mathcal{C})$, where Σ is a signature and $\mathcal{C}$ is a class of Σ-structures; the structures in $\mathcal{C}$ are called the models of T. A Σ-formula φ is T-satisfiable if there exists a Σ-structure $\mathcal{M}$ in $\mathcal{C}$ such that φ is true in $\mathcal{M}$ under a suitable assignment to the free variables

of φ (in symbols, $\mathcal{M} \models \varphi$); it is T-valid (in symbols, $T \models \varphi$) if its negation is T-unsatisfiable. Two formulæ φ_1 and φ_2 are T-equivalent if $\varphi_1 \leftrightarrow \varphi_2$ is T-valid; ψ_1 T-entails ψ_2 (in symbols, $\psi_1 \models_T \psi_2$) iff $\psi_1 \rightarrow \psi_2$ is T-valid. The satisfiability modulo the theory T ($SMT(T)$) problem amounts to establishing the T-satisfiability of quantifier-free Σ-formulæ. A theory T has *quantifier-free interpolation* iff there exists an algorithm that, given two quantifier free formulæ ϕ, ψ such that $\phi \wedge \psi$ is T-unsatisfiable, returns a formula θ such that: (i) $\phi \models_T \theta$; (ii) $\theta \wedge \psi$ is T-unsatisfiable; (iii) only the free variables common to ϕ and in ψ occur in θ.

Array-Based Transition Systems and their Safety. We briefly recall some of the notions underlying the framework of MCMT; for an extensive discussion, the reader is pointed to [10]. Array-based systems are a particular class of guarded assignment systems whose state variables comprise arrays. They are represented symbolically using certain classes of formulæ and are endowed with theories specifying the algebraic structures of the indexes and elements of arrays. Roughly, the input language of MCMT for specifying array-based systems can be seen as a parameterized extension of the one used by UCLID (http://www.cs.cmu.edu/~uclid). Formally, it is a sub-set of multi-sorted first-order logic extended with theories. In particular, we assume a (mono-sorted) theory $T_I = (\Sigma_I, \mathcal{C}_I)$ for indexes of arrays and a multi-sorted theory $T_E = (\Sigma_E, \mathcal{C}_E)$ for the elements of the arrays. The unique sort of T_I is called `INDEX` and a sort of T_E is called $\texttt{ELEM}_\ell$, where ℓ ranges over a given (finite) set. We also assume that the $SMT(T_I)$**- and** $SMT(T_E)$**-problems are decidable** and that T_I **and** T_E **have quantifier-free interpolation** (further hypotheses will be discussed below in connection to specific model-checking features).

The theory $A_I^E = (\Sigma, \mathcal{C})$, specifying the algebraic structures of the array state variables manipulated by an array-based system is obtained by "composing" T_I and T_E as follows. The sort symbols of A_I^E are `INDEX`, $\texttt{ELEM}_\ell$, and $\texttt{ARRAY}_\ell$, its signature Σ contains all the symbols in the (disjoint) union $\Sigma_I \cup \Sigma_E \cup \{\_[\_]_\ell\}_\ell$ where $\_[\_]_\ell : \texttt{ARRAY}_\ell \times \texttt{INDEX} \rightarrow \texttt{ELEM}_\ell$ are the usual "read" operations of an array on a given cell, and a structure $\mathcal{M}$ is in the class $\mathcal{C}$ of the models of A_I^E when (i) the restrictions of $\mathcal{M}$ to Σ_I, Σ_E are models of T_I, T_E, respectively, (ii) the sorts $\texttt{ARRAY}_\ell$ are interpreted as (total) functions from $\texttt{INDEX}^{\mathcal{M}}$ to $\texttt{ELEM}_\ell^{\mathcal{M}}$, and (iii) the operations $\_[\_]_\ell$ are interpreted as function applications. In the following, the subscript ℓ will be omitted to simplify notation.

In this paper, to simplify technicalities, we adopt the following variant of the notion of an array-based system [10]. An **array-based system (for** T_I, T_E**)** is a pair $\mathcal{S} = \langle \mathbf{v}, \{\tau_h\}_h \rangle$, where $\mathbf{v} = \mathbf{a}, \mathbf{c}, \mathbf{d}$ is the tuple of *system variables* and is such that

- the tuple $\mathbf{a} = a_0, \ldots, a_s$ contains variables of sort `ARRAY`;
- the tuple $\mathbf{c} = c_0, \ldots, c_t$ contains variables of sort `INDEX` (called, *counters*);
- the tuple $\mathbf{d} = d_0, \ldots, d_u$ contains variables of sort `ELEM` (called, *simple variables*).

All variables are sorted, e.g., for $\mathbf{a}$, this means that to each $i = 0, \ldots, t$ is assigned some ℓ so that a_i is of type $\texttt{ARRAY}_\ell$. The variable d_0 ranges over a finite set $\{l_0, ..., l_n\}$ of *program locations* and is usually denoted with pc (short for *program counter*) instead of d_0. Among the program locations, we shall distinguish an *initial* location l_I and an *error* location l_E. It is assumed that the *initial state* of the array-based system $\mathcal{S}$ is represented by the formula $I(\mathbf{v}) := (pc = l_I)$ and the *error state* by the formula $U(\mathbf{v}) := (pc = l_E)$.

It is still possible to specify distributed algorithms considered in [10] using the notion of array-based systems introduced above. In fact, although using a program counter may not make sense for such systems, one can nevertheless use a trivial program counter with three locations only: the initial location l_I, the error location l_E, and a "standard" location l_S for the body of the distributed algorithm. Thus, the lazy abstraction technique that we are going to describe below can also be applied, without modifications, to the verification of distributed systems.

The τ_h's are **guarded assignments in functional form**. To precisely specify what this means, we need to introduce the following conventions and definitions. The symbols e range over variables of a sort $\texttt{ELEM}$ in Σ_E while i, j, k, z range over variables of sort $\texttt{INDEX}$. Notation $\mathbf{a}[\underline{i}]$ abbreviates $a_1[i_1], \ldots, a_s[i_1], \ldots, a_s[i_n]$ for a tuple $\underline{i} \equiv i_1, \ldots, i_n$ of variables of sort $\texttt{INDEX}$. Expressions of the form $\phi(\underline{i}, \underline{e}), \psi(\underline{i}, \underline{e})$ (possibly sub/super-scripted) denote *quantifier-free* $(\Sigma_I \cup \Sigma_E)$*-formulæ in which at most the variables* $\underline{i} \cup \underline{e}$ *may occur*. Furthermore, $\phi(\underline{i}, \underline{t}/\underline{e})$ (or simply $\phi(\underline{i}, \underline{t})$) abbreviates the substitution of the Σ-terms $\underline{t}$ for the variables $\underline{e}$. Thus, for instance, $\phi(\underline{i}, \mathbf{a}[\underline{i}], \mathbf{c}, \mathbf{d})$ denotes the formula obtained by replacing $\underline{e}, \underline{j}, \underline{e}'$ with $\mathbf{a}[\underline{i}], \mathbf{c}, \mathbf{d}$ respectively in the quantifier-free formula $\phi(\underline{i}, \underline{e}, \underline{j}, \underline{e}')$. A formula $\forall \underline{i}.\ \phi(\underline{i}, \mathbf{a}[\underline{i}], \mathbf{c}, \mathbf{d})$ is a $\forall^I$*-formula*, one of the form $\exists \underline{i}.\ \phi(\underline{i}, \mathbf{a}[\underline{i}], \mathbf{c}, \mathbf{d})$ is an $\exists^I$*-formula*, and a sentence $\exists \mathbf{a}\, \exists \mathbf{c}\, \exists \mathbf{d}\, \exists \underline{i} \forall \underline{j}.\ \psi(\underline{i}, \underline{j}, \mathbf{a}[\underline{i}], \mathbf{a}[\underline{j}], \mathbf{c}, \mathbf{d})$ is an $\exists^{A,I} \forall^I$*-sentence*. A *guarded assignment in functional form* is a formula of the form

$$\exists \underline{k} \left(\begin{array}{l} \phi_L(\underline{k}, \mathbf{a}[\underline{k}], \mathbf{c}, \mathbf{d}) \ \wedge\ \mathbf{a}' = \lambda j.\ G(\underline{k}, \mathbf{a}[\underline{k}], \mathbf{c}, \mathbf{d}, j, a[j]) \ \wedge \\ \qquad \wedge\ \mathbf{c}' = H(\underline{k}, \mathbf{a}[\underline{k}], \mathbf{c}, \mathbf{d}) \ \wedge\ \mathbf{d}' = K(\underline{k}, \mathbf{a}[\underline{k}], \mathbf{c}, \mathbf{d})) \end{array} \right) \tag{1}$$

where $G = G_0, \ldots, G_s$, $H = H_0, \ldots, H_t$, $K = K_0, \ldots, K_u$ are tuples of case-defined functions (roughly, these can be thought of as nested if-then-else expressions, see [10] for a precise definition). As usual, $\mathbf{a}', \mathbf{c}', \mathbf{d}'$ are renamed copies of the $\mathbf{a}, \mathbf{c}, \mathbf{d}$, denoting the values of the state variables immediately after the execution of the guarded assignment. We assume that the guard ϕ_L of a guarded assignment in functional form (1) always contains a conjunct of the form $pc = l$ and that the update function K_0 is of the form $pc = l'$. In this way, we have mappings from guarded assignments and locations: if the guarded assignment is named τ, the locations l and l' are called the *source* and the *target* locations of τ and are denoted by $src(\tau)$ and $trg(\tau)$, respectively.

The array-based system
$\mathcal{S} = \langle \mathbf{v}, \{\tau_h\}_h \rangle$ is *safe* iff the formulæ

$$I(\mathbf{v}^{(n)}) \wedge \left(\bigvee_h \tau_h(\mathbf{v}^{(n)}, \mathbf{v}^{(n-1)}) \right) \wedge \cdots \wedge \left(\bigvee_h \tau_h(\mathbf{v}^{(1)}, \mathbf{v}^{(0)}) \right) \wedge U(\mathbf{v}^{(0)}) \tag{2}$$

```
   function find ( int a[ ] , int n ) {
1    c = 0;
2    while (c < a.length ∧ a[c] ≠ n) c = c + 1;
3    if ( c ≥ a.length ∧ ∃x.(x ≥ 0 ∧ x < a.length ∧ a[x] = n) )
4      ERROR;
   }
```

Fig. 1. Pseudo-code for the function `find`

are A_I^E-unsatisfiable for $n \geq 0$, where $\mathbf{v}^{(0)}, \ldots, \mathbf{v}^{(n)}$ are renamed copies of $\mathbf{v}$ (at time stamps $0, ..., n$). (Recall that, by assumption, $I(\mathbf{v}) := (pc = l_I)$ and $U(\mathbf{v}) := (pc = l_E)$.) If there exists a value of n for which (2) is A_I^E-satisfiable, then this means that there exists an execution of $\mathcal{S}$ starting in an initial state and ending in an error state.

Notice that, although terms of the form $\mathbf{a}[\mathbf{c}]$ *are not allowed* in formula (1), this is without loss of generality. In fact, any formula $\psi(\cdots \mathbf{a}[\mathbf{c}] \cdots)$ containing such terms can be rewritten to $\exists \underline{j}(\underline{j} = \mathbf{c} \wedge \psi(\cdots \mathbf{a}[\underline{j}] \cdots))$ by using (fresh) existentially quantified variables $\underline{j}$ of sort `INDEX`. (Below, for the sake of brevity and only when discussing examples, we will write $\psi(\cdots \mathbf{a}[\mathbf{c}] \cdots)$ in place of $\exists \underline{j}(\underline{j} = \mathbf{c} \wedge \psi(\cdots \mathbf{a}[\underline{j}] \cdots))$.) Interestingly, this syntactic restriction inherited from the specification language underlying MCMT inspired us an heuristic to abstract away counters dereferencing arrays and replace them with universally quantified variables of sort `INDEX` so as to synthesize universally quantified candidate invariants. Such heuristics, called *term abstraction*, will be described in Section 4.

Example 1. We illustrate how to encode the function `find` in Fig. 1 as an array-based system. The theory T_I is linear integer arithmetic (but notice that integer difference logic suffices), enriched with a constant `a.length`; the theory T_E has one sort constrained to be linear integer arithmetic enriched with a constant `n` (again, a very small fragment suffices) and one sort constrained to be the enumerated datatype theory of the set of locations $\{1, 2, 3, 4\}$ (where $l_I = 1$ and $l_E = 4$). The tuple $\mathbf{a}$ of array state variables contains only `a`, `c` is the unique counter, and `pc` is the only simple variable. The following five transitions specify the instructions of `find` (for simplicity, we omit mentioning identical updates):

$$\tau_1 \equiv \mathtt{pc} = 1 \wedge \mathtt{pc}' = 2 \wedge \mathtt{c}' = 0$$
$$\tau_2 \equiv \mathtt{pc} = 2 \wedge \mathtt{c} < \mathtt{a.length} \wedge \mathtt{a[c]} \neq \mathtt{n} \wedge \mathtt{c}' = \mathtt{c} + 1$$
$$\tau_3 \equiv \mathtt{pc} = 2 \wedge \mathtt{c} \geq \mathtt{a.length} \wedge \mathtt{pc}' = 3$$
$$\tau_4 \equiv \mathtt{pc} = 2 \wedge \mathtt{a[c]} = \mathtt{n} \wedge \mathtt{pc}' = 3$$
$$\tau_5 \equiv \mathtt{pc} = 3 \wedge \mathtt{c} \geq \mathtt{a.length} \wedge \exists x.\ (x \geq 0 \wedge x < \mathtt{a.length} \wedge \mathtt{a}[x] = \mathtt{n}) \wedge \mathtt{pc}' = 4.$$

The error location is unreachable iff $\forall x.\ (x \geq 0 \wedge x < \mathtt{a.length}) \Rightarrow \mathtt{a}[x] \neq \mathtt{n}$ holds when exiting `find`. ⊣

3 Unwinding Array-Based Systems

We adapt some of the notions in [21] so that they can be easily integrated in the framework of MCMT. If only simple variables are considered, our approach closely resembles that in [21]. The main difference is that our technique uses backward instead of forward reachability to explore the set of reachable states.

If ψ is a quantifier-free formula in which at most the index variables $\underline{i}$ occur, we denote by $\psi^{\exists}$ its existential (index) closure, namely the formula $\exists \underline{i}\, \psi$. The *matrix* of a guarded assignment in functional form $\tau(\mathbf{v}, \mathbf{v}')$ of the form (1) is the formula (1) itself without the existential prefix $\exists \underline{k}$; the *proper variables* of τ are the $\underline{k}$. Below, we shall feel free to apply bounded variables renamings to formulæ of the form (1) without explicit mention.

Definition 1. *A* labeled unwinding *of* $\mathcal{S} = \langle \mathbf{v}; \{\tau_h(\mathbf{v}, \mathbf{v}')\}_h \rangle$ *is a quadruple* (V, E, M_E, M_V)*, where* (V, E) *is a finite rooted tree (let* ε *be the root) and* M_E, M_V *are labeling functions for edges and vertices, respectively, such that:*

(*i*) *for every* $v \in V$*, if* $v \neq \varepsilon$*, then* $M_V(v)$ *is a quantifier-free formula of the kind* $\psi(\underline{i}, \mathbf{a}[\underline{i}], \mathbf{c}, \mathbf{d})$ *such that* $M_V(v) \models_{A_I^E} pc = l$ *for some location* l*; otherwise* $M_V(\varepsilon)$ *is* $pc = l_E$*;*

(*ii*) *for every* $(v, w) \in E$*,* $M_E(v, w)$ *is the matrix of some* $\tau \in \{\tau_h(\mathbf{v}, \mathbf{v}')\}_h$*; the proper variables of* τ *do not occur in* $M_V(w)$*; moreover, we have that* $M_V(w) \models_{A_I^E} pc = trg(\tau)$*, that* $M_V(v) \models_{A_I^E} pc = src(\tau)$*, and that*

$$M_E(v, w)(\mathbf{v}, \mathbf{v}') \wedge M_V(w)(\mathbf{v}') \models_{A_I^E} M_V(v)(\mathbf{v}); \tag{3}$$

(*iii*) *for each* $\tau \in \{\tau_h(\mathbf{v}, \mathbf{v}')\}_h$ *and every non-leaf vertex* $w \in V$ *such that* $M_V(w) \models_{A_I^E} pc = trg(\tau)$*, there exist* $v \in V$ *and* $(v, w) \in E$ *such that* $M_E(v, w)$ *is the matrix of* τ*.*

The intuition underlying the definition is that a vertex v in a labeled unwinding corresponds to a program location (i) and an edge (v, w) to the execution of a transition, whose source and target locations match with those of v and w, respectively (*ii* and *iii*). It is interesting to more closely analyze condition (3). To this end, we recall the definition of *pre-image* of a formula $K(\mathbf{v})$ with respect to a transition $\tau(\mathbf{v}, \mathbf{v}')$, which is one of the key ingredients of backward reachability (see, e.g., [10]): $Pre(\tau, K) := \exists \mathbf{v}'.\ (\tau(\mathbf{v}, \mathbf{v}') \wedge K(\mathbf{v}'))$. It is not difficult to see that condition (3) is equivalent to $\exists \mathbf{v}'.\ (M_E(v, w)(\mathbf{v}, \mathbf{v}') \wedge M_V(w)(\mathbf{v}')) \models_{A_I^E} M_V(v)(\mathbf{v})$ which, in turn, implies $Pre(\tau, M_V(w)^{\exists}) \models_{A_I^E} M_V(v)^{\exists}$, if $M_E(v, w)$ is the matrix of τ. It is now clear that $M_V(v)^{\exists}$, i.e. the set of states associated to vertex v, *overapproximates* the set of states in the pre-image of $M_V(w)^{\exists}$ with respect to τ. Thus, the disjunction of the (existential index closure of the) formulæ labeling the nodes of an unwinding is an over-approximation of the set of backward reachable states and its negation (under suitable completeness conditions, see Definition 2 below) is an invariant of the system. A set C of vertexes in a labeled unwinding (V, E, M_E, M_V) *covers* a vertex $v \in V$ iff

$$M_V(v)^{\exists} \models_{A_I^E} \bigvee_{w \in C} M_V(w)^{\exists}. \tag{4}$$

Definition 2. *The labeled unwinding* (V, E, M_E, M_V) *is* safe *iff for all* $v \in V$ *we have that if* $M_V(v) \models pc = l_I$*, then* $M_V(v)$ *is* A_I^E*-unsatisfiable. It is* complete *iff there exists a* covering*, i.e., a set of non-leaf vertexes* C *containing* ε *and such that for every* $v \in C$ *and* $(v', v) \in E$*, it happens that* C *covers* v'*.*

The reader familiar with [21] may have noticed that our notion of covering involves a set of vertexes rather than a single one as in [21]. Indeed, an efficient implementation of our notion is delicate and is discussed in Section 4. Here, we focus on abstract definitions which allow us to prove that safe and complete labeled unwindings can be seen as safety certificates for array-based systems.

Theorem 1. *An array-based system is safe if there exists a safe and complete labeled unwinding for it.*

3.1 Lazy Abstraction with Interpolants in MCMT

We are left with the problem of computing labeled unwindings and checking for their safety and completeness. Similarly to [21], we design a possibly non-terminating procedure, called Unwind, that, given an array-based system $\mathcal{S}$, computes a sequence of (increasingly larger) labeled unwindings. The initial labeled unwinding of $\mathcal{S}$ is the tree containing just the root labeled by $pc = l_E$. Unwind uses two sub-procedures, called Expand and Refine, which can be non-deterministically applied to a labeled unwinding to obtain a new one, if possible. When Refine is applicable but fails, $\mathcal{S}$ is unsafe. If none of the two procedures applies, then the current labeled unwinding is safe and complete; thus $\mathcal{S}$ is safe by Theorem 1.

The core of our procedure is the sub-procedure Refine that performs refinement of labelings in presence of spurious unsafety traces. The distinguishing feature of our method is that, *despite the fact that we use quantified formulæ to represent sets of states and transitions, for refinement we need only quantifier-free interpolation* (even in a restricted form). Technically, this is made possible because the formulæ describing potentially unsafe traces are equisatisfiable with quantifier-free formulæ obtained by a restricted form of instantiation (see below for the technical details). We now describe the two sub-procedures.

Let (V, E, M_E, M_V) be the current labeled unwinding of $\mathcal{S}$. From now on, we assume that *the initial location is not a target location, the error location is not a source location*, and that initial and error locations are *the only locations that are not both a source and a target location.*

Expand. The applicability condition is that (V, E, M_E, M_V) is not complete and that there exists a leaf vertex v whose location is such that $M_V(v) \not\models_{A_I^E} pc = l_I$. By Definition 1$(i)$, we must have $M_V(v) \models_{A_I^E} pc = l$ for some $l \neq l_I$. For each transition $\tau \in \{\tau_h\}_h$ whose target is l, add a new leaf w_τ, label it by $pc = src(\tau)$, add the edge (w_τ, v) to the current tree, and label it by τ. ⊣

Refine. The applicability condition is that (V, E, M_E, M_V) is not complete and there exists a vertex $v \in V$ whose location is l_I and it is such that $M_V(v)$ is

A_I^E-satisfiable. Consider the path $v = v_0 \to v_1 \to \cdots \to v_m = \varepsilon$ from v to the root and let $\tau_1, \ldots, \tau_m$ be the transitions labeling the edges from left to right. If

$$\tau_1(\mathbf{v}^{(0)}, \mathbf{v}^{(1)}) \wedge \cdots \wedge \tau_m(\mathbf{v}^{(m-1)}, \mathbf{v}^{(m)}) \tag{5}$$

is A_I^E-satisfiable (notice that this is decidable, see [1] for details), then fail and report the unsafety of $\mathcal{S}$. Otherwise, update the formulæ labeling $v_0, \ldots, v_m$ by using interpolants as follows. By recalling (1), rewrite (5) as

$$\bigwedge_{k=1}^{m} \exists \underline{i}_k \left(\begin{array}{r} \phi_k(\underline{i}_k, \mathbf{a}^{(k-1)}[\underline{i}_k], \mathbf{c}^{(k-1)}, \mathbf{d}^{(k-1)}) \ \wedge \\ \mathbf{a}^{(k)} = \lambda j.\ G_k(\underline{i}_k, \mathbf{a}^{(k-1)}[\underline{i}_k], \mathbf{c}^{(k-1)}, \mathbf{d}^{(k-1)}, j, \mathbf{a}^{(k-1)}[j]) \ \wedge \\ \mathbf{c}^{(k)} = H_k(\underline{i}_k, \mathbf{a}^{(k-1)}[\underline{i}_k], \mathbf{c}^{(k-1)}, \mathbf{d}^{(k-1)}) \ \wedge \\ \mathbf{d}^{(k)} = K_k(\underline{i}_k, \mathbf{a}^{(k-1)}[\underline{i}_k], \mathbf{c}^{(k-1)}, \mathbf{d}^{(k-1)}) \end{array} \right) \tag{6}$$

which, by Skolemizing existentially quantified variables, is transformed to the equi-satisfiable formula (below, by abuse of notation, we consider the symbols in $\underline{i}_k$ as Skolem constants):

$$\bigwedge_{k=1}^{m} \left(\begin{array}{r} \phi_k(\underline{i}_k, \mathbf{a}^{(k-1)}[\underline{i}_k], \mathbf{c}^{(k-1)}, \mathbf{d}^{(k-1)}) \ \wedge \\ \mathbf{a}^{(k)} = \lambda j.\ G_k(\underline{i}_k, \mathbf{a}^{(k-1)}[\underline{i}_k], \mathbf{c}^{(k-1)}, \mathbf{d}^{(k-1)}, j, \mathbf{a}^{(k-1)}[j]) \ \wedge \\ \mathbf{c}^{(k)} = H_k(\underline{i}_k, \mathbf{a}^{(k-1)}[\underline{i}_k], \mathbf{c}^{(k-1)}, \mathbf{d}^{(k-1)}) \ \wedge \\ \mathbf{d}^{(k)} = K_k(\underline{i}_k, \mathbf{a}^{(k-1)}[\underline{i}_k], \mathbf{c}^{(k-1)}, \mathbf{d}^{(k-1)}) \end{array} \right) \tag{7}$$

Now, observe that $\mathbf{a}^{(k)} = \lambda j\, G_k(\ldots)$ is equivalent to $\forall j.\ \mathbf{a}^{(k)}[j] = G_k(\ldots j \ldots)$ and *instantiate the variable j with the Skolem constants in $\underline{i}_{k+1}, \ldots, \underline{i}_m$* to derive

$$\bigwedge_{k=1}^{m} \left(\begin{array}{r} \phi_k(\underline{i}_k, \mathbf{a}^{(k-1)}[\underline{i}_k], \mathbf{c}^{(k-1)}, \mathbf{d}^{(k-1)}) \ \wedge \\ \bigwedge_{j \in \underline{i}_{k+1}, \ldots, \underline{i}_m} \mathbf{a}^{(k)}[j] = G_k(\underline{i}_k, \mathbf{a}^{(k-1)}[\underline{i}_k], \mathbf{c}^{(k-1)}, \mathbf{d}^{(k-1)}, j, \mathbf{a}^{(k-1)}[j]) \ \wedge \\ \mathbf{c}^{(k)} = H_k(\underline{i}_k, \mathbf{a}^{(k-1)}[\underline{i}_k], \mathbf{c}^{(k-1)}, \mathbf{d}^{(k-1)}) \ \wedge \\ \mathbf{d}^{(k)} = K_k(\underline{i}_k, \mathbf{a}^{(k-1)}[\underline{i}_k], \mathbf{c}^{(k-1)}, \mathbf{d}^{(k-1)}) \end{array} \right). \tag{8}$$

Formula (8) is A_I^E-equisatisfiable to (7), see [1] for a proof. Now, (5) was supposed to be A_I^E-unsatisfiable, hence so are (6), (7) and finally (8). Let us abbreviate the k-th conjunct in the big conjunction (8) as

$$\tilde{\tau}_k(\underline{i}_k, \ldots, \underline{i}_m, \mathbf{a}^{(k-1)}[\underline{i}_k], \ldots, \mathbf{a}^{(k-1)}[\underline{i}_m], \mathbf{a}^{(k)}[\underline{i}_{k+1}], \ldots, \mathbf{a}^{(k)}[\underline{i}_m], \mathbf{c}^{(k-1)}, \mathbf{c}^{(k)}, \mathbf{d}^{(k-1)}, \mathbf{d}^{(k)}) \ , \tag{9}$$

so that (8) is written as $\tilde{\tau}_1 \wedge \cdots \wedge \tilde{\tau}_m$. Finally, let

$$\psi_k(\underline{i}_{k+1}, \ldots, \underline{i}_m, \mathbf{a}[\underline{i}_{k+1}], \ldots, \mathbf{a}[\underline{i}_m], \mathbf{c}, \mathbf{d}) \tag{10}$$

be the (quantifier-free interpolants) computed in (8) from right-to-left such that

$$\psi_0 \equiv \bot, \qquad \psi_m \equiv \top, \tag{11}$$

$$\begin{array}{r} \psi_k(\underline{i}_{k+1}, \ldots, \underline{i}_m, \mathbf{a}^{(k)}[\underline{i}_{k+1}], \ldots, \mathbf{a}^{(k)}[\underline{i}_m], \mathbf{c}^{(k)}, \mathbf{d}^{(k)}) \wedge \tilde{\tau}_k \models_{A_I^E} \\ \psi_{k-1}(\underline{i}_k, \ldots, \underline{i}_m, \mathbf{a}^{(k-1)}[\underline{i}_k], \ldots, \mathbf{a}^{(k-1)}[\underline{i}_m], \mathbf{c}^{(k-1)}, \mathbf{d}^{(k-1)}), \end{array} \tag{12}$$

and update the label of v_k as follows:

$$M_V(v_k) \;\equiv\; M_V(v_k) \wedge \psi_k(\underline{i}_{k+1}, \ldots, \underline{i}_m, \mathbf{a}[\underline{i}_k], \ldots, \mathbf{a}[\underline{i}_m], \mathbf{c}, \mathbf{d}). \tag{13}$$

Notice that, since the matrix of τ_k entails $\tilde{\tau}_k$, the condition (3) is preserved and the vertex $v = v_0$ is labeled by an A_I^E-unsatisfiable formula. ⊣

Both Expand and Refine prescribe to establish if the current unwinding is complete. According to Definition 2, this requires to guess a sub-set C of the set of vertexes in the unwinding and check if C covers v', for every $v \in C$ and $(v', v) \in E$. In turn, this may be reduced to repeatedly check the A_I^E-unsatisfiability of an $\exists^{A,I}\forall^I$-sentence (recall the definition in Section 2), reasoning by refutation from (4). These satisfiability checks are decidable under suitable conditions [10], which will be briefly recalled in Section 3.2 when discussing the completeness of our technique. However, even when the conditions for decidability are not satisfied, it is still possible to use sound but incomplete algorithms which preserves the soundness of Unwind. Concerning Refine, notice that it is possible to predict the numbers e_k of (implicitly existentially quantified) index variables occurring in the formulæ labeling the vertex v_k of a path of the form $v_0 \to \cdots \to v_m = \varepsilon$ by simply counting the existentially quantified index variables in $\tau_{k+1} \wedge \cdots \wedge \tau_m$ from (5). The number of index variables that will occur in the formula labeling v_k after the update (13) is bounded by e_k, because it is derived from the interpolants computed along the path considered above. On the other hand, the number of index variables labeling the leaves may grow very quickly, thereby posing a crucial problem for implementation (e.g., when instantiating universally quantified variables in covering tests). Fortunately, heuristics [9,11] designed to reduce the number of index variables in pre-images developed for the backward reachability procedure of mcmt can also be put to productive use in the main loop of Unwind.

The third observation on Refine concerns the computation of the interpolants. An easy way to derive ψ_{k-1} from ψ_k would be to use the pre-image of ψ_{k-1} with respect to the transition labeling the edge connecting the vertexes whose label is to be updated by ψ_{k-1} and ψ_k. However, for Unwind to be truly an *abstraction*-based procedure, we need to compute interpolants which do not necessarily reduce to the precise preimage. This can be done by combining the available interpolation algorithms for T_I and T_E. Unrestricted combination is not always possible in general (there are negative results in the literature showing e.g. that the addition of free function symbols can destroy quantifier-free interpolation [2]); however, because of the form (9) of the above formulæ $\tilde{\tau}_k$, it follows that whenever Unwind needs to compute an interpolant for an unsatisfiable quantifier-free formula $\psi_1 \wedge \psi_2$, the formulæ ψ_1, ψ_2 satisfy the hypotheses of the following positive result:

Theorem 2. *Suppose that $\psi_1 \wedge \psi_2$ is an A_I^E-unsatisfiable quantifier-free formula such that all variables of sort* `INDEX` *occurring in ψ_2 under the scope of the read operator $\_[\_]$ occur also in ψ_1. Then, there exists a quantifier-free formula ψ_0 such that: (i) $\psi_2 \models_{A_I^E} \psi_0$; (ii) $\psi_0 \wedge \psi_1$ is A_I^E-unsatisfiable; (iii) all free variables occurring in ψ_0 occur both in ψ_1 and ψ_2.*

The soundness of UNWIND is guaranteed by the following result.

Theorem 3. *If neither* EXPAND *nor* REFINE *can be applied to a labeled unwinding* $P = (V, E, M_E, M_V)$*, then* P *is safe and complete.*

Example 2. We briefly discuss how UNWIND is applied to the array-based system of Example 1. Fig. 2 (without boxed literals) reports an unwinding that, starting from the error location ($M_V(\epsilon) \models \mathtt{pc} = 4$) reaches the initial location ($M_V(v_{25}) \models \mathtt{pc} = 1$). An infeasible trace depicted in Fig. 2. The counterexample associated to the trace is the following (for the sake of conciseness, we list only the variables changing their values):

$$
\begin{aligned}
&\mathtt{pc}^{(4)} = 1 \wedge \\
&\mathtt{pc}^{(4)} = 1 \wedge \mathtt{pc}^{(3)} = 2 \wedge \mathtt{c}^{(3)} = 0 \wedge \\
&\mathtt{pc}^{(3)} = 2 \wedge \mathtt{pc}^{(2)} = 2 \wedge \mathtt{a.length} > \mathtt{c}^{(3)} \wedge \mathtt{c}^{(2)} = \mathtt{c}^{(3)} + 1 \wedge i_1 = \mathtt{c}^{(3)} \wedge \mathtt{a}^{(3)}[i_1] \neq \mathtt{n} \wedge \\
&\mathtt{pc}^{(2)} = 2 \wedge \mathtt{pc}^{(1)} = 3 \wedge \mathtt{a.length} \leq \mathtt{c}^{(2)} \wedge \\
&\mathtt{pc}^{(1)} = 3 \wedge \mathtt{pc}^{(0)} = 4 \wedge \mathtt{a.length} \leq \mathtt{c}^{(1)} \wedge \mathtt{a}^{(1)}[i_0] = \mathtt{n} \wedge \mathtt{a.length} > i_0 \wedge \\
&\mathtt{pc}^{(0)} = 4
\end{aligned}
$$

The counterexample is unsatisfiable and it is thus infeasible in the concrete system. A set of interpolants computed from the trace above contains

$$
\begin{aligned}
&\psi_0 \equiv \bot, \quad \psi_1 \equiv \bot, \quad \psi_2 \equiv i_0 \leq \mathtt{c} \wedge i_0 \geq 0, \quad \psi_3 \equiv i_0 \leq \mathtt{c} \wedge i_0 \geq 0, \\
&\psi_4 \equiv i_0 \leq \mathtt{a.length} \wedge i_0 \geq 0, \quad \psi_5 \equiv i_0 \geq 0, \text{ and } \psi_6 \equiv \top.
\end{aligned}
$$

According to (13), the refinement of the infeasible trace is done by adding each interpolant to the corresponding vertex in the unwinding (see the boxed literals in Fig. 2). UNWIND is then able to generate the invariant

$$(\mathtt{pc} = 3 \wedge \mathtt{c} > 0 \wedge \mathtt{a.length} \geq 1) \Rightarrow \forall i.\, ((i < \mathtt{c} \wedge i \leq \mathtt{a.length}) \Rightarrow \mathtt{a}[i] \neq \mathtt{n})$$

as the negation of the label of v_1, which states that if the the loop is executed at least once (antecedent of the main implication), then at every position i (up to $\mathtt{c}$) of the array is stored a value distinct from $\mathtt{n}$. Notice that the predicates $\mathtt{c} > 0$, $\mathtt{a.length} \geq 1$ and $i < \mathtt{c}$ (where i is an universally quantified variable) are new and have been generated by the interpolation algorithm. ⊣

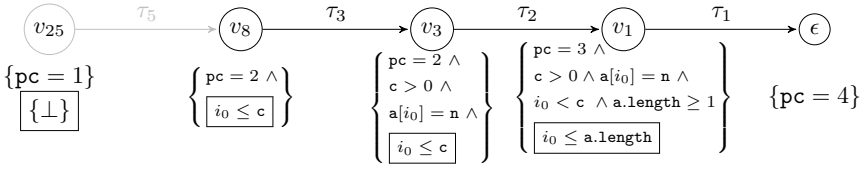


Fig. 2. Counterexample for Example 2. Boxed labels were added by refinement.

3.2 Completeness and Termination

The completeness of UNWIND depends on the decidability of checking whether a labeled unwinding is complete according to Definition 2. We have already argued (see first observation after the description of REFINE in Section 3.1) that this can be reduced to the A_I^E-satisfiability of $\exists^{A,I}\forall^I$-sentences.

Theorem 4 ([10]). *If there are no function symbols in the signature Σ_I of T_I and the class $\mathcal{C}_I$ of models of T_I is closed under substructures, then the A_I^E-satisfiability of $\exists^{A,I}\forall^I$-sentences is decidable.*

In [10], the proof of this result[1] is constructive by showing a procedure which first instantiates the universally quantified index variables with the existentially quantified index variables (considered as Skolem constants) of the sentence in all possible ways and then invokes a combination (*à la* Nelson-Oppen) of the available decision procedures for the $SMT(T_I)$- and $SMT(T_E)$-problems (recall the assumptions in Section 2). The procedure is still sound but incomplete when the assumptions on T_I in Theorem 4 do not hold. For efficiency, heuristics [9] have been designed to reduce the number of possible instantiations.

Conditions for the termination of UNWIND are much more restrictive. First, a fair strategy must be used to apply EXPAND and REFINE. Formally, a strategy is *fair* if it does not indefinitely delay the application of one of the two procedures and does not apply REFINE infinitely many times to the label of the same vertex. Notice that the latter holds if there are no infinitely many non-equivalent formulæ of the form $\psi(\underline{i}, \mathbf{a}[\underline{i}], \mathbf{c}, \mathbf{d})$ for a given $\underline{i}$ or, alternatively, if a refinement based on the computation of interpolants through the precise preimage is eventually applied when repeatedly refining a node. The second condition (since adopting a fair strategy alone is not sufficient) for termination concerns also the theory T_E. To formally state such conditions, we need to adapt some notions from [3,10]. A *wqo-theory* is a theory $T = (\Sigma, \mathcal{C})$ such that $\mathcal{C}$ is closed under substructures and finitely generated models of T are a well-quasi-order with respect to the relation $\preceq$ that holds between $\mathcal{M}_1$ and $\mathcal{M}_2$ whenever $\mathcal{M}_1$ embeds into $\mathcal{M}_2$.

Theorem 5. *Let $\mathcal{S} = \langle \mathbf{v}; \{\tau_h(\mathbf{v}, \mathbf{v}')\}_h \rangle$ be an array-based system for T_I, T_E. Suppose that T_I satisfies the hypotheses of Theorem 4 and that the theory obtained from $T_I \cup T_E$ by adding it the symbols $\mathbf{v}$ (seen as free constants of appropriate sorts) is a wqo theory. Then,* UNWIND *terminates when applied to $\mathcal{S}$ with a fair strategy.*

As a consequence, UNWIND behaves as a decision procedure for those classes of array-based systems satisfying the conditions of Theorem 5. This is the case, for example, of broadcast protocols and lossy channels systems (see [3,10] for details). A similar result for broadcast protocols is given in [7] within forward reachability.

[1] Although in this paper, we have also counters and simple variables in the definition of array-based systems, that were not considered in [10], this does not interfere with the correctness of the algorithm in [10] which can be easily extended to cope with them.

4 Implementation and Experiments

We have implemented UNWIND on top of a re-engineered version of MCMT[2] ; the following two heuristics are the key ingredients for its practical applicability.

Term Abstraction. While experimenting with our prototype, we realized that available interpolating procedures seldom permit to refine the abstraction in the "right" way because some terms are not eliminated. This dramatically decreases performances or, even worse, prevents to find the inductive invariant, when it exists. To alleviate this problem, the goal of the *term abstraction* heuristic is to compute (if possible) an interpolant where a certain term t does not occur. As briefly explained in Section 2, the term t is usually some counter which should be eliminated to synthesize (candidate) invariants involving universally quantified index variables, even when the problem specification mentions no quantifiers. Term abstraction proceeds as follows. Given an A_I^E-unsatisfiable formula of the form $\psi_1 \wedge \psi_2$, if $\psi_1(c_1/t) \wedge \psi_2(c_2/t)$ is A_I^E-unsatisfiable, for c_1 and c_2 fresh constants, then term abstraction returns the interpolant of $\psi_1(c_1/t) \wedge \psi_2(c_2/t)$, computed by running the available interpolation procedure. Otherwise, term abstraction returns the interpolant of the original formula $\psi_1 \wedge \psi_2$. Our prototype tool automatically extracts from the problem specification a list of "relevant" terms, called *term abstraction list*, which contains candidates for the term abstraction heuristic (alternatively, the user can provide such a list).

Covering Strategy. We implemented an additional procedure REDUCE which is to be interleaved with EXPAND so as to reduce as much as possible the invocations to the latter. REDUCE checks, given a vertex v of the unwinding and a set $\tilde{V}$ of nodes such that every $v_i \in \tilde{V}$ is not covered, whether $M_V(v) \models_{A_I^E} \bigvee_{v_i \in \tilde{V}} M_V(v_i)$, i.e., it checks if v is covered by a disjunction of vertexes that share the same value of the program counter. In addition we allow leaves to be covered by younger vertices (while in [21] a vertex can be covered only by one older vertex). REDUCE agrees with the notion of covering introduced in Definition 2. Moreover, before expanding a leaf v, we check if there is at least one v's ancestor u such that u is covered by its ancestors. If so, a descendant of u can neither be expanded nor cover other vertices as long as u is covered.

Table 1 reports the results of our experiments (run on an Intel i7 @2.66 GHz, equipped with 4GB of RAM and running OSX 10.7). Our benchmark set includes simple programs over arrays, e.g., initialization of all elements to 0, copy of one array into another, etc. They have been taken from other papers (e.g., [17,19,23]), or from standard textbooks on algorithms. All benchmarks diverge if no abstraction is provided.

The last two benchmarks in Table 1 are trickier to verify, as they feature two loops. In "init and test", there are two loops in sequence and the safety condition consists in reaching an error location. Although the property does not contain quantifiers, the inductive invariant of the program does need quantifiers. The

[2] The executable of the prototype tool and all the input files can be downloaded at http://www.oprover.org/mcmt_abstraction.html.

Table 1. Table reports: total verification time, number of nodes of the final unwinding, number of calls to the SMT-solver, number of CEGAR iterations, final safety result.

Benchmark	Description	Time (s)	Nodes	SMT-calls	Iter.	Result
find (v1)	Find an element	0.3	5	192	3	SAFE
find (v2)	(as above, alternative encoding)	0.07	5	48	1	SAFE
initialization	Initialize all elements to 0	0.1	5	96	1	SAFE
max in array	Find max element	0.9	72	1192	8	SAFE
partition	Partition an array	0.08	20	62	0	SAFE
strcmp	Compare arrays	0.4	14	329	4	SAFE
strcpy	Copy arrays	0.03	3	15	0	SAFE
vararg	Search for end of arguments	0.03	5	17	0	SAFE
integers	Numerical property	0.02	5	19	0	SAFE
init and test	Init. to 0 and tests	0.3	27	375	3	SAFE
binary sort	Sorting with binary search	0.3	48	457	2	SAFE

methodology applied by our tool to introduce extra quantifers is the following: first, recall from Section 2 that sentences like $\psi(..\mathbf{a}[\mathbf{c}]..)$ are written as $\exists \underline{i}(\underline{i} = \mathbf{c} \wedge \psi(..\mathbf{a}[\underline{i}]..))$, then term abstraction can get rid of (some of) the $\mathbf{c}$ thus letting (some of) the $\underline{i}$ be genuine new quantifiers. As for nested loops, "binary sort" is an encoding of the sorting algorithm based on binary search.

To test the flexibility of our approach, we run the prototype on some randomly generated problems taken from those shipped with the distribution of the ARMC model-checker (`http://www.mpi-sws.org/~rybal/armc/`). They consists of safety properties of numerical programs without arrays. Our tool can solve 22 out of 28 benchmarks with abstraction, but only 9 without using it. For those benchmarks that could be solved even without abstraction, the overhead of abstraction is generally negligible.

5 Conclusion

We have described UNWIND, a verification procedure for safety properties based on the combination of the backward reachability of MCMT and lazy-abstraction with interpolants. Lazy-abstraction is enabled to handle (unbounded) arrays while MCMT is now capable to cope with sequential programs in a uniform way by using abstraction and refinement. Our experiments show that the improved version of MCMT is able to prove safety properties in no time for common-use programs over arrays. As future work, we plan to tune the abstraction and refinement mechanisms to other classes of systems, such as distributed algorithms.

Acknowledgements. The work of the first author was supported by the Hasler Foundation under project 09047 and that of the fourth author was partially supported by the "SIAM" project founded by Provincia Autonoma di Trento in the context of the "team 2009 - Incoming" COFUND action of the European Commission (FP7).

References

1. Alberti, F., Bruttomesso, R., Ghilardi, S., Ranise, S., Sharygina, N.: Lazy Abstraction with Interpolants for Arrays. extended version, http://homes.dsi.unimi.it/~ghilardi/allegati/ABGRS_LPAR.pdf
2. Brillout, A., Kroening, D., Rümmer, P., Wahl, T.: An Interpolating Sequent Calculus for Quantifier-Free Presburger Arithmetic. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS, vol. 6173, pp. 384–399. Springer, Heidelberg (2010)
3. Carioni, A., Ghilardi, S., Ranise, S.: Automated Termination in Model Checking Modulo Theories. In: Delzanno, G., Potapov, I. (eds.) RP 2011. LNCS, vol. 6945, pp. 110–124. Springer, Heidelberg (2011)
4. 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)
5. Cousot, P., Cousot, R., Logozzo, F.: A Parametric Segmentation Functor for Fully Automatic and Scalable Array Content Analysis. In: POPL (2011)
6. Dillig, I., Dillig, T., Aiken, A.: Fluid Updates: Beyond Strong vs. Weak Updates. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 246–266. Springer, Heidelberg (2010)
7. Dimitrova, R., Podelski, A.: Is Lazy Abstraction a Decision Procedure for Broadcast Protocols? In: Logozzo, F., Peled, D.A., Zuck, L.D. (eds.) VMCAI 2008. LNCS, vol. 4905, pp. 98–111. Springer, Heidelberg (2008)
8. Flanagan, C., Qadeer, S.: Predicate abstraction for software verification. In: POPL, pp. 191–202 (2002)
9. Ghilardi, S., Ranise, S.: Model Checking Modulo Theory at work: the integration of Yices in MCMT. In: AFM (2009)
10. Ghilardi, S., Ranise, S.: Backward Reachability of Array-based Systems by SMT solving: Termination and Invariant Synthesis. LMCS 6(4) (2010)
11. Ghilardi, S., Ranise, S.: MCMT: A Model Checker Modulo Theories. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS, vol. 6173, pp. 22–29. Springer, Heidelberg (2010)
12. Gopan, D., Reps, T., Sagiv, M.: A Framework for Numeric Analysis of Array Operations. In: POPL 2005, pp. 338–350 (2005)
13. Graf, S., Saïdi, H.: Construction of Abstract State Graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
14. Halbwachs, N., Mathias, P.: Discovering Properties about Arrays in Simple Programs. In: PLDI 2008, pp. 339–348 (2008)
15. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from Proofs. In: POPL, pp. 232–244 (2004)
16. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy Abstraction. In: POPL, pp. 58–70 (2002)
17. Jhala, R., McMillan, K.L.: Array Abstractions from Proofs. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 193–206. Springer, Heidelberg (2007)
18. Kapur, D., Majumdar, R., Zarba, C.: Interpolation for Data Structures. In: SIGSOFT 2006/FSE-14, pp. 105–116 (2006)
19. Kovács, L., Voronkov, A.: Interpolation and Symbol Elimination. In: Schmidt, R.A. (ed.) CADE-22. LNCS, vol. 5663, pp. 199–213. Springer, Heidelberg (2009)
20. McMillan, K.L.: Quantified Invariant Generation Using an Interpolating Saturation Prover. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 413–427. Springer, Heidelberg (2008)

21. McMillan, K.L.: Lazy Abstraction with Interpolants. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 123–136. Springer, Heidelberg (2006)
22. Ranise, S., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB) (2006), http://www.SMT-LIB.org
23. Seghir, M.N., Podelski, A., Wies, T.: Abstraction Refinement for Quantified Array Assertions. In: Palsberg, J., Su, Z. (eds.) SAS 2009. LNCS, vol. 5673, pp. 3–18. Springer, Heidelberg (2009)
24. Lahiri, S., Bryant, R.: Predicate Abstraction with Indexed Predicates. TOCL 9(1) (2007)
25. Srivastava, S., Gulwani, S.: Program Verification using Templates over Predicate Abstraction. In: PLDI (2009)

Backward Trace Slicing for Conditional Rewrite Theories*

María Alpuente[1], Demis Ballis[2], Francisco Frechina[1], and Daniel Romero[1]

[1] DSIC-ELP, Universitat Politècnica de València,
Camino de Vera s/n, Apdo 22012, 46071 Valencia, Spain
{alpuente,ffrechina,dromero}@dsic.upv.es
[2] DIMI, Università degli Studi di Udine,
Via delle Scienze 206, 33100 Udine, Italy
demis.ballis@uniud.it

Abstract. In this paper, we present a trace slicing technique for rewriting logic that is suitable for analyzing complex, textually-large system computations in rewrite theories that may contain conditional equations and/or rules. Given a conditional execution trace $\mathcal{T}$ and a slicing criterion for the trace (i.e., a set of positions that we want to observe in the final state of the trace), we traverse $\mathcal{T}$ from back to front, and at each rewrite step, we incrementally compute the origins of the observed positions, which is done by inductively processing the conditions of the applied equations and rules. During the traversal, we also carry a boolean compatibility condition that is needed for the executability of the processed rewrite steps. At the end of the traversal, the trace slice is obtained by filtering out the irrelevant data that do not contribute to the criterion of interest.

1 Introduction

The analysis of computation traces plays an important role in many program analysis approaches. Software systems commonly generate large and complex execution traces, whose analysis (or even simple inspection) is extremely time-consuming, and in some cases unfeasible to perform by hand. Trace slicing is a technique for reducing the size of execution traces by focusing on selected execution aspects, which makes it suitable for trace analysis and monitoring [10].

Rewriting Logic (RWL) is a very general *logical* and *semantic framework*, which is particularly suitable for formalizing highly concurrent, complex systems (e.g., biological systems [7] and Web systems [2,5]). RWL is efficiently implemented in the high-performance system Maude [12]. Roughly speaking, a *(conditional) rewriting logic theory* [16] seamlessly combines a *(conditional) term*

* This work has been partially supported by the EU (FEDER) and the Spanish MEC TIN2010-21062-C02-02 project, by Generalitat Valenciana, ref. PROMETEO2011/052, and by the Italian MUR under grant RBIN04M8S8, FIRB project, Internationalization 2004. Also, D. Romero is supported by FPI-MEC grant BES-2008-004860 and F. Frechina is supported by FPU-ME grant AP2010-5681.

N. Bjørner and A. Voronkov (Eds.): LPAR-18, LNCS 7180, pp. 62–76, 2012.

rewriting system (CTRS), together with an *equational theory* (also possibly conditional) that may include equations and axioms (i.e., algebraic laws such as commutativity, associativity, and unity) so that rewrite steps are applied *modulo* the equations and axioms.

In recent years, the debugging and optimization techniques based on RWL have received growing attention. However, to the best of our knowledge, the only trace slicing technique that gives support to the analysis of RWL computations is [3]. Given an execution trace $\mathcal{T}$, [3] generates a trace slice of $\mathcal{T}$ w.r.t. a set of symbols of interest (target symbols) that appear in a given state of $\mathcal{T}$. The technique relies on a suitable mechanism of backward tracing that computes the reverse dependence among the symbols involved in an execution step by using a procedure that dynamically labels the calls (terms) involved in the steps.

Unfortunately, the technique in [3] is only applicable to *unconditional* RWL theories, and hence it cannot be employed when the source program includes conditional equations and/or rules since it would deliver incorrect and/or incomplete trace slices. The following example illustrates why conditions cannot be disregarded by the slicing process, which is what has motivated our work.

Example 1. Consider the Maude specification of the function `_mod_` in Figure 1, which computes the reminder of the division of two natural numbers, and the associated execution trace `4 mod 5` $\rightarrow$ `4`. Assume that we are interested in observing the origins of the target symbol `4` that appears in the final state. If we disregard the condition `Y > X` of the first conditional equation, the slicing technique of [3] computes the trace slice `4 mod` $\bullet$ $\rightarrow$ `4`, whereas the correct trace slice is `4 mod 5` $\rightarrow$ `4` since both arguments of `mod` are required to prove the rewrite step that introduces the symbol `4` in the final state.

```
mod M is inc NAT .
 var X  : Nat .
 var Y : NzNat .
 op _mod_ : Nat NzNat -> Nat .
 ceq X mod Y = X if Y > X .
 ceq X mod Y = (X - Y) mod Y
        if Y <= X .
endm
```

Fig. 1. The `_mod_` operator

Contributions. We present the first conditional trace slicing technique for RWL computations. Our technique is fully general and can be applied for debugging as well as for optimizing any RWL-based tool that manipulates conditional RWL computations such as those delivered as counterexample traces by the Maude model-checker [6]. The backward conditional slicing algorithm in this paper cannot be considered to be a natural extension of the unconditional slicing method of [3], but greatly simplifies [3] by replacing the involved and costly dynamic labeling procedure, based on [8], with a simple mechanism for substitution refinement that allows control and data dependencies to be propagated between consecutive rewrite steps. Moreover, the conditional slicing algorithm copes with three different types of conditions that occur in Maude theories: equational conditions, matching conditions, and rewrite expressions. Our formulation takes into account the precise way in which Maude mechanizes the conditional rewriting process and revisits all those rewrite steps backwards in an instrumented, fine-grained way where each small step corresponds to the application of an equation (conditional equation or equational axiom) or rule. This allows

us to slice the input execution trace with regard to the set of symbols of interest (target symbols) by tracing back the target symbols along the execution trace so that all data that are not antecedents of the observed symbols are simply discarded.

Related Work. Tracing techniques have been extensively used in functional debugging [11]. For instance, Hat [11] is an interactive debugging system that enables a computation to be explored backwards, starting from the program output or an error message (with which the computation aborted). Backward tracing in Hat is carried out by navigating a redex trail (i.e., a graph-like data structure that records dependencies among function calls), whereas our tracing technique does not require handling any supplementary data structure.

There exist very few approaches that address the problem of tracing rewrite sequences in term rewrite systems [3,8,14,18], and all of them apply to unconditional systems. The techniques in [3,8,18] rely on a labeling relation on symbols that allows data content to be traced back within the computation; this is achieved in [14] by formalizing a notion of dynamic dependence among symbols by means of contexts. In [8,18], non-left linear and collapsing rules are not considered or are dealt using ad-hoc strategies, while our approach requires no special treatment of such rules. Furthermore, only [3] describes a tracing methodology for rewrite theories with rules, equations, sorts, and algebraic axioms.

In this paper, we propose a more general slicing technique for conditional rewrite theories that generalizes and simplifies the formal development in [3] by getting rid of the complex dynamic labeling algorithm that was needed to trace back the origins of the symbols of interest. Our technique also avoids manipulating the origins by recording their addressing positions; we simply and explicitly record the origins of the meaningful positions within the computed term slices themselves, without resorting to any other artifact.

To debug Maude programs, Maude has a tracing facility that allows the execution sequence to be traced, and is very customizable: it provides some control over conditions and allows the user to select the statements being applied at each step. A main difference with the trace slicing technique of ours is that the tracer of Maude allows the trace size to be reduced by manually focusing on statements, while slicing is automatic and focuses on terms. Moreover, since each small rewrite step that is obtained by applying a single conditional equation, equational axiom or rule is shown in the trace, the user can easily miss the general view, and when the user detects an incorrect intermediate result, it is difficult to know where the incorrect inference started. In this regard, the trace slices computed by our technique can be very helpful in debugging, since they only consist of the information that is strictly needed to deliver a critical part of the result (see discussion in [1]).

Plan of the Paper. Section 2 recalls some fundamental notions of RWL and Section 3 summarizes the conditional rewriting modulo equational theories defined in Maude. In Section 4, the backward conditional slicing technique is

formalized by means of a transition system that traverses the execution traces from back to front. Finally, Section 5 reports on a prototypical implementation of the proposed slicing technique and its experimental evaluation.

2 Preliminaries

Let us recall some important notions that are relevant to this work. We assume some basic knowledge of term rewriting [18] and Rewriting Logic [16]. Some familiarity with the Maude language [12] is also required.

We consider an *order-sorted signature* Σ, with a finite poset of sorts $(S, <)$ that models the usual subsort relation [12]. We assume an S-sorted family $\mathcal{V} = \{\mathcal{V}_s\}_{s \in S}$ of disjoint variable sets. $\tau(\Sigma, \mathcal{V})_s$ and $\tau(\Sigma)_s$ are the sets of terms and ground terms of sort s, respectively. We write $\tau(\Sigma, \mathcal{V})$ and $\tau(\Sigma)$ for the corresponding term algebras. The set of variables that occur in a term t is denoted by $\mathcal{V}ar(t)$. In order to simplify the presentation, we often disregard sorts when no confusion can arise.

A *position* w in a term t is represented by a sequence of natural numbers that addresses a subterm of t (Λ denotes the empty sequence, i.e., the root position). By notation $w_1.w_2$, we denote the concatenation of positions (sequences) w_1 and w_2. Positions are ordered by the prefix ordering, that is, given the positions w_1 and w_2, $w_1 \leq w_2$ if there exists a position u such that $w_1.u = w_2$. Given a set of positions P, the *prefix closure* of P is the set $\bar{P} = \{u \mid u \leq p \wedge p \in P\}$. Given a term t, we let $\mathcal{P}os(t)$ denote the set of positions of t. By $t|_w$, we denote the *subterm* of t at position w, and by $t[s]_w$, we denote the result of *replacing the subterm* $t|_w$ by the term s.

A substitution σ is a mapping from variables to terms $\{X_1/t_1, \ldots, X_n/t_n\}$ such that $X_i\sigma = t_i$ for $i = 1, \ldots, n$ (with $X_i \neq x_j$ if $i \neq j$), and $X\sigma = X$ for all other variables X. Given a substitution $\sigma = \{X_1/t_1, \ldots, X_n/t_n\}$, the *domain* of σ is the set $Dom(\sigma) = \{X_1, \ldots, X_n\}$. For any substitution σ and set of variables V, $\sigma_{\restriction V}$ denotes the substitution obtained from σ by restricting its domain to V, (i.e., $\sigma_{\restriction V}(X) = X\sigma$ if $X \in V$, otherwise $\sigma_{\restriction V}(X) = X$). Given two terms s and t, a substitution σ is a *matcher* of t in s, if $s\sigma = t$. By $match_s(t)$, we denote the function that returns a matcher of t in s if such a matcher exists, otherwise $match_s(t)$ returns *fail*.

We consider three different kinds of conditions that may appear in a conditional Maude theory: an *equational condition*[1] e is any (ordinary) equation $t = t'$, with $t, t' \in \tau(\Sigma, \mathcal{V})$; a *matching condition* is a pair $p := t$ with $p, t \in \tau(\Sigma, \mathcal{V})$; a *rewrite expression* is a pair $t \Rightarrow p$, with $p, t \in \tau(\Sigma, \mathcal{V})$.

A *conditional* equation is an expression of the form $\lambda = \rho$ *if* C, where $\lambda, \sigma \in \tau(\Sigma, \mathcal{V})$, and C is a (possibly empty) sequence $c_1 \wedge \ldots \wedge c_n$, where each c_i is either an equational condition, or a matching condition. When the condition C is empty, we simply write $\lambda = \rho$. A conditional equation $\lambda = \rho$ *if* $c_1 \wedge \ldots \wedge c_n$ is

[1] A boolean equational condition $b = true$, with $b \in \tau(\Sigma, \mathcal{V})$ of sort Bool, is simply abbreviated as b. A *boolean condition* is a sequence of abbreviated boolean equational conditions.

admissible, iff (i) $\mathcal{V}ar(\rho) \subseteq \mathcal{V}ar(\lambda) \cup \bigcup_{i=1}^{n} \mathcal{V}ar(c_i)$, and (ii) for each c_i, $\mathcal{V}ar(c_i) \subseteq \mathcal{V}ar(\lambda) \cup \bigcup_{j=1}^{i-1} \mathcal{V}ar(c_j)$ if c_i is an equational condition, and $\mathcal{V}ar(e) \subseteq \mathcal{V}ar(\lambda) \cup \bigcup_{j=1}^{i-1} \mathcal{V}ar(c_j)$ if c_i is a matching condition $p := e$.

A *conditional* rule is an expression of the form $\lambda \rightarrow \rho$ *if* C, where $\lambda, \sigma \in \tau(\Sigma, \mathcal{V})$, and C is a (possibly empty) sequence $c_1 \wedge \ldots \wedge c_n$, where each c_i is an equational condition, a matching condition, or a rewrite expression. When the condition C is empty, we simply write $\lambda \rightarrow \rho$. A conditional rule $\lambda \rightarrow \rho$ *if* $c_1 \wedge \ldots \wedge c_n$ is *admissible* iff it fulfils the exact analogous of the admissibility constraints (i) and (ii) for the equational conditions and the matching conditions, plus the following additional constraint: for each rewrite expression c_i in C of the form $e \Rightarrow p$, $\mathcal{V}ar(e) \subseteq \mathcal{V}ar(\lambda) \cup \bigcup_{j=1}^{i-1} \mathcal{V}ar(c_j)$.

The set of variables that occur in a (conditional) rule/equation r is denoted by $\mathcal{V}ar(r)$. Note that admissible equations and rules can contain extra-variables (i.e., variables that appear in the right-hand side or in the condition of a rule/equation but do not occur in the corresponding left-hand side). The admissibility requirements ensure that all the extra-variables will become instantiated whenever an admissible rule/equation is applied.

3 Conditional Rewriting Modulo Equational Theories

An *order-sorted equational theory* is a pair $E = (\Sigma, \Delta \cup B)$, where Σ is an order-sorted signature, Δ is a collection of (oriented) admissible, conditional equations, and B is a collection of unconditional equational axioms (e.g., associativity, commutativity, and unity) that can be associated with any binary operator of Σ. The equational theory E induces a congruence relation on the term algebra $T(\Sigma, \mathcal{V})$, which is denoted by $=_E$. A *conditional rewrite theory* (or simply, rewrite theory) is a triple $\mathcal{R} = (\Sigma, \Delta \cup B, R)$, where $(\Sigma, \Delta \cup B)$ is an order-sorted equational theory, and R is a set of admissible conditional rules[2].

Example 2. The following Maude rewrite theory defines a simple banking system. It includes three conditional rules: `credit`, `debit`, and `transfer`.

```
mod BANK is inc INT .
  sorts Account Msg State Id .
  subsorts  Account Msg < State .
  var Id Id1 Id2 : Id .
  var bal bal1 bal2 newBal newBal1 newBal2 M : Nat .
  op empty-state : -> State .
  op _;_ : State State -> State [assoc comm id: empty-state] .
  op <_|_> : Id Nat -> Account [ctor] .
  ops credit debit : Id Nat -> Msg [ctor] .
  op  transfer : Id Id Nat -> Msg  [ctor] .
  crl [credit] : <Id|bal>;credit(Id,M) => <Id|newBal> if newBal := bal + M .
  crl [debit] : <Id|bal>;debit(Id,M) => <Id|newBal> if bal >= M /\ newBal := bal - M .
  crl [transfer] : <Id1|bal1>;<Id2|bal2>;transfer(Id1,Id2,M) => <Id1|newBal1>;<Id2|newBal2>
   if <Id1|bal1>;debit(Id1,M) => <Id1|newBal1> /\ <Id2|bal2>;credit(Id2,M) => <Id2|newBal2> .
endm
```

[2] Equational specifications in Maude can be theories in membership equational logic, which may include conditional membership axioms not addressed in this paper.

The rule `credit` contains a matching condition `newBal := bal + M`. The rule `debit` contains an equational condition `bal >= M` and a matching condition `newBal := bal - M`. Finally, the rule `transfer` has a rule condition that contains two rewrite expressions `<Id1|bal1>;debit(Id1,M) => <Id1|newBal1>` and `<Id2|bal2>;credit(Id2,M) => <Id2|newBal2>`.

Given a conditional rewrite theory (Σ, E, R), with $E = \Delta \cup B$, the conditional rewriting modulo E relation (in symbols, $\rightarrow_{R/E}$) can be defined by lifting the usual conditional rewrite relation on terms [15] to the E-congruence classes $[t]_E$ on the term algebra $\tau(\Sigma, \mathcal{V})$ that are induced by $=_E$ [9], that is, $[t]_E$ is the class of all terms that are equal to t *modulo* E. Unfortunately, $\rightarrow_{R/E}$ is in general undecidable, since a rewrite step $t \rightarrow_{R/E} t'$ involves searching through the possibly infinite equivalence classes $[t]_E$ and $[t']_E$.

The conditional slicing technique formalized in this work is formulated by considering the precise way in which Maude proves the conditional rewriting steps (see Section 5.2 in [12]). Actually, the Maude interpreter implements conditional rewriting modulo E by means of two much simpler relations, namely $\rightarrow_{\Delta,B}$ and $\rightarrow_{R,B}$, that allow rules and equations to be intermixed in the rewriting process by simply using an algorithm of matching modulo B. We define $\rightarrow_{R\cup\Delta,B}$ as $\rightarrow_{R,B} \cup \rightarrow_{\Delta,B}$. Roughly speaking, the relation $\rightarrow_{\Delta,B}$ uses the equations of Δ (oriented from left to right) as simplification rules: thus, for any term t, by repeatedly applying the equations as simplification rules, we eventually reach a term $t{\downarrow_\Delta}$ to which no further equations can be applied. The term $t{\downarrow_\Delta}$ is called a *canonical form* of t w.r.t. Δ. On the other hand, the relation $\rightarrow_{R,B}$ implements rewriting with the rules of R, which might be non-terminating and non-confluent, whereas Δ is required to be terminating and Church-Rosser modulo B in order to guarantee the existence and unicity (modulo B) of a canonical form w.r.t. Δ for any term [12].

Formally, $\rightarrow_{R,B}$ and $\rightarrow_{\Delta,B}$ are defined as follows. Given a rewrite rule $r = (\lambda \rightarrow \rho \; \textit{if} \; C) \in R$ (resp., an equation $e = (\lambda = \rho \; \textit{if} \; C) \in \Delta$), a substitution σ, a term t, and a position w of t, $t \overset{r,\sigma,w}{\rightarrow}_{R,B} t'$ (resp., $t \overset{e,\sigma,w}{\rightarrow}_{\Delta,B} t'$) iff $\lambda\sigma =_B t_{|w}$, $t' = t[\rho\sigma]_w$, and C *evaluates to true* w.r.t σ. When no confusion can arise, we simply write $t \rightarrow_{R,B} t'$ (resp. $t \rightarrow_{\Delta,B} t'$) instead of $t \overset{r,\sigma,w}{\rightarrow}_{R,B} t'$ (resp. $t \overset{e,\sigma,w}{\rightarrow}_{\Delta,B} t'$).

Note that the evaluation of a condition C is typically a recursive process, since it may involve further (conditional) rewrites in order to normalize C to *true*. Specifically, an equational condition e *evaluates to true* w.r.t. σ if $e\sigma{\downarrow_\Delta} =_B$ *true*; a matching equation $p := t$ *evaluates to true* w.r.t. σ if $p\sigma =_B t\sigma{\downarrow_\Delta}$; a rewrite expression $t \Rightarrow p$ *evaluates to true* w.r.t. σ if there exists a rewrite sequence $t\sigma \rightarrow^*_{R\cup\Delta,B} u$, such that $u =_B p\sigma$[3]. Although rewrite expressions and matching/equational conditions can be intermixed in any order, we assume that their satisfaction is attempted sequentially from left to right, as in Maude.

[3] Technically, to properly evaluate a rewrite expression $t \Rightarrow p$ or a matching condition $p := t$, the term p is required to be a Δ-pattern —i.e., a term p such that, for every substitution σ, if $x\sigma$ is a canonical form w.r.t. Δ for every $x \in Dom(\sigma)$, then $p\sigma$ is also a canonical form w.r.t. Δ.

Under appropriate conditions on the rewrite theory, a rewrite step modulo E on a term t can be implemented without loss of completeness by applying the following rewrite strategy [13]: (i) reduce t w.r.t. $\rightarrow_{\Delta,B}$ until the canonical form $t\downarrow_{\Delta}$ is reached; (ii) rewrite $t\downarrow_{\Delta}$ w.r.t. $\rightarrow_{R,B}$.

An *execution trace* $\mathcal{T}$ in the rewrite theory $(\Sigma, \Delta \cup B, R)$ is a rewrite sequence

$$s_0 \rightarrow^*_{\Delta,B} s_0\downarrow_{\Delta} \rightarrow_{R,B} s_1 \rightarrow^*_{\Delta,B} s_1\downarrow_{\Delta} \ldots$$

that interleaves $\rightarrow_{\Delta,B}$ rewrite steps and $\rightarrow_{R,B}$ steps following the strategy mentioned above.

Given an execution trace $\mathcal{T}$, it is always possible to expand $\mathcal{T}$ in an *instrumented* trace $\mathcal{T}'$ in which every application of the matching modulo B algorithm is mimicked by the explicit application of a suitable equational axiom, which is also oriented as a rewrite rule [3]. This way, any given instrumented execution trace consists of a sequence of (standard) rewrites using the conditional equations ($\rightarrow_{\Delta}$), conditional rules ($\rightarrow_{R}$), and axioms ($\rightarrow_{B}$).

Example 3. Consider the rewrite theory in Example 2 together with the following execution trace $\mathcal{T}$: `credit(A,2+3);<A|10>` $\rightarrow_{\Delta,B}$ `credit(A,5);<A|10>` $\rightarrow_{R,B}$ `<A|15>` Thus, the corresponding instrumented execution trace is given by expanding the commutative "step" applied to the term `credit(A,2+3);<A|10>` using the implicit rule (`X;Y` $\rightarrow$ `Y;X`) in B that models the commutativity axiom for the (juxtaposition) operator _;_.

`credit(A,2+3);<A|10>` $\rightarrow_{\Delta}$ `credit(A,5);<A|10>` $\rightarrow_{B}$ `<A|10>;credit(A,5)` $\rightarrow_{R}$ `<A|15>`

Also, typically hidden inside the B-matching algorithms, some transformations allow terms that contain operators that obey associative-commutative axioms to be rewritten by first producing a single representative of their AC congruence class [3]. For example, consider a binary AC operator f together with the standard lexicographic ordering over symbols. Given the B-equivalence $f(b, f(f(b, a), c)) =_B f(f(b, c), f(a, b))$, we can represent it by using the "internal sequence" of transformations $f(b, f(f(b, a), c)) \rightarrow^*_{flat_B} f(a, b, b, c) \rightarrow^*_{unflat_B} f(f(b, c), f(a, b))$, where the first one corresponds to a *flattening* transformation sequence that obtains the AC canonical form, while the second one corresponds to the inverse, unflattening one.

In the sequel, we assume all execution traces are instrumented as explained above. By abuse of notation, we frequently denote the rewrite relations $\rightarrow_{\Delta}$, $\rightarrow_{R}$, $\rightarrow_{B}$ by $\rightarrow$. Also, by $\rightarrow^*$ (resp. $\rightarrow^+$), we denote the transitive and reflexive (resp. transitive) closure of the relation $\rightarrow_{\Delta} \cup \rightarrow_{R} \cup \rightarrow_{B}$.

4 Backward Conditional Slicing

In this section, we formulate our backward conditional slicing algorithm for RWL computations. The algorithm is formalized by means of a transition system that traverses the execution traces from back to front. The transition system is given by a single inference rule that relies on a *backward rewrite step slicing* procedure that is based on substitution refinement.

4.1 Term Slices and Term Slice Concretizations

A term slice of a term t is a term abstraction that disregards part of the information in t, that is, the irrelevant data in t are simply replaced by special $\bullet$-variables, denoted by $\bullet_i$, with $i = 0, 1, 2, \ldots$, which are generated by calling the auxiliary function $fresh^{\bullet}$[4]. More formally, a term slice is defined as follows.

Definition 1 (term slice). *Let $t \in \tau(\Sigma, \mathcal{V})$ be a term, and let P be a set of positions s.t. $P \subseteq \mathcal{P}os(t)$. A* term slice *of t w.r.t. P is defined as follows:*

$$slice(t, P) = rslice(t, P, \Lambda), \text{ where}$$

$$rslice(t, P, p) = \begin{cases} f(rslice(t_1, P, p.1), .., rslice(t_n, P, p.n)) & \text{if } t = f(t_1, .., t_n) \text{ and } p \in \bar{P} \\ x & \text{if } t = x \text{ and } x \in \mathcal{V} \text{ and } p \in \bar{P} \\ fresh^{\bullet} & \text{otherwise} \end{cases}$$

When P is understood, a term slice of t w.r.t. P is simply denoted by $t^{\bullet}$.

Roughly speaking, a term slice t w.r.t. a set of positions P includes all symbols of t that occur within the paths from the root to any position in P, while each maximal subterm $t_{|p}$, with $p \notin P$, is abstracted by means of a $\bullet$-variable.

Given a term slice $t^{\bullet}$, a *meaningful* position p of $t^{\bullet}$ is a position $p \in \mathcal{P}os(t^{\bullet})$ such that $t^{\bullet}_{|p} \neq \bullet_i$, for some $i = 0, 1, \ldots$. By $\mathcal{MP}os(t^{\bullet})$, we denote the set that contains all the meaningful positions of $t^{\bullet}$. Symbols that occur at meaningful positions are called *meaningful* symbols.

Example 4. Let $t = d(f(g(a, h(b)), c), a)$ be a term, and let $P = \{1.1,\ 1.2\}$ be a set of positions of t. By applying Definition 1, we get the term slice $t^{\bullet} = slice(t, P) = d(f(g(\bullet_1, \bullet_2), y), \bullet_3)$ and the set of meaningful positions $\mathcal{MP}os(t^{\bullet}) = \{\Lambda,\ 1,\ 1.1,\ 1.2\}$.

Now we show how we particularize a term slice, i.e., we instantiate $\bullet$-variables with data that satisfy a given boolean condition that we call *compatibility* condition. Term slice concretization is formally defined as follows.

Definition 2 (term slice concretization). *Let $t, t' \in \tau(\Sigma, \mathcal{V})$ be two terms. Let $t^{\bullet}$ be a term slice of t and let $B^{\bullet}$ be a boolean condition. We say that t' is a* concretization *of $t^{\bullet}$ that is compatible with $B^{\bullet}$ (in symbols $t^{\bullet} \propto^{B^{\bullet}} t'$), if (i) there exists a substitution σ such that $t^{\bullet}\sigma = t'$, and (ii) $B^{\bullet}\sigma$ evaluates to true.*

Example 5. Let $t^{\bullet} = \bullet_1 + \bullet_2 + \bullet_2$ and $B^{\bullet} = (\bullet_1 > 6 \wedge \bullet_2 \leq 7)$. Then, $10 + 2 + 2$ is a concretization of $t^{\bullet}$ that is compatible with $B^{\bullet}$, while $4 + 2 + 2$ is not.

In the following, we formulate a backward trace slicing algorithm that, given an execution trace $\mathcal{T} : s_0 \rightarrow^* s_n$ and a term slice $s^{\bullet}_n$ of s_n, generates the sliced counterpart $\mathcal{T}^{\bullet} : s^{\bullet}_0 \rightarrow^* s^{\bullet}_n$ of $\mathcal{T}$ that only encodes the information required to reproduce (the meaningful symbols of) the term slice $s^{\bullet}_n$. Additionally, the algorithm returns a companion compatibility condition $B^{\bullet}$ that guarantees the soundness of the generated trace slice.

[4] Each invocation of $fresh^{\bullet}$ returns a (fresh) variable $\bullet_i$, which is distinct from any previously generated variable $\bullet_j$.

4.2 Backward Slicing for Execution Traces

Consider an execution trace $\mathcal{T} : s_0 \rightarrow^* s_n$. A trace slice $\mathcal{T}^\bullet$ of $\mathcal{T}$ is defined w.r.t. a *slicing criterion* — i.e., a set of positions $\mathcal{O}_{s_n} \subseteq \mathcal{P}os(s_n)$ that refer to those symbols of s_n that we want to observe. Basically, the trace slice $\mathcal{T}^\bullet$ of $\mathcal{T}$ is obtained by removing all the information from $\mathcal{T}$ that is not required to produce the term slice $s_n^\bullet = slice(s_n, \mathcal{O}_{s_n})$. A trace slice is formally defined as follows.

Definition 3. *Let $\mathcal{R} = (\Sigma, \Delta \cup B, R)$ be a conditional rewrite theory, and let $\mathcal{T} : s_0 \overset{r_1,\sigma_1,w_1}{\rightarrow} s_1 \overset{r_2,\sigma_2,w_2}{\rightarrow} \ldots \overset{r_n,\sigma_n,w_n}{\rightarrow} s_n$ be an execution trace in $\mathcal{R}$. Let $\mathcal{O}_{s_n}$ be a slicing criterion for $\mathcal{T}$. A* trace slice *of $\mathcal{T}$ w.r.t. $\mathcal{O}_{s_n}$ is a pair $[s_0^\bullet \rightarrow s_1^\bullet \rightarrow \ldots \rightarrow s_n^\bullet,\ B^\bullet]$, where*

1. *$s_i^\bullet$ is a term slice of s_i, for $i = 0, \ldots, n$, and $B^\bullet$ is a boolean condition;*
2. *$s_n^\bullet = slice(s_n, \mathcal{O}_{s_n})$;*
3. *for every term s'_0 such that $s_0^\bullet \propto^{B^\bullet} s'_0$, there exists an execution trace $s'_0 \rightarrow s'_1 \rightarrow \ldots \rightarrow s_n$ in $\mathcal{R}$ such that*
   i) $s'_i \rightarrow s'_{i+1}$ is either the rewrite step $s'_i \overset{r_{i+1},\sigma'_{i+1},w_{i+1}}{\rightarrow} s'_{i+1}$ or $s'_i = s'_{i+1}$, $i = 0, \ldots, n-1$;
   ii) $s_i^\bullet \propto^{B^\bullet} s'_i$, $i = 1, \ldots, n$.

Note that Point 3 of Definition 3 ensures that the rules involved in the sliced steps of $\mathcal{T}^\bullet$ can be applied again, at the corresponding positions, to every concrete trace $\mathcal{T}'$ that can be obtained by instantiating all the $\bullet$-variables in $s_0^\bullet$ with arbitrary terms. The following example illustrates the slicing of an execution trace.

Example 6. Consider the Maude specification of Example 2 together with the following execution trace $\mathcal{T}$: `(<a|30>;debit(a,5));credit(a,3)` $\overset{\texttt{debit}}{\rightarrow}$ `<a|25>;credit(a,3)` $\overset{\texttt{credit}}{\rightarrow}$ `<a|28>`. Let `<a|`$\bullet_1$`>` be a term slice of `<a|28>` generated with the slicing criterion $\{1\}$ —i.e., `<a|`$\bullet_1$`>`$=slice($`<a|28>`$, \{1\})$. Then, the trace slice for $\mathcal{T}$ is $[\mathcal{T}^\bullet,\ \bullet_8 \geq \bullet_9]$ where $\mathcal{T}^\bullet$ is as follows

`(<a|`$\bullet_8$`>;debit(a,`$\bullet_9$`));credit(a|`$\bullet_4$`)` $\overset{\texttt{debit}}{\rightarrow}$ `<a|`$\bullet_3$`>;credit(a,`$\bullet_4$`)` $\overset{\texttt{credit}}{\rightarrow}$ `<a|`$\bullet_1$`>`

Note that $\mathcal{T}^\bullet$ needs to be endowed with the compatibility condition $\bullet_8 \geq \bullet_9$ in order to ensure the applicability of the `debit` rule. In other words, any instance $s^\bullet\sigma$ of `<a|`$\bullet_8$`>;debit(a,`$\bullet_9$`)` can be rewritten by the `debit` rule only if $\bullet_8\sigma \geq \bullet_9\sigma$.

Informally, given a slicing criterion $\mathcal{O}_{s_n}$ for the execution trace $\mathcal{T} = s_0 \rightarrow^* s_n$, at each rewrite step $s_{i-1} \rightarrow s_i$, $i = n, \ldots, 1$, our technique inductively computes the association between the meaningful information of s_i and the meaningful information in s_{i-1}. For each such rewrite step, the conditions of the applied rule are recursively processed in order to ascertain from s_i the meaningful information in s_{i-1}, together with the accumulated condition $B_i^\bullet$. The technique proceeds backwards, from the final term s_n to the initial term s_0. A simplified trace is obtained where each s_i is replaced by the corresponding term slice $s_i^\bullet$.

We define a transition system $(Conf,\ \bullet\!\!\rightarrow)$ [17] where *Conf* is a set of *configurations* and $\bullet\!\!\rightarrow$ is the transition relation that implements the backward trace slicing algorithm. Configurations are formally defined as follows.

Definition 4. *A configuration, written as* $\langle \mathcal{T},\ S^\bullet,\ B^\bullet \rangle$*, consists of three components:*

- *the execution trace* $\mathcal{T} : s_0 \rightarrow^* s_{i-1} \rightarrow s_i$ *to be sliced;*
- *the term slice* $s_i^\bullet$*, that records the computed term slice of* s_i
- *a boolean condition* $B^\bullet$*.*

The transition system $(Conf,\ \bullet\!\!\rightarrow)$ is defined as follows.

Definition 5. *Let* $\mathcal{R} = (\Sigma, \Delta \cup B, R)$ *be a rewrite theory, let* $\mathcal{T} = U \rightarrow^* W$ *be an execution trace in* $\mathcal{R}$*, and let* $V \rightarrow W$ *be a rewrite step. Let* $B_W^\bullet$ *and* $B_V^\bullet$ *be two boolean conditions, and* $W^\bullet$ *be a term slice of* W*. Then, the transition relation* $\bullet\!\!\rightarrow \subseteq Conf \times Conf$ *is the smallest relation that satisfies the following rule:*

$$\frac{(V^\bullet,\ B_V^\bullet) = \textit{slice-step}(V \rightarrow W,\ W^\bullet,\ B_W^\bullet)}{\langle U \rightarrow^* V \rightarrow W,\ W^\bullet,\ B_W^\bullet \rangle \bullet\!\!\rightarrow \langle U \rightarrow^* V,\ V^\bullet,\ B_V^\bullet \rangle}$$

Roughly speaking, the relation $\bullet\!\!\rightarrow$ transforms a configuration $\langle U \rightarrow^* V \rightarrow W,\ W^\bullet,\ B_W^\bullet \rangle$ into a configuration $\langle U \rightarrow^* V,\ V^\bullet,\ B_V^\bullet \rangle$ by calling the function *slice-step*$(V \rightarrow W,\ W^\bullet,\ B_W^\bullet)$ of Section 4.3, which returns a rewrite step slice for $V \rightarrow W$. More precisely, *slice-step* computes a suitable term slice $V^\bullet$ of V and a boolean condition $B_V^\bullet$ that updates the compatibility condition specified by $B_W^\bullet$.

The initial configuration $\langle s_0 \rightarrow^* s_n,\ slice(s_n, \mathcal{O}_{s_n}),\ true \rangle$ is transformed until a terminal configuration $\langle s_0,\ s_0^\bullet,\ B_0^\bullet \rangle$ is reached. Then, the computed trace slice is obtained by replacing each term s_i by the corresponding term slice $s_i^\bullet$, $i = 0, \ldots, n$, in the original execution trace $s_0 \rightarrow^* s_n$. The algorithm additionally returns the accumulated compatibility condition $B_0^\bullet$ attained in the terminal configuration.

More formally, the backward trace slicing of an execution trace w.r.t. a slicing criterion is implemented by the function *backward-slicing* defined as follows.

Definition 6 (Backward trace slicing algorithm). *Let* $\mathcal{R} = (\Sigma, \Delta \cup B, R)$ *be a rewrite theory, and let* $\mathcal{T} : s_0 \rightarrow^* s_n$ *be an execution trace in* $\mathcal{R}$*. Let* $\mathcal{O}_{s_n}$ *be a slicing criterion for* $\mathcal{T}$*. Then, the function backward-slicing is computed as follows:*

$$\textit{backward-slicing}(s_0 \rightarrow^* s_n, \mathcal{O}_{s_n}) = [s_0^\bullet \rightarrow^* s_n^\bullet,\ B_0^\bullet]$$

iff there exists a transition sequence in $(Conf,\ \bullet\!\!\rightarrow)$

$$\langle s_0 \rightarrow^* s_n,\ s_n^\bullet,\ true \rangle \bullet\!\!\rightarrow \langle s_0 \rightarrow^* s_{n-1},\ s_{n-1}^\bullet,\ B_{n-1}^\bullet \rangle \bullet\!\!\rightarrow^* \langle s_0,\ s_0^\bullet,\ B_0^\bullet \rangle$$
$$\textit{where } s_n^\bullet = slice(s_n,\ \mathcal{O}_{s_n})$$

In the following, we formulate the auxiliary procedure for the slicing of conditional rewrite steps.

4.3 The Function *slice-step*

The function *slice-step*, which is outlined in Figure 2, takes as input three parameters, namely, a rewrite step $\mu : s \overset{r,\sigma,w}{\rightarrow} t$ (with $r = \lambda \rightarrow \rho$ *if* C[5]), a term slice $t^\bullet$ of t, and a compatibility condition $B^\bullet_{prev}$; and delivers the term slice $s^\bullet$ and a new compatibility condition $B^\bullet$. Within the algorithm *slice-step*, we use an auxiliary operator $\langle\!|\sigma_1, \sigma_2|\!\rangle$ that refines (overrides) a substitution σ_1 with a substitution σ_2, where both σ_1 and σ_2 may contain $\bullet$-variables. The main idea behind $\langle\!|\_, \_|\!\rangle$ is that, for the slicing of the step μ, all variables in the applied rewrite rule r are naïvely assumed to be initially bound to irrelevant data $\bullet$, and the bindings are incrementally refined as we (partially) solve the conditions of r.

```
function slice-step(s ^{r,σ,w}→ t, t•, B•_prev)
 1. if w ∉ MPos(t•)
 2.   then
 3.     s• = t•
 4.     B• = B•_prev
 5.   else
 6.     θ = {x/fresh• | x ∈ Var(r)}
 7.     ρ• = slice(ρ, MPos(t•|w))
 8.     ψρ = ⟪θ, match_{ρ•θ}(t•|w)⟫
 9.     for i = n downto 1 do
10.       (ψi, Bi•) = process-condition(ci, σ,
                                  ⟪ψρ, ψn...ψi+1⟫)
11.     od
12.     s• = t•[λ⟪ψρ, ψn...ψ1⟫]w
13.     B• = (B•_prev ∧ B•n ... ∧ B•1)(ψ1ψ2 ... ψn)
14. fi
15. return (s•, B•)
```

Fig. 2. Backward step slicing function

Definition 7 (refinement). *Let σ_1 and σ_2 be two substitutions. The refinement of σ_1 w.r.t. σ_2 is defined by the operator $\langle\!|\_, \_|\!\rangle$ as follows: $\langle\!|\sigma_1, \sigma_2|\!\rangle = \sigma_{\restriction Dom(\sigma_1)}$, where*

$$x\sigma = \begin{cases} x\sigma_2 & \text{if } x \in Dom(\sigma_1) \cap Dom(\sigma_2) \\ x\sigma_1\sigma_2 & \text{if } x \in Dom(\sigma_1) \setminus Dom(\sigma_2) \wedge \sigma_2 \neq \mathit{fail} \\ x\sigma_1 & \text{otherwise} \end{cases}$$

Note that $\langle\!|\sigma_1, \sigma_2|\!\rangle$ differs from the (standard) instantiation of σ_1 with σ_2. We write $\langle\!|\sigma_1, \ldots, \sigma_n|\!\rangle$ as a compact denotation for $\langle\!|\langle\!| \ldots \langle\!|\sigma_1, \sigma_2|\!\rangle, \ldots, \sigma_{n-1}|\!\rangle, \sigma_n|\!\rangle$.

Example 7. Let $\sigma_1 = \{x/\bullet_1,\ y/\bullet_2\}$ and $\sigma_2 = \{x/a,\ \bullet_2/g(\bullet_3),\ z/5\}$ be two substitutions. Thus, $\langle\!|\sigma_1, \sigma_2|\!\rangle = \{x/a,\ y/g(\bullet_3)\}$.

Roughly speaking, the function *slice-step* works as follows. When the rewrite step μ occurs at a position w that is not a meaningful position of $t^\bullet$ (in symbols, $w \notin \mathcal{MP}os(t^\bullet)$), trivially μ does not contribute to producing the meaningful symbols of $t^\bullet$. Therefore, the function returns $s^\bullet = t^\bullet$, with the input compatibility condition $B^\bullet_{prev}$.

Example 8. Consider the Maude specification of Example 2 and the following rewrite step μ: `(<a|30>;debit(a,5));credit(a,3)` $\overset{\texttt{debit}}{\rightarrow}$ `<a|25>;credit(a,3)`. Let $\bullet_1$; `credit(a,3)` be a term slice of `<a|25>;credit(a,3)`. Since the rewrite step μ occurs at position $1 \notin \mathcal{MP}os(\bullet_1;\texttt{credit(a,3)})$, the term `<a|25>` introduced by μ in `<a|25>;credit(a,3)` is completely ignored in $\bullet_1$; `credit(a,3)`. Hence, the

[5] Since equations and axioms are both interpreted as rewrite rules in our formulation, we often abuse the notation $\lambda \rightarrow \rho$ *if* C to denote rules as well as (oriented) equations and axioms.

computed term slice for `(<a|30>;debit(a,5));credit(a,3)` is the very same $\bullet_1$; `credit(a,3)`.

On the other hand, when $w \in \mathcal{MP}os(t^\bullet)$, the computation of $s^\bullet$ and $B^\bullet$ involves a more in-depth analysis of the rewrite step, which is based on an inductive refinement process that is obtained by recursively processing the conditions of the applied rule.

More specifically, we initially define the substitution $\theta = \{x/fresh^\bullet \mid x \in Var(r)\}$ that binds each variable in r to a fresh $\bullet$-variable. This corresponds to assuming that all the information in μ, which is introduced by the substitution σ, can be marked as irrelevant. Then, θ is incrementally refined using the following two-step procedure.

```
function process-condition(c, σ, θ)
1.  case c of
2.  (p := t) ∨ (t ⇒ p) :
3.    if (tσ = pσ)
4.      then return ({}, true) fi
5.    Q = MPos(pθ)
6.    [t• →* p•, B•] =
          backward-slicing(tσ →* pσ, Q)
7.    t•′ = slice(t, MPos(t•))
8.    ψ = match_{t•′θ}(t•)
9.  e :
10.   ψ = { }
12.   B• = eθ
12. end case
13. return (ψ, B•)
```

Fig. 3. Condition processing function

Step 1. We compute the matcher $match_{\rho\theta}(t^\bullet_{|w})$, and then generate the refinement ψ_ρ of θ w.r.t. $match_{\rho\theta}(t^\bullet_{|w})$ (in symbols, $\psi_\rho = \langle\!|\theta, match_{\rho\theta}(t^\bullet_{|w})|\!\rangle$). Roughly speaking, the refinement ψ_ρ updates the bindings of θ with the meaningful information extracted from $t^\bullet_{|w}$.

Example 9. Consider the rewrite theory in Example 2 together with the following rewrite step $\mu_{\texttt{debit}}$: `<a|30>;debit(a,5)` $\overset{\texttt{debit}}{\rightarrow}$ `<a|25>` that involves the application of the `debit` rule whose right-hand side is $\rho_{\texttt{debit}}$ =`<Id|newBal>`. Let $t^\bullet$ =`<a|`$\bullet_1$`>` be a term slice of `<a|25>`. Then, the initially ascertained substitution for μ is θ=$\{$`Id/`$\bullet_2$, `bal/`$\bullet_3$, `M/`$\bullet_4$, `newBal/`$\bullet_5\}$, and $match_{\rho_{\texttt{debit}}\theta}(t^\bullet) = match_{\texttt{<}\bullet_2\texttt{|}\bullet_5\texttt{>}}$(`<a|`$\bullet_1$`>`) = $\{$$\bullet_2$`/a`, $\bullet_5$ /$\bullet_1\}$. Thus, the substitution $\psi_{\rho_{\texttt{debit}}} = \langle\!|\theta, \psi_{\rho_{\texttt{debit}}}|\!\rangle = \{$`Id/a, bal/`$\bullet_3$, `M/`$\bullet_4$, `newBal/`$\bullet_1\}$. That is, $\psi_{\rho_{\texttt{debit}}}$ refines θ by replacing the uninformed binding `Id/`$\bullet_2$, with `Id/a`.

Step 2. Let $C\sigma = c_1\sigma \wedge \ldots \wedge c_n\sigma$ be the instance of the condition in the rule r that enables the rewrite step μ. We process each (sub)condition $c_i\sigma$, $i = 1, \ldots, n$, in reversed evaluation order, i.e., from $c_n\sigma$ to $c_1\sigma$, by using the auxiliary function *process-condition* given in Figure 3 that generates a pair $(\psi_i,\ B_i^\bullet)$ such that ψ_i is used to further refine the partially ascertained substitution $\langle\!|\psi_\rho, \psi_n, \ldots, \psi_{i+1}|\!\rangle$ computed by incrementally analyzing conditions $c_n\sigma, c_{n-1}\sigma \ldots, c_{i+1}\sigma$, and $B_i^\bullet$ is a boolean condition that is derived from the analysis of the condition c_i.

When the whole $C\sigma$ has been processed, we get the refinement $\langle\!|\psi_\rho, \psi_n, \ldots, \psi_1|\!\rangle$, which basically encodes all the instantiations required to construct the term slice $s^\bullet$ from $t^\bullet$. More specifically, $s^\bullet$ is obtained from $t^\bullet$ by replacing the subterm $t^\bullet_{|w}$ with the left-hand side λ of r instantiated with $\langle\!|\psi_\rho, \psi_n, \ldots, \psi_1|\!\rangle$. Furthermore, $B^\bullet$ is built by collecting all the boolean compatibility conditions $B_i^\bullet$ delivered by *process-condition* and instantiating them with the composition of the computed

refinements $\psi_1 \ldots \psi_n$. It is worth noting that *process-condition* handles rewrite expressions, equational conditions, and matching conditions differently. More specifically, the pair $(\psi_i,\ B_i)$ that is returned after processing each condition c_i is computed as follows.

– **Matching Conditions.** Let c be a matching condition with the form $p := m$ in the condition of rule r. During the execution of the step $\mu : s \stackrel{r,\sigma,w}{\rightarrow} t$, recall that c is evaluated as follows: first, $m\sigma$ is reduced to its canonical form $m\sigma{\downarrow_\Delta}$, and then the condition $m\sigma \downarrow_\Delta =_B p\sigma$ is checked. Therefore, the analysis of the matching condition $p := m$ during the slicing process of μ implies slicing the (internal) execution trace $\mathcal{T}_{int} = m\sigma \rightarrow^* p\sigma$, which is done by recursively invoking the function *backward-slicing* for execution trace slicing with respect to the meaningful positions of the term slice $p\theta$ of p, where θ is a refinement that records the meaningful information computed so far. That is, $[m^\bullet \rightarrow^* p^\bullet,\ B^\bullet] = \mathit{backward\text{-}slicing}(m\sigma \rightarrow^* p\sigma,\ \mathcal{MP}os(p\theta))$. The result delivered by the function *backward-slicing* is a trace slice $m^\bullet \rightarrow^* p^\bullet$ with compatibility condition $B^\bullet$.

  In order to deliver the final outcome for the matching condition $p := m$, we first compute the substitution $\psi = \mathit{match}_{m\theta}(m^\bullet)$, which is the substitution needed to refine θ, and then the pair $(\psi,\ B^\bullet)$ is returned.

  Example 10. Consider the the rewrite step $\mu_{\texttt{debit}}$ of Example 9 together with the refined substitution $\theta = \{\texttt{Id/a},\ \texttt{bal}/\bullet_3,\ \texttt{M}/\bullet_4,\ \texttt{newBal}/\bullet_1\}$. We process the condition `newBal := bal - M` of `debit` by considering an internal execution trace $\mathcal{T}_{int} = \texttt{30} - \texttt{5} \rightarrow \texttt{25}$ [6]. By invoking the function *backward-slicing* the trace slice result is $[\bullet_6 \rightarrow \bullet_6,\ \mathit{true}]$. The final outcome is given by $\mathit{match}_{\bullet_7 - \bullet_8}(\bullet_6)$, that is fail. Thus θ does not need any further refinement.

– **Rewrite Expressions.** The case when c is a rewrite expression $t \Rightarrow p$ is handled similarly to the case of a matching equation $p := t$, with the difference that t can be reduced by using the rules of R in addition to equations and axioms.

– **Equational Conditions.** During the execution of the rewrite step $\mu : s \stackrel{r,\sigma,w}{\rightarrow} t$, the instance $e\sigma$ of an equational condition e in the condition of the rule r is just fulfilled or falsified, but it does not bring any instantiation into the output term t. Therefore, when processing $e\sigma$, no meaningful information to further refine the partially ascertained substitution θ must be added. However, the equational condition e must be recorded in order to compute the compatibility condition $B^\bullet$ for the considered conditional rewrite step. In other words, after processing an equational condition e, we deliver the tuple $(\psi,\ B^\bullet)$, with $\psi = \{\ \}$ and $B^\bullet = e\theta$. Note that the condition e is instantiated with the updated substitution θ, in order to transfer only the meaningful information of $e\sigma$ computed so far in e.

[6] Note that the trace `30-5→25` involves an application of the Maude built-in operator "`-`". Given a built-in operator `op`, in order to handle the reduction `a op b` → `c` as an ordinary rewrite step, we add the rule `a op b` ⇒ `c` to the considered rewrite theory.

Example 11. Consider the refined substitution given in Example 10 $\theta = \{\texttt{Id/a},\ \texttt{bal/}\bullet_3,\ \texttt{M/}\bullet_4,\ \texttt{newBal/}\bullet_1\}$ together with the rewrite step $\mu_{\texttt{debit}}$ of Example 9 that involves the application of the `debit` rule. After processing the condition `bal >= M` of `debit`, we deliver $B^\bullet = (\bullet_3 >= \bullet_4)$.

Soundness of our conditional slicing technique is established by the following theorem. The proof can be found in [4].

Theorem 1 (soundness). *Let $\mathcal{R}$ be a rewrite theory. Let $\mathcal{T} : s_0 \overset{r_1,\sigma_1,w_1}{\rightarrow} \ldots \overset{r_n,\sigma_n,w_n}{\rightarrow} s_n$ be an execution trace in the rewrite theory $\mathcal{R}$, with $n > 0$, and let $\mathcal{O}_{s_n}$ be a slicing criterion for $\mathcal{T}$. Then, the pair $[s_0^\bullet \rightarrow \ldots \rightarrow s_n^\bullet,\ B_0^\bullet]$ computed by backward-slicing($\mathcal{T}$, $\mathcal{O}_{s_n}$) is a trace slice for $\mathcal{T}$.*

5 Implementation and Experimental Evaluation

The conditional slicing methodology presented so far has been implemented in a prototype tool that is written in Maude and publicly available at `http://users.dsic.upv.es/grupos/elp/soft.html`. The prototype takes in input a slicing criterion and a Maude execution trace, which is a term of sort `Trace` (generated by means of the the Maude metalevel operator `metaSearchPath`), and delivers the corresponding trace slice.

We have tested our prototype on rather large execution traces, such as the counterexamples generated by the model checker for Web applications WEB-TLR [2]. In our experiments, we have considered a Webmail application together with four LTLR properties that have been refuted by Web-TLR. For each refuted property, WEB-TLR has produced the corresponding counterexample in the form of a huge, textual execution trace $\mathcal{T}_i$, $i = 1, ..., 4$, in the range $10 - 100Kb$ that has been used to feed our slicer.

Table 1. Backward trace slicing benchmarks

Example trace	Original trace size	Slicing criterion	Sliced trace size	% *reduction*
Web-TLR.$\mathcal{T}_1$	19114	Web-TLR.$\mathcal{T}_1$.O_1	3982	79.17%
		Web-TLR.$\mathcal{T}_1$.O_2	3091	83.83%
Web-TLR.$\mathcal{T}_2$	22018	Web-TLR.$\mathcal{T}_2$.O_1	2984	86.45%
		Web-TLR.$\mathcal{T}_2$.O_2	2508	88.61%
Web-TLR.$\mathcal{T}_3$	38983	Web-TLR.$\mathcal{T}_3$.O_1	2045	94.75%
		Web-TLR.$\mathcal{T}_3$.O_2	2778	92.87%
Web-TLR.$\mathcal{T}_4$	69491	Web-TLR.$\mathcal{T}_4$.O_1	8493	87.78%
		Web-TLR.$\mathcal{T}_4$.O_2	5034	92.76%

Table 1 shows the size of the original counterexample trace and that of the computed trace slice, both measured as the length of the corresponding string, w.r.t. two slicing criteria, that are detailed in the tool website. The considered criteria allow one to monitor the messages exchanged by a specific Web browser and the Webmail server, as well as to isolate the changes on the data structures of the two interacting entities. The *%reduction* column in Table 1 refers to the percentage of reduction achieved. The results we have obtained are very encouraging, and show an impressive reduction rate (up to $\sim 95\%$) in reasonable time (max. 0.9s on a Linux box equipped with an Intel Core 2 Duo 2.26GHz and 4Gb of RAM memory). Actually, sometimes the trace slices are small enough to be easily inspected by the users, who can restrict their attention to the part of the computation that they want to observe.

References

1. Alpuente, M., Ballis, D., Espert, J., Frechina, F., Romero, D.: Debugging of Web Applications with WEB-TLR. In: 7th Int'l Workshop on Automated Specification and Verification of Web Systems WWV 2011. EPTCS, vol. 61, pp. 66–80 (2011)
2. Alpuente, M., Ballis, D., Espert, J., Romero, D.: Model-Checking Web Applications with Web-TLR. In: Bouajjani, A., Chin, W. (eds.) ATVA 2010. LNCS, vol. 6252, pp. 341–346. Springer, Heidelberg (2010)
3. Alpuente, M., Ballis, D., Espert, J., Romero, D.: Backward Trace Slicing for Rewriting Logic Theories. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS (LNAI), vol. 6803, pp. 34–48. Springer, Heidelberg (2011)
4. Alpuente, M., Ballis, D., Frechina, F., Romero, D.: Trace Slicing of Conditional Rewrite Theories. Tech. rep., Universidad Politécnica de Valencia (2012)
5. Alpuente, M., Ballis, D., Romero, D.: Specification and Verification of Web Applications in Rewriting Logic. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 790–805. Springer, Heidelberg (2009)
6. Bae, K., Meseguer, J.: A Rewriting-Based Model Checker for the Linear Temporal Logic of Rewriting. In: 9th Int'l Workshop on Rule-Based Programming RULE 2008. ENTCS. Elsevier (2008)
7. Baggi, M., Ballis, D., Falaschi, M.: Quantitative Pathway Logic for Computational Biology. In: Degano, P., Gorrieri, R. (eds.) CMSB 2009. LNCS, vol. 5688, pp. 68–82. Springer, Heidelberg (2009)
8. Bethke, I., Klop, J.W., de Vrijer, R.: Descendants and origins in term rewriting. Inf. Comput. 159(1-2), 59–124 (2000)
9. Bruni, R., Meseguer, J.: Semantic Foundations for Generalized Rewrite Theories. Theoretical Computer Science 360(1–3), 386–414 (2006)
10. Chen, F., Rosu, G.: Parametric Trace Slicing and Monitoring. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 246–261. Springer, Heidelberg (2009)
11. Chitil, O., Runciman, C., Wallace, M.: Freja, Hat and Hood - A Comparative Evaluation of Three Systems for Tracing and Debugging Lazy Functional Programs. In: Mohnen, M., Koopman, P. (eds.) IFL 2000. LNCS, vol. 2011, pp. 176–193. Springer, Heidelberg (2001)
12. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: Maude Manual (Version 2.6). Tech. rep., SRI Int'l Computer Science Laboratory (2011), http://maude.cs.uiuc.edu/maude2-manual/
13. Durán, F., Meseguer, J.: A Maude Coherence Checker Tool for Conditional Order-Sorted Rewrite Theories. In: Ölveczky, P.C. (ed.) WRLA 2010. LNCS, vol. 6381, pp. 86–103. Springer, Heidelberg (2010)
14. Field, J., Tip, F.: Dynamic Dependence in Term Rewriting Systems and its Application to Program Slicing. In: Hermenegildo, M., Penjam, J. (eds.) PLILP 1994. LNCS, vol. 844, pp. 415–431. Springer, Heidelberg (1994)
15. Klop, J.: Term Rewriting Systems. In: Abramsky, S., Gabbay, D., Maibaum, T. (eds.) Handbook of Logic in Computer Science, vol. I, pp. 1–112. Oxford University Press (1992)
16. Meseguer, J.: Conditional Rewriting Logic as a Unified Model of Concurrency. Theoretical Computer Science 96(1), 73–155 (1992)
17. Plotkin, G.D.: A structural approach to operational semantics. J. Log. Algebr. Program., 17–139 (2004)
18. TeReSe (ed.): Term Rewriting Systems. Cambridge University Press, Cambridge (2003)

Forgetting for Defeasible Logic

Grigoris Antoniou[1], Thomas Eiter[2], and Kewen Wang[3]

[1] FORTH-ICS, Greece and University of Huddersfield, UK
antoniou@ics.forth.gr
[2] Institut für Informationssysteme, Technische Universität Wien, Austria
eiter@kr.tuwien.ac.at
[3] School of Information and Communication Technology, Griffith University, Australia
k.wang@griffith.edu.au

Abstract. The concept of *forgetting* has received significant interest in artificial intelligence recently. Informally, given a knowledge base, we may wish to forget about (or discard) some redundant parts (such as atoms, predicates, concepts, etc) but still preserve the consequences for certain forms of reasoning. In nonmonotonic reasoning, so far forgetting has been studied only in the context of extension based approaches, mainly answer-set programming. In this paper forgetting is studied in the context of defeasible logic, which is a simple, efficient and sceptical nonmonotonic reasoning approach.

1 Introduction

The concept of *forgetting* has received significant interest in Artificial Intelligence recently. Informally, given a knowledge base, we may wish to forget about (or discard) some redundant parts (such as atoms, predicates, concepts, etc) but still preserve the consequences for certain forms of reasoning. Forgetting has been introduced in many formalisms for knowledge representation, for instance, in propositional logic [17,18], first-order logic [19,31], modal logic [25,14,30], description logic [28,27,16], and logic programming [7,26,29]. The theory of forgetting has also been fruitfully applied in various contexts, e.g. in cognitive robotics [19], for resolving conflicts and inconsistencies [18,7], and in particular, in ontology engineering [15].

Regarding nonmonotonic logics, so far forgetting has been studied only in the context of extension based approaches, mainly answer-set programming. On the other hand, defeasible reasoning is a nonmonotonic reasoning approach in which the gaps due to incomplete information are closed through the use of defeasible rules. Defeasible logics were introduced by Nute [23] and developed over several years [3,10,6]. They allow for defeasible reasoning, where a conclusion supported by a rule might be overturned by the effect of another rule; they also have a monotonic reasoning component, and a priority on rules. One feature of their design is that they are quite simple, allowing efficient reasoning; in fact, basic defeasible logics have linear time complexity [20]. Its properties have been thoroughly studied and analyzed, with strong results in terms of proof theory [22,3] and semantics [10,21].

Defeasible logic has recently attracted considerable interest. Its use in various application domains has been advocated, including the modeling of regulations and business

N. Bjørner and A. Voronkov (Eds.): LPAR-18, LNCS 7180, pp. 77–91, 2012.

rules [2], modeling of contracts [8], legal reasoning [13], agent negotiations [9,24], modeling of agents and agent societies [12,11], and applications to the Semantic Web [1,4] and ambient intelligence [5].

However, to our best knowledge, *a theory of forgetting for defeasible logic is still missing*. In this paper, we first examine a naive definition of forgetting for defeasible logic and explain why such a definition is insufficient for practical applications. Specifically, the naive definition does not preserve any syntactical information of the original theory. For this reason, we develop a new approach to forgetting for defeasible logic and establish some of its formal properties, including completeness, computational complexity, and structure preservation. Salient features of the solution provided include linear time complexity, and linear size of the output of (iterated) forgetting.

The paper is organised as follows. Section 2 presents the basics of defeasible logic. Section 3 introduces the problem of forgetting in the context of defeasible logic, and presents a first, naive solution. An analysis of the weaknesses of this approach leads to an improved approach, presented in Section 4, where its properties and computational complexity are also analyzed. We conclude the paper with plans for future work in Section 5.

2 Defeasible Logic

In this paper we restrict attention to essentially propositional defeasible logic. Rules with free variables are interpreted as rule schemas, that is, as the set of all ground instances. If q is a literal, $\sim q$ denotes the complementary literal (if q is a positive literal p then $\sim q$ is $\neg p$; and if q is $\neg p$, then $\sim q$ is p).

Rules are defined over a *language* (or *signature*) Σ, the set of propositions (atoms) and labels that may be used in the rule.

A rule $r : A(r) \hookrightarrow C(r)$ consists of its unique *label* r, its *antecedent* $A(r)$, which is a finite set of literals (possibly omitted if empty), an arrow $\hookrightarrow$ (which is a placeholder for concrete arrows to be introduced in a moment), and its *head* (or *consequent*) $C(r)$ which is a literal. In writing rules we omit set notation for antecedents, and sometimes we omit the label when it is not relevant for the context. There are three kinds of rules, each designated by a different arrow:

- $\rightarrow$ designates *strict rules* (definitional rules);
- $\Rightarrow$ designates *defeasible rules* (nonmonotonic rules that may be attacked by other rules); and
- $\rightsquigarrow$ designates *defeaters* (special rules that do not support positive conclusions but may only attack other rules).

Given a set R of rules, we denote by R_s the set of all strict rules in R, by R_{sd} the set of all strict and defeasible rules in R, and by $R[q]$ the set of rules in R with consequent q.

A *superiority relation on* R is a binary relation $>$ on R. When $r_1 > r_2$, then r_1 is called *superior* to r_2, and r_2 *inferior* to r_1. Intuitively, $r_1 > r_2$ expresses that r_1 overrules r_2, should both rules be applicable. Typically we assume $>$ to be acyclic (that is, the transitive closure of $>$ is irreflexive).

A *defeasible theory* is a triple $D = (F, R, >)$ where F is a finite set of literals (called *facts*), R a finite set of rules, and $>$ is an acyclic superiority relation on R. We call D *decisive*, if the atom dependency graph of D is acyclic.

Example 1. The following is a theory $D = (F, R, >)$ in defeasible logic, where

$F = \{\ \}$

$R = \{r_0 :\rightarrow d$
$r_1 :\Rightarrow a$
$r_2 :\Rightarrow \neg a$
$r_3 : a \Rightarrow b$
$r_4 :\Rightarrow \neg b$
$r_5 : d, b \Rightarrow c$
$r_6 :\Rightarrow \neg c \quad \},$

and $r_1 > r_2 \ \ r_3 > r_4$.

A *conclusion* of a theory D in defeasible logic is a tagged literal and can have one of the following four forms:

- $+\Delta q$, which is intended to mean that q is definitely provable in D.
- $-\Delta q$, which is intended to mean that we have proved that q is not definitely provable in D.
- $+\partial q$, which is intended to mean that q is defeasibly provable in D.
- $-\partial q$, which is intended to mean that we have proved that q is not defeasibly provable in D.

If we are able to prove q definitely, then q is also defeasibly provable. This is a direct consequence of the formal definition below. It resembles the situation in, say, Reiter's default logic: a formula is sceptically provable from a default theory $T = (W, D)$ (in the sense that it is included in each extension) if it is provable from the set W of classical formulas.

Let us consider the theory in Example 1. Intuitively, d can be derived by the strict rule r_0 and thus $+\Delta d$ is provable from the theory, and so is $+\partial d$. Also, $-\Delta\neg d$ is provable, as an attempt to prove $\neg d$ fails finitely. Similarly, $+\partial a$ is provable (as $r_1 > r_2$), and so is $-\partial\neg a$ is provable (defeasible logic is sceptical, and the rule supporting a is superior to the rule supporting $\neg a$).

Provability for defeasible logic is based on the concept of a *derivation* (or *proof*) in $D = (F, R, >)$. A *derivation* is a finite sequence $P = (P(1), ..., P(n))$ of tagged literals constructed by inference rules. There are four inference rules (corresponding to the four kinds of conclusion) that specify how a derivation may be extended. ($P(1..i)$ denotes the initial part of the sequence P of length i):

$+\Delta$: We may append $P(i+1) = +\Delta q$ if either
$q \in F$ or
$\exists r \in R_s[q]\, \forall a \in A(r) : +\Delta a \in P(1..i)$

To prove $+\Delta q$, this means we need to establish a proof for q using facts and strict rules only. This is a deduction in the classical sense. No proofs for the negation of q need to be considered (in contrast to defeasible provability below, where opposing chains of reasoning must be taken into account, too).

To prove $-\Delta q$, that is, that q is not definitely provable, q must not be a fact. In addition, we need to establish that every strict rule with head q is *known to be* inapplicable.

Thus for every such rule r there must be at least one antecedent a for which we have established that a is not definitely provable ($-\Delta a$).

$-\Delta$: We may append $P(i+1) = -\Delta q$ if
$q \notin F$ and
$\forall r \in R_s[q]\, \exists a \in A(r) : -\Delta a \in P(1..i)$

It is worth noticing that this definition of nonprovability does not involve loop detection. Thus if D consists of the single rule $p \rightarrow p$, we can see that p cannot be proven, but defeasible logic is unable to prove $-\Delta p$.

$+\partial$: We may append $P(i+1) = +\partial q$ if either

(1) $+\Delta q \in P(1..i)$ or
(2) (2.1) $\exists r \in R_{sd}[q]\ \forall a \in A(r)$: $+\partial a \in P(1..i)$ and
(2.2) $-\Delta \sim q \in P(1..i)$ and
(2.3) $\forall s \in R[\sim q]$ either
(2.3.1) $\exists a \in A(s)$: $-\partial a \in P(1..i)$ or
(2.3.2) $\exists t \in R_{sd}[q]$ such that
$\forall a \in A(t)$: $+\partial a \in P(1..i)$ and $t > s$

Let us illustrate this definition. To show that q is defeasibly provable, we have two choices: (1) we show that q is already definitely provable; or (2) we need to argue using the defeasible part of D as well. In particular, we require that there must be a strict or defeasible rule with head q which can be applied (2.1). But now we need to consider possible attacks, that is, reasoning chains in support of $\sim q$. To be more specific: to prove q defeasibly we must show that $\sim q$ is not definitely provable (2.2). Also (2.3) we must consider the set of all rules (including defeaters) which are not known to be inapplicable and which have head $\sim q$. Essentially each such rule s attacks the conclusion q. For q to be provable, each such rule must be counterattacked by a rule t with head q with the following properties: (i) t must be applicable at this point, and (ii) t must be stronger than s. Thus each attack on the conclusion q must be counterattacked by a stronger rule.

The definition of the proof theory of defeasible logic is completed by the condition $-\partial$. It is nothing more than a strong negation of the condition $+\partial$.

$-\partial$: We may append $P(i+1) = -\partial q$ if

(1) $-\Delta q \in P(1..i)$ and
(2) (2.1) $\forall r \in R_{sd}[q]\ \exists a \in A(r)$: $-\partial a \in P(1..i)$ or
(2.2) $+\Delta \sim q \in P(1..i)$ or
(2.3) $\exists s \in$ R[$\sim q$] such that
(2.3.1) $\forall a \in A(s)$: $+\partial a \in P(1..i)$ and
(2.3.2) $\forall t \in R_{sd}[q]$
either $\exists a \in A(t)$: $-\partial a \in P(1..i)$ or $t \not> s$

To prove that q is not defeasibly provable, we must first establish that it is not definitely provable. Then we must establish that it cannot be proven using the defeasible part of the theory. There are three possibilities to achieve this: either we have established that none of the (strict and defeasible) rules with head q can be applied (2.1); or $\sim q$ is definitely provable (2.2); or there must be an applicable rule r with head $\sim q$ such that no possibly applicable rule s with head $\sim q$ is superior to s (2.3).

The elements of a derivation P in D are called *lines* of the derivation. We say that a tagged literal L *is provable in* $D = (F, R, >)$, denoted $D \vdash L$, if there is a derivation in D such that L is a line of P. When D is obvious from the context we write $\vdash L$.

Consider the theory in Example 1 again. Then the conclusions that can be drawn from this theory are: $-\Delta a$, $-\Delta\neg a$, $+\partial a$, $-\partial\neg a$, $-\Delta b$, $-\Delta\neg b$, $+\partial b$, $-\partial\neg b$, $-\Delta c$, $-\Delta\neg c$, $-\partial c$, $-\partial\neg c$, $+\Delta d$, $+\partial d$, $-\Delta\neg d$, $-\partial\neg d$.

Two defeasible theories D and D' are *equivalent*, denoted $D \equiv D'$, if for each tagged literal L, it holds that $D \vdash L$ iff $D' \vdash L$.

3 Forgetting in Defeasible Logic: A Naive Approach

Intuitively, given a knowledge base K and a set of atoms A, which we wish to forget from K, we are interested in obtaining a knowledge base $K' = forget(K, A)$ such that: (a) K' does not contain any occurrence of any atom in A, and (b) K' is equivalent to K for all atoms not belonging to A.

In the context of defeasible logic, this intuitive idea translates to the following task:

> Given a defeasible theory D and a set of atoms A, find a defeasible theory D' such that (a) D' does not contain any occurrences of any atom in A, and (b) $D' \vdash q \Leftrightarrow D \vdash q$ for any tagged literal q that does not contain any atom in A.

We denote such a theory D' by $forget(D, A)$; for singleton $A = \{a\}$, we also write $forget(D, a)$. Note that, by definition, any two different defeasible theories D' and D'' satisfying conditions (a) and (b) above are equivalent. Thus $forget(D, A)$ is uniquely defined, modulo equivalence of defeasible theories.

A first, naive approach to solving this problem is applying the following algorithm.

Algorithm 1

1. Compute the set $cons(D)$ of all conclusions of D.
2. Delete all conclusions from $cons(D)$ that contain an atom in A.
3. Construct a defeasible theory D' that has exactly those conclusions contained in the result of step 2. □

Before continuing, let us make an initial remark about this approach: While quite naive, it is computationally feasible, in contrast to, say, propositional logic or description logics where forgetting has already been studied; the reason is that the conclusions of a defeasible theory (step 1) can be computed in linear time, as shown in [20]. Indeed, the algorithm provided in that work constitutes the first step in our algorithm.

Thus, the only question that remains is whether step 3 can be carried out. To provide an answer to this question, we have to carefully analyze the proof theory of defeasible logic. This analysis provides a number of relationships among different conclusions; for example, if $D \vdash +\Delta p$ then also $D \vdash +\partial p$. And also $D \nvdash +\partial p$ or $D \nvdash -\partial p$ (or both). In addition, there are interrelationships between possible conclusions involving an atom p and its negation. For example, if $D \vdash +\Delta\neg p$ and $D \vdash -\Delta p$ then $D \vdash -\partial p$.

Maher et al. [22] provide a thorough analysis of the proof theory of defeasible logic, and reveal that regarding the provability of a set of conclusions involving p and its

Table 1. Distinct cases of provable sets of conclusions involving p and $\neg p$ (cf. [22])

Case	Set of conclusions	Rules with this outcome
1		$p \to p, \neg p \to \neg p$
2	$+\Delta\neg p, +\partial\neg p$	$p \to p, \to \neg p$
3	$-\Delta\neg p$	$p \to p, \neg p \Rightarrow \neg p$
4	$-\Delta\neg p, -\partial\neg p$	$p \to p$
5	$+\partial p, -\Delta\neg p, -\partial\neg p$	$\Rightarrow p, p \to p$
6	$+\Delta p, +\partial p, +\Delta\neg p, +\partial\neg p$	$\to p, \to \neg p$
7	$+\Delta p, +\partial p, -\Delta\neg p, -\partial\neg p$	$\to p$
8	$-\Delta p, +\partial p, -\Delta\neg p, -\partial\neg p$	$\Rightarrow p$
9	$-\Delta p, -\Delta\neg p$	$p \Rightarrow p, \neg p \Rightarrow \neg p$
10	$-\Delta p, -\Delta\neg p, -\partial\neg p$	$p \Rightarrow p$
11	$-\Delta p, -\partial p, -\Delta\neg p, -\partial\neg p$	

negation, there are essentially eleven distinct cases, out of $2^8 = 256$ syntactically possible sets of conclusions, which are summarized in Table 1. For each of these cases, we provide in the rightmost column a respective defeasible theory that contains only the literals p and $\neg p$. Thus we demonstrate how step 3 of Algorithm 1 can be carried out.

Let us explain a few of these cases. Case 1 is the one where no conclusion regarding p or $\neg p$ can be drawn. This outcome is achieved by strict loops for both p and $\neg p$, which are not detected by the proof theory of Section 2, therefore no conclusion can be drawn.

Case 2 represents that $D \not\vdash -\Delta p$ and $D \not\vdash +\partial p$ while $D \vdash +\Delta\neg p$ and $D \vdash +\partial\neg p$. The theory for it has again a loop for p, but $\neg p$ is definitely, thus also defeasibly, provable.

The theory for case 3 consists of the rules $\{p \to p, \neg p \Rightarrow \neg p\}$. Here there is a strict loop for p, but a defeasible loop for $\neg p$. The latter allows one to derive at least the definite non-provability of $\neg p$ (as there is no fact nor a strict rule with head $\neg p$).

Case 6 is the one of inconsistency, which may only occur if both p and $\neg p$ are definitely (hence also defeasibly) provable, as achieved by the theory provided. Finally, in case 10 absence of facts or strict rules achieves definite non-provability. The loop on p prevents us from deriving $-\partial p$, whereas absence of a rule with head $\neg p$ gives us $-\partial\neg p$.

Summarizing the discussion so far, all three steps of the algorithm provided in this section are feasible, and provide a solution to the problem of forgetting.

Proposition 1. *Given a defeasible theory D and a set of atoms A, Algorithm 1 computes a defeasible theory $D' = forget(D, A)$.*

Moreover, the algorithm can be evaluated efficiently: given Table 1, step 1 is the only involved step which however can be done in linear time [20].

Proposition 2. *Algorithm 1 computes $forget(D, A)$ in linear time.*

Though these theoretical results are positive, the algorithm is still naive, in the sense that it completely abandons the original structure of the knowledge base and returns an artificial theory.

Example 2. Take the theory in Example 1 and now suppose we wish to forget about b. Then Algorithm 1 returns $forget(D, A) = (F', R', >')$ where

$$R' = \{ \rightarrow d \\ \Rightarrow a \}$$

and F' and $>'$ are empty.

This approach is clearly not suitable for practical purposes. The question arises whether the theoretical properties can be achieved while maintaining the original knowledge structure to the extent possible. This question is studied in the next section.

4 Forgetting in Defeasible Logic: An Improved Approach

The idea for the improved algorithm is the following: assuming that we have the complete picture regarding the derivability of p and $\neg p$ (one of the eleven cases of the previous section), we transform the set of rules in a way that takes into account the derivability of p and $\neg p$. For example, consider the rules

$$r_1 : p, s \rightarrow \neg a$$
$$r_2 : p, q \Rightarrow a$$

and the derivability $D \vdash -\Delta p$ and $D \vdash +\partial p$. Rule r_1 cannot fire as we have established that p is not definitely derivable. And we can delete p from the body of r_2 as p is defeasibly derivable. As a result of this transformation, the resulting set of rules contains no occurrence of p, and represents the result of forgetting p. Let us now consider another interesting case. Consider the rule

$$r : p \Rightarrow s$$

and the derivability $D \vdash -\Delta p$ and $D \nvdash +\partial p$ and $D \nvdash -\partial p$. Clearly rule r cannot fire ever, as p cannot be derived. However, $-\partial p$ cannot be derived either (indicating a cyclic situation; see e.g. case 10 in Table 1). Suppose we have in our theory another rule

$$r' :\Rightarrow \neg s.$$

Rule r' does not fire to prove $\neg s$ because it is attacked by r and r cannot be discarded as we cannot prove non-derivability of its antecedent p. Simply deleting r when forgetting p would delete this attack on r', thus enabling derivability of $\neg s$ and altering the set of conclusions not involving p. The solution to this problem is to turn r into a defeater

$$r :\rightsquigarrow s.$$

This rule fails to support derivation of s but is able to attack r' and prevents it from firing.

As a final example, consider the rule

$$r : p \rightarrow q$$

and suppose $D \vdash +\partial p$ and $D \vdash -\Delta p$. In this case, the strict rule r fails to prove q strictly, but is able to prove q defeasibly. If we were to simply remove r in the process of forgetting p, we would lose this defeasible provability of q. To avoid this problem, we replace r by the defeasible rule

$$r :\Rightarrow q$$

Based on these ideas, we provide in the following the full transformation in all eleven cases of derivability of p and $\neg p$.

Case 1: $\{\ \}$

- Replace all rules $r : A \hookrightarrow q$ where p or $\neg p$ appear in A with $r : A \setminus \{p, \neg p\} \rightsquigarrow q$.
- Delete all facts and rules with p or $\neg p$ in their head.

Case 2: $\{+\Delta\neg p, +\partial\neg p\}$

- Replace all rules $r : A \hookrightarrow q$ such that p appears in A with $r : A \setminus \{p\} \rightsquigarrow q$.
- Remove all occurrences of $\neg p$ from all rule bodies.
- Delete all facts and rules with p or $\neg p$ in their head.

Case 3: $\{-\Delta\neg p\}$

- Replace all rules $r : A \hookrightarrow q$ such that p appears in A with $r : A \setminus \{p\} \rightsquigarrow q$.
- Replace all rules $r : A \hookrightarrow q$ such that $\neg p$ appears in A with $r : A \setminus \{\neg p\} \rightsquigarrow q$.
- Delete all facts and rules with p or $\neg p$ in their head.

Case 4: $\{-\Delta\neg p, -\partial\neg p\}$

- Replace all rules $r : A \hookrightarrow q$ such that p appears in A with $r : A \setminus \{p\} \rightsquigarrow q$.
- Remove all rules with $\neg p$ as one of its antecedents.
- Delete all facts and rules with p or $\neg p$ in their head.

Case 5: $\{+\partial p, -\Delta\neg p, -\partial\neg p\}$

- Replace all strict rules $r : A \rightarrow q$ with p in A by the defeasible rule $r : A\setminus\{p\} \Rightarrow q$.
- Remove all occurrences of p in the bodies of defeasible rules and defeaters.
- Remove all rules with $\neg p$ as one of its antecedents.
- Delete all facts and rules with p or $\neg p$ in their head.

Case 6: $\{+\Delta p, +\partial p, +\Delta\neg p, +\partial\neg p\}$

- Remove all occurrences of p and $\neg p$ from all rule bodies.
- Delete all facts and rules with p or $\neg p$ in their head.

Case 7: $\{+\Delta p, +\partial p, -\Delta\neg p, -\partial\neg p\}$

- Remove all occurrences of p from the bodies of all rules.
- Remove all rules with $\neg p$ as one of its antecedents.
- Delete all facts and rules with p or $\neg p$ in their head.

Case 8: $\{-\Delta p, +\partial p, -\Delta\neg p, -\partial\neg p\}$

- Replace all strict rules $r : A \to q$ with p in A by the defeasible rule $r : A\setminus\{p\} \Rightarrow q$.
- Remove all occurrences of p from the bodies of all defeasible rules and defeaters.
- Remove all rules with $\neg p$ as one of its antecedents.
- Delete all facts and rules with p or $\neg p$ in their head.

Case 9: $\{-\Delta p, -\Delta\neg p\}$

- Replace all rules $r : A \hookrightarrow q$ where p or $\neg p$ appear in A with $r : A \setminus \{p, \neg p\} \rightsquigarrow q$.
- Delete all facts and rules with p or $\neg p$ in their head.

Case 10: $\{-\Delta p, -\Delta\neg p, -\partial\neg p\}$

- Replace all rules $r : A \hookrightarrow q$ where p appears in A with $r : A \setminus \{p\} \rightsquigarrow q$.
- Remove all rules with $\neg p$ as one of its antecedents.
- Delete all facts and rules with p or $\neg p$ in their head.

Case 11: $\{-\Delta p, -\partial p, -\Delta\neg p, -\partial\neg p\}$

- Remove all rules in which p or $\neg p$ occurs.

This leads us then to the following improved algorithm for forgetting.

Algorithm 2

1. Compute all conclusions concerning atoms in A.
2. For each atom p in A, transform the rules in D to obtain D'.
3. Delete all priority pairs $r > s$ where r or s was deleted in step 2. □

Step 1 can be carried out, in the worst case, by computing all conclusions of D. Step 2 is based on the analysis provided above.

Example 3. Reconsider the theory D in Example 1. Note that $-\Delta b$, $+\partial b$, $-\Delta\neg b$, and $-\partial\neg b\}$ are provable from D. So, by Case 8 above, application of Algorithm 2 yields the theory $D' = (\emptyset, R', >')$ as $forget(D, b)$, where

$$R' = \{\ r_0 :\to d \quad r_1 :\Rightarrow a \quad r_2 :\Rightarrow \neg a \quad r_5 : d \Rightarrow c \quad r_6 :\Rightarrow \neg c\ \}$$

and $r_1 >' r_2$.

Example 4. Consider the defeasible theory $D = (F, R, >)$ where

$$R = \{\ r_1 : a \Rightarrow a \quad r_2 : a \Rightarrow b \quad r_3 :\Rightarrow \neg b\ \}$$

and F and $>$ are empty. Here $\neg b$ is not defeasibly derivable because a cannot be shown to be non-provable ($-\partial a$ cannot be derived). Thus it would be a mistake to simply delete r_2 when forgetting a. Instead, Algorithm 2 correctly turns a modified rule r_2 (without a in the body) into a defeater, giving as result the defeasible theory $D' = (\emptyset, R', \emptyset)$ where R' contains the rules

$r_2 :\rightsquigarrow b$
$r_3 :\Rightarrow \neg b$.

The following proposition states that Algorithm 2 indeed provided a correct solution for forgetting.

Proposition 3. *Given a defeasible theory D and a set of atoms A, Algorithm 2 computes a defeasible theory $D' = forget(D, A)$.*

Proof. (Sketch) We want to show that for each tagged literal L whose literal is neither p nor $\neg p$, $D \vdash L$ iff $D' \vdash L$.

If $D \vdash L$, then there exists a derivation of L in D: $P = (P(1), ..., P(n))$. We use P' to denote the sequence of tagged literals obtained from P by removing every tagged literal whose literal is either p or $\neg p$. Then we can prove that P' is a derivation of L in D' by induction on the length n of P. The induction step can be done by examining the Cases 1-11 in Algorithm 2.

On the other hand, if $P' = (P'(1), ..., P'(m))$ is a derivation of L in D', we can construct a derivation P of L in D inductively as follows.

Assume that a derivation $(P(1), ..., P(u))$ of $P'(m-1)$ in D has been constructed. Again, the induction step can be done by examining the Cases 1-11 in Algorithm 2. For instance, here we consider the Case 7 in Algorithm 2: $D \vdash \{+\Delta p, +\partial p, -\Delta\neg p, -\partial\neg p\}$.

To construct a derivation of $P'(m) = L$ in D, consider four possible cases:
Case 1. $L = +\Delta q$: Then either $q \in F$ or $\exists r' \in R'_s[q]\ \forall a \in A(r') :\ +\Delta a \in P'(1..m-1)$. If $q \in F$, then it is done. If $q \notin F$, by the (last) induction assumption, $\forall a \in A(r') : +\Delta a \in P(1..u)$. If $r' \in R$, then $(P(1), ..., P(u), L)$ is already a derivation of L in D. So we assume that $r' \notin R$. By the Case 7 in Algorithm 2, there exists $r \in R$ such that $A(r) = A(r') \cup \{p\}$. Let $P'' = (P''(1), ..., P''(v))$ be a derivation of $+\Delta p$; then $P = (P(1), ..., P(u), P''(1), ..., P''(v), L)$ is a derivation of L in D.
Cases 2,3,4: $L = -\Delta q$, $L = +\partial q$, $L = -\partial q$: we can similarly construct a derivation of L in D. □

4.1 Semantic Properties

Algorithm 2 satisfies a number of desirable properties. One is that forgetting atoms which do not occur in the theory cause no change.

Proposition 4. *Let D be a defeasible theory and let A be a set of atoms. Then forget $(D, A) \equiv forget(D, A')$ where $A' \subseteq A$ is the set of atoms from A that occur in D; in particular $forget(D, \emptyset) = D$.*

Another such property is irrelevance of syntax with respect to equivalence.

Proposition 5. *Let D and D' be defeasible theories such that $D \equiv D'$. Then, for every set A of atoms, $forget(D, A) \equiv forget(D', A)$.*

Proof. Let D and D' be two defeasible theories that are equivalent. Then, for each tagged literal L whose literal is not in A,

$forget(D, A) \vdash L$
iff $D \vdash L$
iff $D' \vdash L$
iff $forget(D', A) \vdash L$.

Thus, $forget(D, A)$ and $forget(D', A)$ are also equivalent. □

In addition, the result is independent of whether atoms are forgotten successively (in some order), or altogether.

Proposition 6. *Let D be a defeasible theory and let $A = \{a_1, \ldots, a_n, a_{n+1}\}$ be a set of atoms. Then $forget(D, A) \equiv forget(forget(D, \{a_1, \ldots, a_n\}), a_{n+1})$.*

Proof. (Sketch) This can be shown by a simple induction on the size n of A. Let $A_n = \{a_1, \ldots, a_n\}$. We need only to observe that

$$forget(D, A) \equiv forget(D, A_n) \equiv forget(forget(D, A_n), a_{n+1}).$$ □

Note that Algorithm 2 is as little disruptive to the original theory as possible. A close inspection reveals that its only operations are to:

- remove all rules containing atoms in A in their heads;
- remove provable atoms from A from rule bodies;
- remove rules with an atom from A in their body, in cases where this atom is not provable;
- turn rules into defeaters in certain cases caused by cyclicity (see Example 2);
- remove priority pairs where one of the rules involved was deleted.

All these changes are necessary and as little obstructive as possible in the attempt to compile the derivability of literals from A into the knowledge base. The following formal property seeks to partially capture this intuitive notion of "structure preservation". It makes use of direct dependency. Formally, an atom a is *directly dependent* on an atom b iff either a and b are identical, or there is a rule r with head a or $\neg a$, such that b or $\neg b$ appears in the body of r. An atom a is *dependent* on an atom b iff there is a sequence $a_1, a_2, \ldots, a_t$ of atoms such that $t > 0$ and a_i is directly dependent on a_{i+1} for $i = 1, \ldots, t-1$.

Proposition 7. *$forget(D, A)$ differs from D only on rules whose head is directly dependent on an atom in A.*

4.2 Complexity

Like Algorithm 1, also Algorithm 2 can be run efficiently, as all steps 1-3 are feasible in linear time (for steps 2 and 3, suitable standard data structures are used).

Proposition 8. *Algorithm 2 computes $forget(D, A)$ in linear time.*

Finally, the size of the outcome remains linear, ensuring (in conjunction with Proposition 5) that the result of iterated forgetting is of linear size. The latter does not necessarily follow from the linear time complexity, as there might be an exponential increase in the number of iterations.

Corollary 1 (of Propositions 4, 6 and 8). *For every defeasible theory D and sets of atoms $A_1, \ldots, A_m$, The size of $D' = forget(forget(\cdots forget(D, A_1), \cdots), A_m)$ computed by Algorithm 2 is linear in the size of D, and D' is computed in time linear in the size of D and $A_1, \ldots, A_m$.*

Proof. (Sketch) By Proposition 4, without loss of generality the sets A_i are pairwise disjoint and nonempty. For each A_i, we can write the forgetting of $A_i = \{a_{i,1}, \ldots, a_{i,n_i}\}$ as iterated forgetting of each $a_{i,j}$, $1 \leq j \leq n_i$. Thus, we obtain that

$$D' = forget(forget(\cdots forget(D, a_{1,1}), \cdots), a_{m,n_m}).$$

By Proposition 6 again, $D' = forget(D, A)$ where $A = \bigcup_{i=1}^{m} A_i$. The result follows then from Proposition 8. □

4.3 Modularity

Let us, given any two defeasible theories $D_1 = (F_1, R_1, <_1)$ and $D_2 = (F_2, R_2, <_2)$, define their union $D_1 \cup D_2$ to be the defeasible theory $(F_1 \cup F_2, R_1 \cup R_2, <_1 \cup <_2)$.

Given a theory D and a set A of atoms, it is often the case that D is large but only a small fraction of D is relevant to A. That is, D can be split into two parts D_1 and D_2 where D_2 is irrelevant to A. In this case, to forget about A in D, one would expect to perform forgetting only on D_1. Unfortunately, this is not true in general. Consider the following theory $D = (F, R, >)$, where

$$\begin{aligned} R = \{\ & r_1 : a' \Rightarrow a \\ & r_2 : a \Rightarrow b \\ & r_3 : b \Rightarrow c \\ & r_4 : \ \rightarrow a' \ \} \end{aligned}$$

and F and $>$ are empty. Then $D = D_1 \cup D_2$ where $D_1 = (\emptyset, \{r_1, r_2, r_3\}, \emptyset)$ and $D_2 = (\emptyset, \{r_4\}, \emptyset)$. Then for $A = \{b\}$, $forget(D, A)$ is not equivalent to $forget(D_1, A) \cup D_2$: We note that $+\partial b$ is provable from D and the body of r_3 is b. So, if $A = \{b\}$ is forgotten from D, $+\partial c$ remains provable from $forget(D, A)$.

On the other hand, $+\partial b$ is not provable in D and thus when forgetting about $A = \{b\}$, the rule r_3 is deleted. As a result, even when $r_5 : a'$ is added, $+\partial c$ is still not provable from $forget(D_1, A) \cup D_2$.

We see that the solution provided by Algorithm 2 is not modular. But at least the following restricted form of splitting for forgetting holds, which can be seen from the correctness of Algorithm 2.

Proposition 9. *Let A be a set of atoms and let $D = D_1 \cup D_2$ be a defeasible theory without defeaters such that (1) no atom in D_1 depends on an atom in D_2, and (2) no atoms in A appear in D_2. Then $forget(D, A)$ is equivalent to $forget(D_1, A) \cup D_2$.*

Proof. (Sketch) Without loss of generality, we assume that $A = \{p\}$. Given a tagged literal L whose literal is q instead of p, we consider two possible cases.
Case 1. q appears only in D_1: Then

$D \vdash L$
iff $D_1 \vdash L$
iff $forget(D_1, A) \vdash L$
iff $forget(D_1, A) \cup D_2 \vdash L$.

Case 2. q appears in D_2: It can also be shown that $D \vdash L$ iff $forget(D_1, A) \cup D_2 \vdash L$. The basic idea is that, given a tagged literal L' whose literal is in D_1, we can always replace the derivation of a tagged literal L' in D_1 with a derivation of L' in $forget(D_1, A)$ or vice versa. □

As a final point, we wish to place this partial modularity result into context. There is an intuitive trade-off between dependency/derivability of literals and modularity. Intuitively, if we want to achieve unrestricted modularity, then information about the connection (derivability) between pairs of literals must be retained in the theory; but to record such information, without additional symbols, requires quadratic space. Let us consider as an example the following set of rules:

$a_i \Rightarrow b$ with $i \in \{1, \ldots, n\}$,
$b \Rightarrow c_j$ with $j \in \{1, \ldots, n\}$.

Here each c_j depends on all a_i's, and we have quadratically many "implied rules" $a_i \Rightarrow c_j$ to incorporate when forgetting about b, which cannot be done in linear time. And in lack of any further information to discriminate among different rules, it would also be unclear which of these implied rules to add, so adding none would make sense; this is exactly what Algorithm 2 does. So we would argue that Algorithm 2 is a reasonable core for linear-time forgetting algorithms in the context of defeasible logic. Adding all, or some (according to some external criteria) "implied rules" would give refinements of this basic algorithm.

5 Conclusion

In this paper we studied the problem of forgetting in the context of defeasible logic. We provided two algorithms for computing the result of forgetting a set of literals, a naive algorithm and an advanced one (Algorithm 2) whose output has several desired properties. Salient features of the solutions provided are linear time complexity, and the linear size of iterated forgetting. In addition, Algorithm 2 preserves a lot of structure of the original defeasible theory; to measure how much in formal terms (e.g., using similarly via tree edit distance) remains to be considered.

We intend to continue work on forgetting. One task is to determine a modular approach to forgetting: Algorithm 2 is not modular, as it has to recompute all conclusions related to p and $\neg p$ every time the theory is changed. An interesting question would be to define a pure transformation approach, in the spirit of step 2 of Algorithm 2, without

the need for step 1. Many research questions follow on from this, including a thorough analysis of time complexity, space of the outcome, and tradeoff between modularity and dependency and derivability, continuing the discussion at the end of Section 4.

Another idea for future work is to apply forgetting to multi-context theories, and in particular to the work of [5] which is essentially a contextual defeasible logic. Finally, we might apply forgetting in the context of the Semantic Web, in particular to defeasible rule systems over RDF [1,4], as a means of information integration and knowledge dynamics, and to deal with inconsistencies in this context.

Acknowledgments. The authors would like to thank the three anonymous referees for their helpful comments. This work was partially supported by the Australia Research Council (ARC) Discovery Projects DP1093652 and DP110101042, by an Olga Taussky Fellowship of the Wolfgang Pauli Institute (WPI) Vienna, by the Austrian Science Fund (FWF) project P20841 and by the Vienna Science and Technology Fund (WWTF) grant ICT 08-020.

References

1. Antoniou, G., Bikakis, A.: DR-Prolog: A system for defeasible reasoning with rules and ontologies on the semantic web. IEEE Transactions on Knowledge and Data Engineering 19(2), 233–245 (2007)
2. Antoniou, G., Billington, D., Governatori, G., Maher, M.: On the modeling and analysis of regulations. In: Proc. Australian Conference Information Systems, pp. 20–29 (1999)
3. Antoniou, G., Billington, D., Governatori, G., Maher, M.: Representation results for defeasible logic. ACM Transactions on Computational Logic 2(2), 255–287 (2001)
4. Bassiliades, N., Antoniou, G., Vlahavas, I.: A defeasible logic reasoner for the semantic web. International Journal of Semantic Web Information Systems 2(1), 1–41 (2006)
5. Bikakis, A., Antoniou, G.: Defeasible contextual reasoning with arguments in ambient intelligence. IEEE Transactions on Knowledge and Data Engineering 22(11), 1492–1506 (2010)
6. Billington, D., Antoniou, G., Governatori, G., Maher, M.: An inclusion theorem for defeasible logics. ACM Transactions on Computational Logic 12(1), 6 (2010)
7. Eiter, T., Wang, K.: Semantic forgetting in answer set programming. Artificial Intelligence 14, 1644–1672 (2008)
8. Governatori, G.: Representing business contracts in RuleML. International Journal of Cooperative Information Systems 14(2-3), 181–216 (2005)
9. Governatori, G., Dumas, M., ter Hofstede, A., Oaks, P.: A formal approach to legal negotiation. In: International Conference on Artificial Intelligence and Law, pp. 168–177 (2001)
10. Governatori, G., Maher, M., Antoniou, G., Billington, D.: Argumentation semantics for defeasible logic. Journal of Logic Computation 14(5), 675–702 (2004)
11. Governatori, G., Padmanabhan, V., Sattar, A.: A Defeasible Logic of Policy-Based Intention (Extended Abstract). In: McKay, B., Slaney, J.K. (eds.) Canadian AI 2002. LNCS (LNAI), vol. 2557, p. 723. Springer, Heidelberg (2002)
12. Governatori, G., Rotolo, A.: Defeasible Logic: Agency, Intention and Obligation. In: Lomuscio, A., Nute, D. (eds.) DEON 2004. LNCS (LNAI), vol. 3065, pp. 114–128. Springer, Heidelberg (2004)
13. Governatori, G., Rotolo, A., Sartor, G.: Temporalised normative positions in defeasible logic. In: Proc. 10th International Conference on Artificial Intelligence and Law, pp. 25–34 (2005)

14. Herzig, A., Mengin, J.: Uniform Interpolation by Resolution in Modal Logic. In: Hölldobler, S., Lutz, C., Wansing, H. (eds.) JELIA 2008. LNCS (LNAI), vol. 5293, pp. 219–231. Springer, Heidelberg (2008)
15. Konev, B., Walther, D., Wolter, F.: Forgetting and uniform interpolation in large-scale description logic terminologies. In: Proc. 20th International Joint Conference on Artificial Intelligence (IJCAI 2009), pp. 830–835 (2009)
16. Kontchakov, R., Wolter, F., Zakharyaschev, M.: Can you tell the difference between DL-Lite ontologies? In: Proc. 11th International Conference on Principles of Knowledge Representation and Reasoning (KR 2008), pp. 285–295 (2008)
17. Lang, J., Liberatore, P., Marquis, P.: Propositional independence: Formula-variable independence and forgetting. Journal of Artificial Intelligence Research 18, 391–443 (2003)
18. Lang, J., Marquis, P.: Resolving inconsistencies by variable forgetting. In: Proc. 8th International Conference on Principles of Knowledge Representation and Reasoning (KR 2002), pp. 239–250 (2002)
19. Lin, F., Reiter, R.: Forget it. In: Proc. AAAI Fall Symposium on Relevance, New Orleans (LA), pp. 154–159 (1994)
20. Maher, M.: Propositional defeasible logic has linear complexity. Theory and Practice of Logic Programming 1(6), 691–711 (2001)
21. Maher, M.: A model-theoretic semantics for defeasible logic. In: Proc. Workshop on Paraconsistent Computational Logic, pp. 67–80 (2002)
22. Maher, M., Antoniou, G., Billington, D.: A Study of Provability in Defeasible Logic. In: Antoniou, G., Slaney, J.K. (eds.) Canadian AI 1998. LNCS, vol. 1502, pp. 215–226. Springer, Heidelberg (1998)
23. Nute, D.: Defeasible logic. In: Handbook of Logic in Artificial Intelligence and Logic Programming, vol. 3. Oxford University Press (1994)
24. Skylogiannis, T., Antoniou, G., Bassiliades, N., Governatori, G.: Dr-negotiate - a system for automated agent negotiation with defeasible logic-based strategies. In: Proc. 2005 IEEE International Conference on e-Technology, e-Commerce and e-Service (EEE 2005) on e-Technology, e-Commerce and e-Service, pp. 44–49 (2005)
25. van Ditmarsch, H.P., Herzig, A., Lang, J., Marquis, P.: Introspective Forgetting. In: Wobcke, W., Zhang, M. (eds.) AI 2008. LNCS (LNAI), vol. 5360, pp. 18–29. Springer, Heidelberg (2008)
26. Wang, K., Sattar, A., Su, K.: A theory of forgetting in logic programming. In: Proc. 20th National Conference on Artificial Intelligence (AAAI 2005), pp. 682–687 (2005)
27. Wang, K., Wang, Z., Topor, R., Pan, J.Z., Antoniou, G.: Concept and Role Forgetting in $\mathcal{ALC}$ Ontologies. In: Bernstein, A., Karger, D.R., Heath, T., Feigenbaum, L., Maynard, D., Motta, E., Thirunarayan, K. (eds.) ISWC 2009. LNCS, vol. 5823, pp. 666–681. Springer, Heidelberg (2009)
28. Wang, Z., Wang, K., Topor, R., Pan, J.Z.: Forgetting Concepts in DL-Lite. In: Bechhofer, S., Hauswirth, M., Hoffmann, J., Koubarakis, M. (eds.) ESWC 2008. LNCS, vol. 5021, pp. 245–257. Springer, Heidelberg (2008)
29. Zhang, Y., Foo, N., Wang, K.: Solving logic program conflicts through strong and weak forgettings. In: Proc. 19th International Joint Conference on Artificial Intelligence (IJCAI 2005), pp. 627–632 (2005)
30. Zhang, Y., Zhou, Y.: Knowledge forgetting: Properties and applications. Artificial Intelligence 173(16-17), 1525–1537 (2009)
31. Zhou, Y., Zhang, Y.: Bounded forgetting. In: Proc. 25th AAAI Conference on Artificial Intelligence, pp. 280–285 (2011)

Querying Proofs

David Aspinall[1,⋆], Ewen Denney[2,⋆⋆], and Christoph Lüth[3,⋆⋆⋆]

[1] LFCS, School of Informatics, University of Edinburgh,
Edinburgh EH8 9AB, Scotland
[2] SGT, NASA Ames Research Center
Moffett Field, CA 94035, USA
[3] Deutsches Forschungszentrum für Künstliche Intelligenz
Bremen, Germany

This work is dedicated fondly to the memory of Kostas Tourlas, one of the originators of hiproofs.

Abstract. We motivate and introduce a query language *PrQL* designed for inspecting machine representations of proofs. PrQL natively supports *hiproofs* which express proof structure using hierarchical nested labelled trees. The core language presented in this paper is *locally structured*, with queries built using recursion and patterns over proof structure and rule names. We define the syntax and semantics of locally structured queries, demonstrate their power, and sketch some implementation experiments.

1 Introduction

Automated proof tools and interactive theorem provers are increasingly required to produce evidence of their claims as formal proof objects that may be independently checked or, perhaps, imported into other systems or transformed in particular ways. Proofs connect together atomic rules of inference and axioms in a sound way according to an underlying logic. Checking that this has been done correctly is essentially straightforward, although producing a proof in the first place may be extraordinarily difficult.

Real proofs can be very large, perhaps consisting of tens or hundreds of thousands of atomic rules of inference. There are many things that are interesting to know about such objects, beyond the basic fact that they are correctly constructed. For example, some natural questions when *inspecting* a proof are:

- What is the high-level structure of this proof, (how) can we break it down into pieces to understand it?
- Given a proof of a property which exploits a set of domain-specific axioms, which axioms actually occurred in the proof?
- Given a problem statement which contains some existential propositions as sub-formulae, which, if any, witnesses were found to make them true?

⋆ Research supported by EPSRC grant EP/J001058/1.
⋆⋆ Research supported by NASA contract NNA10DE83C.
⋆⋆⋆ Research supported by BMBF grants 01IS09044B (IGEL) and 01IW10002 (SHIP).

N. Bjørner and A. Voronkov (Eds.): LPAR-18, LNCS 7180, pp. 92–106, 2012.

– Does a large proof contain duplicated parts that could be abstracted into a lemma, to reduce the size of the proof?

When the user is trying to understand the proof construction process, there are natural questions which relate the constructed proof back to the procedures that produced it. If tactics are our notion of proof producing procedure, some questions *relating* the proof to the tactics that produced it are:

– Given a set of tactics and a proof, which tactics were invoked in producing the proof and what subgoals did they solve?
– Were any tactics used recursively?
– Does one particular tactic always lead to another being invoked?
– Did some tactics get invoked but do no useful work?

These sort of questions are not idle curiosities: they are useful for practical *proof engineering*, when managing and maintaining sets of properties, proofs and programs which create and check them. One of us (Denney) routinely resorts to low-level scripted tools to perform these kind of examinations when building large safety cases supported by formal proofs.

We consider querying proofs here in a rigorous, generic manner with the hope of enabling general tools with clear foundations. In this paper, we introduce the basis of a query language *PrQL* designed specifically for querying proofs.

Hierarchical Structured Proofs. The foundation we start from is *hiproofs* [1,2], which provide a simple abstract notion of proof tree by composing atomic rules of inference from an unspecified underlying logic. Going beyond ordinary trees, they have a notion of *hierarchy*, by allowing labelling and nesting of subtrees. This simple addition provides a precise and useful notion of *structure* in the proof which can be used, for example, for noting where a lemma was applied, or where a particular tactic or external proof tool produced a subtree.

Contributions and Paper Outline. This paper contributes towards generic foundational aspects of theorem proving systems. Query languages for tree and graph structured data have been studied over the last decade or so, but have rarely been applied to formal proofs. We design a new core proof query language from first principles, directly connected with a precise abstract notion of proof. With motivating examples and implementation experiments, we establish its utility.

The rest of this paper is structured as follows. Section 2 introduces the foundation of hiproofs used in the rest of the paper. Section 3 describes the design decisions we took for our query language, and introduces it with a sequence of informal examples and their intended meanings. Section 4 describes the meaning of queries formally, so one can check that example queries indeed have the desired meanings; it also provides a baseline decidability result. In Section 5 we sketch a simple prototype implementation, which we use to validate our language design; full-scale experiments on large proofs remain as future work. We mention some of our future plans and discuss some related work in the concluding Section 6.

2 Hiproofs

Hiproofs add structure to an underlying *derivation system*, a simple form of logical framework. We give a brief recap here, for fuller details please see [1,2].

A hiproof is built from (inverted) atomic inference rules a in the underlying derivation system, to which we give a functional reading: a hiproof maps a finite list of input goals $[\gamma_1, \ldots, \gamma_n]$ to a list of output subgoals $[\gamma'_1, \ldots, \gamma'_m]$. Such a hiproof has the *arity* $n \to m$. A nested hiproof, appearing immediately inside a labelled box, has a single input goal which is the root of the tree at that level.

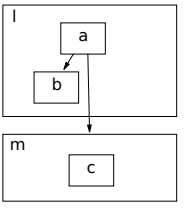
Informally and graphically, we draw hiproofs as inverted trees with a nested structure. Denotationally, a hiproof can be understood as a pair of an ordered tree and a forest with the same set of nodes, subject to some well-formedness conditions. Syntactically, a hiproof can be written as a term s in this grammar:

$$
\begin{array}{lll}
s ::= a & & \text{atomic} \\
\quad \mid \ \mathsf{id} & & \text{identity} \\
\quad \mid \ [l]\, s & & \text{labelling} \\
\quad \mid \ s_1 \ ; \ s_2 & & \text{sequencing} \\
\quad \mid \ s_1 \otimes s_2 & & \text{tensor (juxtaposition)} \\
\quad \mid \ \langle\rangle & & \text{empty}
\end{array}
\tag{1}
$$

Fig. 1 shows an example hiproof term and its graphical representation in the middle. Boxes indicate nestings and have labels in their top corners, indicating the tactic which gave rise to the contents in the box; unlabelled boxes contain atomic rules. Tensor $\otimes$ places hiproofs side-by-side and sequencing ; builds "wiring" to connect hiproofs together, using identity to create wires where a goal is not manipulated. In the example, id exports the second subgoal from the atomic rule a outside the box labelled l. The empty proof $\langle\rangle$ is useful when building proofs programmatically.

Valid Hiproofs. A hiproof is called *valid* if it corresponds to a real proof tree in the underlying derivation system. The hiproof term in Fig. 1 validates the proof tree shown on the right-hand side, where an input goal γ_1 is proved using the atomic inference rules a, b and c. Validity extends naturally to arbitrary hiproof terms that have more than one input goal; such a term corresponds to a finite sequence of proof trees. We write $s \vdash g_1 \longrightarrow g_2$ if s is valid in this more

$([l]\, a \ ; \ b \otimes \mathsf{id}) \ ; \ [m]\, c$



$$\dfrac{\dfrac{}{\gamma_2}\,\mathrm{b} \qquad \dfrac{}{\gamma_3}\,\mathrm{c}}{\gamma_1}\,\mathrm{a}$$

Fig. 1. A hiproof, its graphical representation and a proof it validates

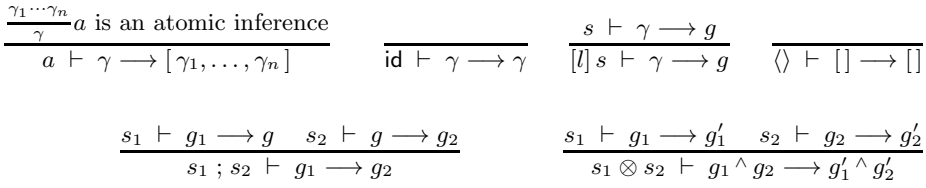


Fig. 2. Validation of hiproofs (the symbol $^\wedge$ stands for list append)

general sense, taking a list of input (proven) goals g_1 to produce a list of output (unsolved) goals g_2. This relation is defined by the rules in Fig. 2.

Validity checking can be seen as a way of adding goals to a hiproof; correspondingly, a valid hiproof can be seen as a nested labelling applied to a flat proof. A hiproof thus represents the outcome of a proof process rather than the method by which it was obtained, and is independent of the direction (forwards from axioms or backwards from conjecture) of construction. In this paper we restrict our attention to valid hiproofs and we assume that the goals are uniquely determined by the validated hiproof.

3 Local Structured Queries

How should we express queries on proofs such as those in Sect. 1? One design choice would be to take an existing query language for graph (or semi-structured) data models (e.g., see surveys [3,4]), and then map from hiproofs into the existing language and use queries there. The drawback with that approach is that we immediately lose connection with our particular source language. Since our initial aim is to understand the concepts and constructs specific to querying proofs, rather than more general objects, we start from queries written in a minimal native query language, and investigate a direct semantics for them.

Our queries follow the hiproof structure, matching on leaves with atomics, structured proofs using labels, or on input or output goals of subproofs. In this paper, we consider queries that specify structure *locally*, in the sense that they cannot directly compare one part of the tree with another, or measure absolute position within the global proof. This restriction arises intentionally, because we use only first-order variables that refer to names and goals, not to subtrees or paths. Despite this, the language is still rather expressive and captures our desired queries fairly succinctly, so it is a good candidate core query language.

To introduce the language, we begin with constructs for matching leaves, boxes and goals in proofs, and then build up following the hiproof syntax.

Matches. We build *matches* inside queries using wildcards and match variables, constants (atoms, sets and predicates) and negation (to construct the complement of a match). Let Var_N be a set of schematic variables standing for names,

ranged over by N in general and A when we suggest an atomic rule name or L a label name. Let Var_G be a set of variables standing for lists of goals. The name matches and goal matches are given by:

$$nm ::= a \mid l \mid * \mid \xi \mid N \mid \neg nm$$
$$gm ::= [\psi_1, \ldots, \psi_n] \mid G \mid \neg gm$$

where ξ stands for a logic-dependent predicate on names, and ψ stands for a logic-dependent predicate on goals used to check some structural property of the goal term. For example we might have a predicate that checks whether a goal γ is in the form of a horn clause, when $\phi_{hornclause}(\gamma)$ holds. Most simply, we suppose that we always have a predicate to check for equality with any specific goal γ and we overload γ to stand for that predicate.

We use matches to build up the *basic queries* that specify local structure. Informally, a basic query may hold for a given hiproof and a substitution of variables the query contains; we will define the result of a query to be the set of variable instantiations that make it true. As (merely) a matter of style, we use a verbose SQL-like textual notation:

$q ::= *$	anything non-empty
$\mid$ **atomic** nm	atomic rule match
$\mid$ **nothing**	nothing (matches only identity)
$\mid$ **inside** nm q	q satisfied inside box with label matching
$\mid$ q_1 **then** q_2	q_1 and q_2 satisfied by successive nodes in ;
$\mid$ q_1 **beside** q_2	q_1 and q_2 satisfied by adjacent nodes in $\otimes$
$\mid$ **ingoals** gm	goals into sub-proof match
$\mid$ **outgoals** gm	goals out of sub-proof match

Basic queries are almost the same language as the hiproof syntax itself, omitting empty proofs and adding the ability to match on goals within. Thus, phrases act as structural patterns matching against an implicit hiproof subject.

For the hiproof given in Fig. 1, the following queries are each satisfied (the alignment around **then** matches the vertical split):

$$\begin{array}{r} (\textbf{inside}\ \mathsf{l}\ *)\ \textbf{then}\ (\textbf{inside}\ \mathsf{m}\ *) \\ (\textbf{inside}\ *\ *\ \textbf{then}\ *\ \textbf{beside}\ \textbf{nothing})\ \textbf{then}\ * \\ (\textbf{inside}\ L_1\ *)\ \textbf{then}\ (\textbf{inside}\ *\ \textbf{atomic}\ A) \end{array}$$

The first two are purely structural, matching the form of the tree. The first matches the outer structure consisting of the box labelled l followed by the box labelled m. The second examines the shape inside the first box. The final query is satisfiable with the unique instantiation $\{L_1 \mapsto \mathsf{l}, A \mapsto \mathsf{c}\}$.

Connectives. We allow propositional logical connectives to build compound queries, with familiar intended meanings:

$$q ::= \ldots \mid q_1 \wedge q_2 \mid q_1 \vee q_2 \mid \neg q$$

Search and Check. Two important quantifier combinators on queries allow us to search within a proof for somewhere that a query is satisfied, or check that a query is satisfied everywhere.

$q ::= \ldots$		
	$\mid$ **somewhere** q	q holds in some subproof
	$\mid$ **everywhere** q	q holds in every subproof

With a syntactic interpretation, the natural domain of quantification is by subterm; because any subterm of a valid hiproof is also valid, this makes sense and we take "subproof" to mean subterm. The scope of **somewhere** and **everywhere** extends as far right as possible. These queries might be added directly to the language, but we will define them instead using recursion (introduced below).

The **somewhere** combinator is used in many of our examples. For example, a proof uses a tactic tac if the query

$$\textbf{somewhere inside } \mathsf{tac} * $$

is satisfied. As another example, we use a match on a goal-list variable G to find the goals passed into a tactic. The query

$$(\textbf{somewhere inside } \mathsf{m} \textbf{ ingoals } G) \vee (\textbf{somewhere atomic } \mathsf{b} \wedge \textbf{ingoals } G)$$

can be read as "tell me the goals that are input to tactic m or the atomic rule b". The result should be the pair of instantiations $\{G \mapsto [\gamma_2]\}, \{G \mapsto [\gamma_3]\}$ for the hiproof in Fig. 1.

When is **everywhere** useful? Not for anything that requires a fixed structure, but with a goal-matching assertion that checks the format of the goals, for example, the check **everywhere outgoals** $[\phi_{hornclause}]$ requires that every goal appearing in the tree must have that certain form. With conditional queries, we use it to specify that goals appearing in certain places must have some property.

Recursive Queries. Just as with tactics we can allow recursively defined queries. Recursively defined queries allow us to build up regular patterns and are defined using query variables Q:

$$q \ ::= \ldots \mid \mu Q.q$$

where q is a query in which Q can appear free. An example recursive pattern is:

$$\mu Q.\, (\textbf{atomic } \mathsf{a} \textbf{ then } (\textbf{ingoals } [\gamma_2] \textbf{ beside } Q)) \vee (\textbf{inside } \mathsf{m} *)$$

which is satisfied by proofs that repeatedly apply the atomic rule a, until reaching a box named m.

Using recursion we can define the searching and checking quantifiers:

$$\textbf{somewhere } q \stackrel{def}{=} \mu Q.\, q \vee \ (\textbf{inside } * \ Q) \vee (Q \textbf{ then } *) \vee (* \textbf{ then } Q) \vee (Q \textbf{ beside } *) \vee (* \textbf{ beside } Q)$$

$$\textbf{everywhere } q \stackrel{def}{=} \mu Q.\, q \wedge (\textbf{atomic } * \ \vee \textbf{ nothing} \vee (\textbf{inside } * \ Q) \vee (Q \textbf{ then } Q) \vee (Q \textbf{ beside } Q))$$

these ensure that q holds at one (or every) node following the structure of the proof; notice that exactly one of the disjuncts must hold in the recursive cases. Later on we will show that these definitions have the intended meaning.

Derived Forms. Using this core, we can readily add more derived forms:

$$
\begin{array}{rcl@{\qquad}rcl}
q_1 \textbf{ when } q_2 & \stackrel{def}{=} & \neg q_2 \vee q_1 & \textbf{provesgoal } \gamma & \stackrel{def}{=} & \textbf{ingoals } [\gamma] \wedge \textbf{outgoals } [] \\
\textbf{isthen} & \stackrel{def}{=} & * \textbf{ then } * & \textbf{axiom } nm & \stackrel{def}{=} & \textbf{atomic } nm \wedge \textbf{outgoals } [] \\
\textbf{isbeside} & \stackrel{def}{=} & * \textbf{ beside } * & \textbf{islabel } nm & \stackrel{def}{=} & \textbf{inside } nm \; *
\end{array}
$$

$$
\begin{array}{rcl}
\textbf{whenin } nm \; q & \stackrel{def}{=} & \textbf{inside } nm \; q \textbf{ when islabel } nm \\
\textbf{somewherebeside } q & \stackrel{def}{=} & \mu Q.\, q \vee (Q \textbf{ beside } *) \vee (* \textbf{ beside } Q) \\
\textbf{nearby } q & \stackrel{def}{=} & \mu Q.\, q \vee (Q \textbf{ then } *) \;\vee\; (* \textbf{ then } Q) \\
 & & \quad \vee (Q \textbf{ beside } *) \;\vee\; (* \textbf{ beside } Q) \\
\textbf{separately } q_1 \textbf{ and } q_2 & \stackrel{def}{=} & \mu Q.\, (\textbf{inside } * \; Q) \\
 & & \quad \vee (\textbf{somewhere } q_1 \textbf{ then somewhere } q_2) \\
 & & \quad \vee (\textbf{somewhere } q_1 \textbf{ beside somewhere } q_2)
\end{array}
$$

The **when** conditional combinator is satisfied if q_1 is satisfied whenever q_2 is; by convention, the scope of q_1 and q_2 extend as far as possible. The last three combinators again use recursion to expand the scope of the local structure specifications. The query **somewherebeside** q is satisfied if q is satisfied in a $\otimes$-list of hiproofs; **nearby** q is an adjusted version of **somewhere** which restricts to the same level, without descending into boxes. The query **separately** q_1 **and** q_2 requires that q_1 and q_2 hold on disjoint portions of the proof.

3.1 Examples

We show some of our motivating examples relating proofs and tactics. First, the tactic tac occurs recursively in a hiproof if the query

$$\textbf{somewhere inside } \mathsf{tac} \textbf{ somewhere islabel } \mathsf{tac}$$

is satisfied. The tactic inner always occurs whenever the tactic outer is invoked if this query is satisfied:

$$\textbf{everywhere whenin } \mathsf{outer} \textbf{ somewhere islabel } \mathsf{inner}.$$

More elaborately, a tactic named base always appears alongside a tactic named step inside the tactic induct:

$$\textbf{everywhere whenin } \mathsf{induct} \textbf{ somewhere } (\textbf{somewherebeside islabel } \mathsf{base}) \wedge (\textbf{somewherebeside islabel } \mathsf{step}).$$

Examples returning results are given in Sect. 4.1 after introducing the semantics.

4 Semantics

We will define the semantics of queries using a satisfaction relation $s \models_\sigma q$. This denotes satisfaction of a query on a hiproof s with respect to a substitution σ for match variables. The substitution maps variables N to names for atomic tactics and labels, and variables G to lists of the form $[\gamma_1, \ldots, \gamma_n]$.

Two base satisfaction relations define matching on names and goal lists:

$$\begin{array}{rcll} * & \models_\sigma & n & \text{always} \\ n' & \models_\sigma & n & \text{iff } n = n' \\ \xi & \models_\sigma & n & \text{iff } \xi(n) \\ N & \models_\sigma & n & \text{iff } \sigma(N) = n \\ (\neg N) & \models_\sigma & n & \text{iff } \neg(N \models_\sigma n) \end{array} \qquad \begin{array}{rcll} [\psi_1, \ldots, \psi_n] & \models_\sigma & g & \text{iff } \exists \gamma_1 \cdots \gamma_n.\, g = [\gamma_1, \ldots, \gamma_n] \\ & & & \quad \text{and } \psi_1(\gamma_1) \cdots \psi_n(\gamma_n) \\ G & \models_\sigma & g & \text{iff } \sigma(G) = g \\ (\neg G) & \models_\sigma & g & \text{iff } \neg(G \models_\sigma g) \end{array}$$

Before giving the main relation, we consider hiproof terms in more detail. Terms s in the hiproof grammar denote tree-based models in the denotational semantics of hiproofs [1]. Under the denotational interpretation, certain terms are equivalent. We will give our interpretation over the syntax, considering valid hiproofs modulo the following equations generating this equivalence:

$$\begin{array}{rcl} s \,;\, \mathsf{id} = s & \quad \mathsf{id} \,;\, s = s & \quad \mathsf{id} \text{ is an identity for sequencing} \\ s \otimes \langle\rangle = s & \quad \langle\rangle \otimes s = s & \quad \langle\rangle \text{ is an identity for juxtaposition} \\ s \,;\, \langle\rangle = s & & \quad \langle\rangle \text{ is a right-identity for sequencing} \\ s_1 \,;\, (s_2 \,;\, s_3) = (s_1 \,;\, s_2) \,;\, s_3 & & \quad ; \text{ is associative} \\ s_1 \otimes (s_2 \otimes s_3) = (s_1 \otimes s_2) \otimes s_3 & & \quad \otimes \text{ is associative} \\ (s_1 \,;\, s_2) \otimes (s_3 \,;\, s_4) = (s_1 \otimes s_3) \,;\, (s_2 \otimes s_4) & & \quad ; \text{ and } \otimes \text{ can be exchanged} \end{array}$$

It is easy to confirm that the equations preserve validity on the same lists of input and output goals for the rules in Fig. 2. We will write $s = s'$ if two terms are equal in the theory generated by these equations (i.e., closing under congruence).

Definition 1 (Query satisfaction). *Let s be a valid hiproof and q a query in the minimal query language. The satisfaction of q for s with the substitution σ is defined as the least relation $s \models_\sigma q$ satisfying:*

$$\begin{array}{rcll} s & \models_\sigma & * & \textit{when } s \neq \langle\rangle \\ a & \models_\sigma & \textbf{atomic } nm & \textit{when } nm \models_\sigma a \\ \mathsf{id} & \models_\sigma & \textbf{nothing} & \\ [l]\, s & \models_\sigma & \textbf{inside } nm\ q & \textit{when } nm \models_\sigma l \textit{ and } s \models_\sigma q \\ s_1 \,;\, s_2 & \models_\sigma & q_1 \textbf{ then } q_2 & \textit{when } s_1 \models_\sigma q_1 \textit{ and } s_2 \models_\sigma q_2 \\ s_1 \otimes s_2 & \models_\sigma & q_1 \textbf{ beside } q_2 & \textit{when } s_1 \models_\sigma q_1 \textit{ and } s_2 \models_\sigma q_2 \\ s & \models_\sigma & \textbf{ingoals } gm & \textit{when } gm \models_\sigma g \textit{ where } s \vdash g \longrightarrow h \\ s & \models_\sigma & \textbf{outgoals } gm & \textit{when } gm \models_\sigma h \textit{ where } s \vdash g \longrightarrow h \\ s & \models_\sigma & q_1 \wedge q_2 & \textit{when } s \models_\sigma q_1 \textit{ and } s \models_\sigma q_2 \\ s & \models_\sigma & q_1 \vee q_2 & \textit{when } s \models_\sigma q_1 \textit{ or } s \models_\sigma q_2 \\ s & \models_\sigma & \neg q & \textit{when } \neg(s \models_\sigma q) \\ s & \models_\sigma & \mu Q.q & \textit{when } s \models_\sigma q[\mu Q.q/Q] \\ s & \models_\sigma & q & \textit{when } \exists s'.\, s' \models_\sigma q \textit{ and } s' = s. \end{array}$$

Recursive queries $\mu Q.q$ are interpreted using unfolding; this suffices since we query only finitely deep trees. More precisely, we can define satisfaction using an auxiliary relation $\models_n$ indexed by the maximum depth of the number of unfoldings of a recursive query, where $\mu_n Q.q$ can be unfolded at most n times. Then $\models$ is defined as the union of all finite unfolding relations $\models_n$. The definition works for singly recursive queries where we do not need to interpret queries with free query variables, but can be extended for mutually recursive queries.

Proposition 1. *Let s be a valid hiproof. Then*

1. $s \models_\sigma$ **somewhere** q *iff* $\exists s'.s'$ *is a subterm of* s *and* $s' \models_\sigma q$,
2. $s \models_\sigma$ **everywhere** q *iff* $\forall s'.s'$ *is a subterm of* s *and* $s' \models_\sigma q$.

(where quantification ranges over non-empty terms, and s is a subterm of itself).

Thus these important derived forms have the intended meanings.

How precise are our queries? The following proposition establishes, as intended, that every term can be characterised up to equality by a query. Thus, we can use queries to describe finite sets of hiproofs.

Proposition 2. *Given any hiproof s not containing $\langle\rangle$, there is a query $Q(s)$ which characterises s precisely.*

Proof. *Let $Q(s)$ be given by the embedding:*

$$\begin{aligned} Q(a) &= \textbf{atomic}\ a \\ Q(\mathsf{id}) &= \textbf{nothing} \\ Q([l]\,s) &= \textbf{inside}\ l\ Q(s) \\ Q(s_1\ ;\ s_2) &= Q(s_1)\ \textbf{then}\ Q(s_2) \\ Q(s_1 \otimes s_2) &= Q(s_1)\ \textbf{beside}\ Q(s_2) \end{aligned}$$

Now we claim that whenever $s' \models_\sigma Q(s)$ for some s', we must have $s = s'$.

Using a simple normal form, Prop. 2 can be extended to cover all hiproofs.

4.1 Examples and Their Results

Now we demonstrate the remainder of our motivating queries; meanings can be calculated using the semantics above to show that they are correct. The invocation of a query to get some results can be written in SQL style as:

$$\textbf{select}\ e\ \textbf{from}\ s\ \textbf{where}\ q$$

which denotes the set of expressions $\sigma(e)$ for all substitutions σ that satisfy the query (see Sect. 5 on how this can be implemented). That is:

$$\{\sigma(e) \mid s \models_\sigma q\}.$$

The kind of expressions e chosen here depends on what we want to do with query results. We don't consider a general transformation language for query results here, but one could easily allow expressions that combine pieces of query results in arbitrary ways. Our examples below restrict to simple query variables.

- To find all the axioms in a valid hiproof s:

$$\mathit{Axioms}(s) = \textbf{select } A \textbf{ from } s \textbf{ where somewhere axiom } A$$

  Applied to $s = ([l]\, a\ ;\ b \otimes \mathsf{id})\ ;\ [m]\, c$, this query returns $\{A \mapsto \mathsf{c}, A \mapsto \mathsf{b}\}$.
- To find the existential witnesses inside a valid hiproof s, we can find uses of the existential introduction rule:

$$\mathit{Wit}(s) = \textbf{select } A \textbf{ from } s \textbf{ where somewhere atomic } A \wedge \textbf{atomic } \mathit{ExI}_t$$

  Here, the *ExI* rule is annotated by the witness t that is chosen as part of its name, and we use ExI_t to denote the predicate selecting all such rule names.
- Which tactics are used in a proof?

$$\mathit{Tactics}(s) = \textbf{select } L \textbf{ from } s \textbf{ where somewhere inside } L\ *$$

- Which goals are input to (or output from) a tactic called tac?

$$\mathit{Input}(\mathsf{tac}, s) = \textbf{select } G \textbf{ from } s \textbf{ where somewhere inside } \mathsf{tac} \textbf{ ingoals } G$$

$$\mathit{Output}(\mathsf{tac}, s) = \textbf{select } G \textbf{ from } s \textbf{ where somewhere inside } \mathsf{tac} \textbf{ outgoals } G$$

- Which tactics call themselves recursively? (shown earlier for fixed tac)

$$\mathit{Rec}(s) = \textbf{select } L \textbf{ from } s \textbf{ where somewhere inside } L \textbf{ somewhere islabel } L$$

- Which tactic uses atomic tactic a, i.e., inside which label does a occur? Using the **nearby** combinator defined in the last section, this query returns all labels L which contain a directly, i.e., labels which are the immediate surrounding parent of a, not a more distant ancestor.

$$\mathit{Inside}(a, s) = \textbf{select } L \textbf{ from } s \textbf{ where somewhere inside } L \textbf{ nearby atomic } a$$

- Are there steps in the proof which have no effect?

$$\mathit{UselessTacs}(s) = \textbf{select } L \textbf{ from } s \textbf{ where somewhere inside } L \textbf{ ingoals } G \wedge \textbf{outgoals } G$$

  This returns useless tactics that return the same goal that they were given (necessarily G is a single element list by the hiproof structure). Some tactics may be even worse and return the same goal that they were given and more besides! To catch those, we could add subset inclusion to goal matching.

– Are there duplicated subproofs inside a proof? We answer this by finding labelled subtrees that have the same input and output goals, using the **separately** operator introduced earlier:

$$\begin{aligned} \mathit{Duplicates}(s) = \textbf{select } L_1, L_2, G_i, G_o \textbf{ from } s \textbf{ where} \\ \textbf{separately inside } L_1\ q \textbf{ and inside } L_2\ q \end{aligned}$$

where q abbreviates **ingoals** $G_i \wedge$ **outgoals** G_o.

In the last example, we might want to return (or replace) the actual duplicate subtrees. To do that we would need to add variables ranging over hiproofs (or paths in hiproofs) to the language; see Sect. 6 for remarks on this extension.

4.2 Query Equivalence and Decidability

Prop. 2 characterises proofs by queries. We can turn this around, and ask whether queries can be characterised by the proofs that satisfy them. This motivates a Leibniz-style equality between queries.

Definition 2. *Two queries p, q are* equivalent*, written $p \cong q$, if for all proofs s and substitutions σ, we have $s \models_\sigma q \iff s \models_\sigma p$.*

We can now state a number of equations over queries. These are proven by expanding Def. 2 and using Def. 1. First, conjunction and disjunction commute over the basic queries; we write this as a family of equations:

$$\textbf{inside } nm\ (p \Diamond q) \cong (\textbf{inside } nm\ p) \Diamond (\textbf{inside } nm\ pq) \tag{2}$$

$$(p_1 \Diamond p_2) \oplus q \cong (p_1 \oplus q) \Diamond (p_2 \oplus q) \tag{3}$$

$$p \oplus (q_1 \Diamond q_2) \cong (p \oplus q_1) \Diamond (p \oplus q_2) \tag{4}$$

for $\Diamond \in \{\wedge, \vee\}$ and $\oplus \in \{\textbf{then}, \textbf{beside}\}$. Negation distributes over the basic queries variously. E.g., the query **ingoals** gm is not satisfied by s iff the goals of s do not match gm, whereas the query **atomic** am is not satisfied by s iff either s is an atom that does not match am, or if it is not an atom. We give three equations, and omit similar ones for **outgoals**, **inside**, **then**, and **nothing**:

$$\neg(\textbf{ingoals } gm) \cong \textbf{ingoals } (\neg gm) \tag{5}$$

$$\begin{aligned} \neg(\textbf{atomic } am) \cong\ & \textbf{atomic } (\neg am) \vee (\textbf{islabel } *) \\ & \vee \textbf{nothing} \vee \textbf{isbeside} \vee \textbf{isthen} \end{aligned} \tag{6}$$

$$\begin{aligned} \neg(p \textbf{ beside } q) \cong\ & ((\neg p) \textbf{ beside } *) \vee (* \textbf{ beside } (\neg q)) \\ & \vee (\textbf{atomic } *) \vee \textbf{nothing} \vee (\textbf{islabel } *) \vee \textbf{isthen} \end{aligned} \tag{7}$$

Finally, we have the usual laws of propositional logic: De Morgan equalities, double negation, commutativity and distributivity of conjunction and disjunction. By reading our equations as rewrite rules from left to right, we get a decision procedure for equivalence of queries, as long as they do not contain any recursion.

Definition 3 (DNF). *A query q is in disjunctive normal form (DNF), if it is of the shape $\bigvee_{i=1\ldots n} \bigwedge_{j=1\ldots m_i} \phi_{i,j}$ where $\phi_{i,j}$ are basic queries, or in other words a disjunction of conjunctions of basic queries.*

Proposition 3. *For each recursion-free query q there is an equivalent query q' in DNF, denoted as DNF(q).*

The *size* of *DNF*(q) is exponential in the size of q. Most equations are linear in the query argument (that is, the query arguments occur once on each side of the equation), and hence only introduce a constant size increase when applied left to right, but (3) and (4) and similarly distributivity for $\vee$ and $\wedge$ contain the query argument q and p twice on the right-hand side. Thus, each of **then**, **besides** or $\wedge$ may double the size, leading to exponential increase. Of course, the size of the resulting *DNF*(q) will usually be much smaller; we can cut it further down by eliminating contradictory conjunctions such as **atomic** $a \wedge$ **isthen**.

Checking that a basic query q satisfies a given hiproof s is linear in the size of q, as we just traverse the structure of q and s. Hence, checking that a query q' in DNF satisfies a given hiproof s is also linear in the size of q', as we merely need to check each of the basic queries $\phi_{i,j}$ against s. Hence, because of the size of *DNF*(q), satisfiability of recursion-free queries is decidable in exponential time.

Proposition 3 does not hold for queries containing the recursion operator. To check that a given recursive query q and substitution σ satisfy a hiproof s, we can unfold the recursion in q as often as needed, and then use the DNF of the unfolded term. The size of the hiproof s bounds the size of the unfolding, as a hiproof cannot be smaller than a basic query it satisfies, and *DNF*(q) is always larger than q. Thus:

Proposition 4. *$s \models_\sigma q$ is decidable in exponential time over size(s) + size(q).*

This straightforward argument establishes decidability. Better complexity bounds surely exist, as they are known for related query languages and various fragments (see e.g., [5]), but mappings into other languages are beyond our scope here.

5 Implementing Queries

We have built a simple implementation of the query language in order to validate its design by running example queries on small proofs and checking the results. We directly use the semantics and turn Def. 1 into a function $sat(s, q)$ which implements the **select** statement from Sect. 4.1 and returns the (minimal) set of all substitutions which satisfy q.

Substitutions are given as partial functions $Var \rightharpoonup \mathcal{T}$, where $\mathcal{T}$ is the set of names or goal lists. Given two substitutions ρ and σ, their unification $unify(\rho, \sigma)$ is defined iff $\forall a \in dom\,\rho \cup dom\,\sigma.\rho(a) = \sigma(a)$, and it is defined pointwise to be $\sigma(a)$ if $\sigma(a)$ is defined, $\rho(a)$ if $\rho(a)$ is defined, and undefined everywhere else. To combine two sets Φ and Ψ of substitutions, as returned by recursive calls of the *sat* function, we define the combinator

$$\Psi \gg \Phi = \{unify(\rho, \sigma) \mid \rho \in \Psi, \sigma \in \Phi, unify(\rho, \sigma) \text{ is defined}\}$$

For the basic queries, there are simple functions sat_N and sat_G which return the set of substitutions matching a given name or goal match. Then *sat* can be recursively defined as follows (we only give some of the representative cases):

$$\begin{aligned} sat(a, \textbf{atomic}\ nm) &= sat_N(a, nm) \\ sat([l]\, s, \textbf{inside}\ nm\ q) &= sat_N(l, nm) \gg sat(s, q) \\ sat(s_1\ ;\ s_2, q_1\ \textbf{then}\ q_2) &= sat(s_1, q_1) \gg sat(s_2, q_2) \\ sat(s, q_1 \wedge q_2) &= sat(s, q_1) \gg sat(s, q_2) \\ sat(s, q_1 \vee q_2) &= sat(s, q_1) \cup sat(s, q_2) \end{aligned}$$

Note that when combining the results for a disjunctive query, we can just take the union of the results. We can show the correctness of this definition, namely that if $\sigma \in sat(s, q)$ then $s \models_\sigma q$, and also that if $s \models_\sigma q$ then there is $\rho \in sat(s, q)$ such that $\rho \subseteq \sigma$ (so *sat* returns a minimal set of substitutions).

Implementation. Using this definition, our prototype implements the query language for small experiments. It represents queries as an algebraic datatype `Q`, and in time-honoured fashion uses SML as both implementation platform and scriptable command-line interface. Hiproofs are represented modulo the equations in Sect. 4, following the denotational semantics in [1]. The implementation is a functor which is generic over the proofs in question, reflecting the generic nature of the query language.

We provide two instantiations of the generic implementation: one for the syntactic hiproofs, where we have a datatype `S` as in (1), and one which models Isabelle proof objects [6] as hiproofs. Taking existing proofs such as those in Isabelle as hiproofs, we need to derive the hierarchical structure. We use theorems to do this. That is, a box $[l]\, s$ is a theorem named l, together with its proof s. This leads to an interesting example: the query $Rec(s)$ applied to an Isabelle hiproof would return all theorems which are used in their own proof.

6 Related Work and Conclusions

This paper introduced *locally structured* proof queries in our proof query language, PrQL. These build up patterns of structure that are matched to a position in the implicit tree. Using logical connectives, variable substitution and structural recursion, queries can span and relate different portions of the tree and express many natural queries on proofs. But, in this locally structured fragment it is not possible to write a query that directly refers to (or returns) a position in the tree, or does any counting. This limitation can be lifted, e.g., by adding a notion of path to the language. In future work we will report on *globally structured* queries allowed by this, as well as a slightly different language where queries are defined directly over our semantic models.

Related Work in Theorem Proving. The idea of a general query language for inspecting formal proofs appears novel, although there are many investigations

into exploiting proofs in particular ad hoc ways. We can't survey all but mention a few. Researchers have connected decision procedures to theorem proving by grafting invocation records of decision procedures (with possible justifications) into an overall proof (e.g., [7]). Noteworthy sub-trees may be represented using names for reference (and then shared to create a dag structure) as in TPTP and its proof format TSTP [8]. Many systems use debugging output for proof procedures to create a lengthy log, which explains where things were tried and failed. Some tools use representations of proof trees in the first place which connect the proof-producing mechanism to the proof and are equipped with browsing and editing mechanisms, e.g., NuPrl [9]. Besides checking proofs [10], other researchers have made efforts to translate proofs between systems [11]; discover dependencies between parts of proofs [12] to help simplify or rearrange; and data-mine proofs to discover common patterns [13].

To exploit a formal generic proof representation language like hiproofs, it is appealing to use a generic concrete representation like TSTP. A TSTP proof consists of the sequence of formulas output by an automated theorem prover along with their sources, and is hence a more "operational" format than hiproofs, which can be translated into TSTP in either forwards-style (deriving conclusions from axioms) or backwards-style (decomposing conjectures to back to axioms). Going in the opposite direction, although TSTP does not represent tactics, inference rules can be nested, giving a simple form of hierarchy. We could decompose the derivations in various ways thus deriving an implicit hierarchy, or extend the language with labels on sub-derivations to represent hierarchy explicitly. Proofs in the TSTP archive can be queried online [14] using a range of primitive and quantitative predicates, or by translation [15] into the Proof Markup Language (PML) [16], which serves as an interlingua representation for the justification of results produced by Semantic Web services. Queries in PML are simply partial proofs, rather than expressions in a separate query language (of course, PrQL also has close ties to its underlying proof language), and query evaluation seeks to return (possibly partial) proofs that "fill in the blanks" in the initial query. Our original motivation for developing a query language was to extract information from TSTP proofs in order to construct safety cases, and we plan to extend our prototype implementation to support this.

Query Languages for Structured Data and Programs. Away from theorem proving, query languages for trees and graphs have been studied for some time. Languages related to PrQL include those aimed at semi-structured (XML-like) models such as UnQL [5] which uses structural recursion on tree (and graph) representations, similarly to PrQL's recursive queries, and Graph Logic [17] which uses a separating conjunction to destruct the graph subject of queries. Checking for patterns in programs, ASTLog [18] is a Prolog variant for examining syntax trees and PQL [19] is a more general framework for querying programs at varying levels of abstraction. Establishing precise connections with PrQL would let us exploit known complexity results, existing algorithms and their implementations.

Acknowledgements. We would like to thank Geoff Sutcliffe for help with TPTP.

References

1. Denney, E., Power, J., Tourlas, K.: Hiproofs: A hierarchical notion of proof tree. Electr. Notes Theor. Comput. Sci. 155, 341–359 (2006)
2. Aspinall, D., Denney, E., Lüth, C.: Tactics for hierarchical proof. Mathematics in Computer Science 3(3), 309–330 (2010)
3. Angles, R., Gutierrez, C.: Survey of graph database models. ACM Comput. Surv. 40 (February 2008)
4. Bailey, J., Bry, F., Furche, T., Schaffert, S.: Web and Semantic Web Query Languages: A Survey. In: Eisinger, N., Małuszyński, J. (eds.) Reasoning Web. LNCS, vol. 3564, pp. 35–133. Springer, Heidelberg (2005)
5. Buneman, P., Fernandez, M., Suciu, D.: UnQL: a query language and algebra for semistructured data based on structural recursion. The VLDB Journal 9(1), 76–110 (2000)
6. Berghofer, S., Nipkow, T.: Proof Terms for Simply Typed Higher Order Logic. In: Aagaard, M.D., Harrison, J. (eds.) TPHOLs 2000. LNCS, vol. 1869, pp. 38–52. Springer, Heidelberg (2000)
7. Harrison, J., Théry, L.: A skeptic's approach to combining HOL and Maple. Journal of Automated Reasoning 21, 279–294 (1998)
8. 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)
9. Allen, S.F., Bickford, M., Constable, R.L., Eaton, R., Kreitz, C., Lorigo, L., Moran, E.: Innovations in computational type theory using Nuprl. Journal of Applied Logic 4(4), 428–469 (2006)
10. Necula, G., Lee, P.: Proof Generation in the Touchstone Theorem Prover. In: McAllester, D. (ed.) CADE-17. LNCS (LNAI), vol. 1831, pp. 25–44. Springer, Heidelberg (2000)
11. Denney, E.: A Prototype Proof Translator from HOL to Coq. In: Aagaard, M.D., Harrison, J. (eds.) TPHOLs 2000. LNCS, vol. 1869, pp. 108–125. Springer, Heidelberg (2000)
12. Pons, O., Bertot, Y., Rideau, L.: Notions of dependency in proof assistants. In: Proc. User Interfaces for Theorem Provers, UITP 1998 (1998)
13. Urban, J.: MizarMode - an integrated proof assistance tool for the Mizar way of formalizing mathematics. J. Applied Logic 4(4), 414–427 (2006)
14. Sutcliffe, G., Suttner, C.: The TPTP problem library for automated theorem proving, Homepage and online tools, `http://www.tptp.org` (visited November 2011)
15. da Silva, P.P., Sutcliffe, G., Chang, C., Ding, L., Rio, N.D., McGuinness, D.L.: Presenting TSTP proofs with inference web tools. In: Konev, B., Schmidt, R.A., Schulz, S. (eds.) PAAR/ESHOL. CEUR Workshop Proceedings, vol. 373. CEUR-WS.org (2008)
16. da Silva, P.P., McGuinness, D., Fikes, R.: A proof markup language for semantic web services. Information Systems 31(4-5), 381–395 (2006)
17. Cardelli, L., Gardner, P., Ghelli, G.: A Spatial Logic for Querying Graphs. In: Widmayer, P., Triguero, F., Morales, R., Hennessy, M., Eidenbenz, S., Conejo, R. (eds.) ICALP 2002. LNCS, vol. 2380, pp. 597–610. Springer, Heidelberg (2002)
18. Crew, R.F.: ASTLOG: A language for examining abstract syntax trees. In: DSL, USENIX (1997)
19. Jarzabek, S.: Design of flexible static program analyzers with PQL. IEEE Trans. Software Eng. 24(3), 197–215 (1998)

Solving Language Equations and Disequations with Applications to Disunification in Description Logics and Monadic Set Constraints*

Franz Baader[1] and Alexander Okhotin[2]

[1] Institute for Theoretical Computer Science, TU Dresden, Germany
[2] Department of Mathematics, University of Turku, Finland

Abstract. We extend previous results on the complexity of solving language equations with one-sided concatenation and all Boolean operations to the case where also disequations (i.e., negated equations) may occur. To show that solvability of systems of equations and disequations is still in ExpTime, we introduce a new type of automata working on infinite trees, which we call looping automata with colors. As applications of these results, we show new complexity results for disunification in the description logic $\mathcal{FL}_0$ and for monadic set constraints with negation. We believe that looping automata with colors may also turn out to be useful in other applications.

1 Introduction

Equations with formal languages as constant parameters and unknowns are among the basic notions of *formal language theory*, first introduced by Ginsburg and Rice [9], who gave a characterization of the context-free languages by solutions of systems of equations of the resolved form $X_i = \varphi_i(X_1, \ldots, X_n)$. For equations of the general form $\varphi(X_1, \ldots, X_n) = \psi(X_1, \ldots, X_n)$ built using union and two-sided concatenation, testing their solvability is easily shown to be undecidable [15]. The state-of-the-art in this area as of 2007 is presented in a survey by Kunc [11]. More recent work shows that undecidability already holds for equations over a one-letter alphabet with concatenation as the only operation [10,12]. In contrast, solvability of language equations with concatenation restricted to *one-sided concatenation with constants* can often be shown to be decidable by encoding the problem into monadic second-order logic on infinite trees (MSO) [16], but this usually does not yield optimal complexity results.

In *logic for programming and artificial intelligence*, language equations with one-sided concatenation are, for instance, relevant in the context of monadic set constraints and unification in description logics (DLs). *Unification in DLs* has been proposed [4] as a novel inference service that can, for example, be used to detect redundancies in ontologies. As a simple example, assume that one knowledge engineer has defined the concept of "women having only daughters" by the

* Supported by DFG (BA 1122/14-1) and the Academy of Finland (grant 134860).

N. Bjørner and A. Voronkov (Eds.): LPAR-18, LNCS 7180, pp. 107–121, 2012.

concept term $\mathsf{Woman} \sqcap \forall \mathsf{child}.\mathsf{Woman}$. A second knowledge engineer might represent this notion in a somewhat more fine-grained way, e.g., by using the term $\mathsf{Female} \sqcap \mathsf{Human}$ in place of Woman. The concept terms $\mathsf{Woman} \sqcap \forall \mathsf{child}.\mathsf{Woman}$ and $\mathsf{Female} \sqcap \mathsf{Human} \sqcap \forall \mathsf{child}.(\mathsf{Female} \sqcap \mathsf{Human})$ are not equivalent, but they are meant to represent the same concept. The two terms can obviously be made equivalent by viewing the concept name Woman as a concept variable and replacing it in the first term by the concept term $\mathsf{Female} \sqcap \mathsf{Human}$. Unification in DLs checks for the existence of such substitutions, and thus can be used to alert the knowledge engineers to potential redundancies in the ontology. In [4] it was shown that unification in the DL $\mathcal{FL}_0$ can be reduced to *finite solvability*[1] of language equations with one-sided concatenation and union, and that this problem is in turn ExpTime-complete. In [3] it was shown that the same complexity result holds for *solvability*,[2] and in [5] this result was extended to language equations with one-sided concatenation and *all Boolean operations*, and to *other decision problems than just solvability*.

Language equations with one-sided concatenation and all Boolean operations can also be regarded as a particular case of equations on sets of terms, known as *set constraints*, which received significant attention [14] in logic for programming since they can be used in program analysis. In fact, solvability of such language equations corresponds to solvability of *monadic* set constraints, where all function symbols are at most unary. In [1] it was already shown that solvability of monadic set constraints is an ExpTime-complete problem.

In the present paper, we extend the existing results for language equations with one-sided concatenation and all Boolean operations to the case of finite systems of language equations and *disequations* (i.e., negated equations). We will show that solvability and finite solvability of such systems are still *in ExpTime*. The motivation comes again from description logics and from set constraints. Set constraints with negation have been investigated in several papers [8,17,2], where it is shown that solvability in the general case is NExpTime-complete. The exact complexity of the monadic case has, to the best of our knowledge, not been determined yet. In description logics, it makes sense to consider not only unification, but also disunification problems in order to prevent certain unifiers. For example the concept term $\mathsf{Woman} \sqcap \forall \mathsf{child}.\mathsf{Woman}$ also unifies with $\mathsf{Male} \sqcap \mathsf{Human} \sqcap \forall \mathsf{child}.(\mathsf{Male} \sqcap \mathsf{Human})$, which could, e.g., be prevented by stating that Woman should not become a subconcept of Male, i.e., that $\mathsf{Woman} \sqcap \mathsf{Male}$ must not be unified with Woman.

In Section 2, we formally define language equations and disequations with one-sided concatenation and all Boolean operations, and show that their (finite) solvability can be reduced to the existence of certain runs of a corresponding looping tree automaton. In Section 3, we introduce looping tree automata with colors, which can express the condition on the runs formulated in the previous section, and then analyze the complexity of their emptiness problem. Finally, in Section 4 we use these results to determine the complexity of testing (finite)

[1] i.e., existence of a solution consisting of finite languages.
[2] i.e., existence of a solution consisting of arbitrary (not necessarily finite) languages.

solvability of the systems of language (dis)equations introduced in Section 2, and then in turn apply this result to identify the complexity of solving disunification problems in $\mathcal{FL}_0$ as well as monadic set constraints with negation.

2 Language (Dis)equations With One-Sided Concatenation

In this section, we first introduce the language (dis)equations that we want to solve, and then we show how solvability can be reduced to a problem for looping automata working on infinite trees.

2.1 The Problem Definition

Given a finite alphabet Σ and finitely many variables $X_1, \ldots, X_n$, the set of *language expressions* is defined by induction:

- any variable X_i is a language expression;
- the empty word ε is a language expression;
- a concatenation φa of a language expression φ with a symbol $a \in \Sigma$ is a language expression;[3]
- if φ, φ' are language expressions, then so are $(\varphi \cup \varphi')$, $(\varphi \cap \varphi')$ and $(\sim\varphi)$.

Given a mapping $\theta = \{X_1 \mapsto L_1, \ldots, X_n \mapsto L_n\}$ of the variables to languages $L_1, \ldots, L_n$ over Σ, its extension to language expressions is defined as

- $\theta(X_i) := L_i$ for all $i, 1 \leq i \leq n$;
- $\theta(\varepsilon) := \{\varepsilon\}$;
- $\theta(\varphi a) := \theta(\varphi) \cdot \{a\}$ for $a \in \Sigma$;
- $\theta(\varphi \cup \varphi') := \theta(\varphi) \cup \theta(\varphi')$, $\theta(\varphi \cap \varphi') := \theta(\varphi) \cap \theta(\varphi')$, and $\theta(\sim\varphi) := \Sigma^* \setminus \theta(\varphi)$.

We call such a mapping a *substitution*.

A *language equation* is of the form $\varphi = \psi$ and a *language disequation* is of the form $\varphi \neq \psi$, where φ, ψ are language expressions. The substitution θ solves the equation $\varphi = \psi$ (the disequation $\varphi \neq \psi$) iff $\theta(\varphi) = \theta(\psi)$ ($\theta(\varphi) \neq \theta(\psi)$). We are interested in solvability of finite systems of language equations and disequations, where a substitution θ solves such a system iff it solves every (dis)equation in the system. Such a solution is called *finite* iff the languages $L_1 = \theta(X_1), \ldots, L_n = \theta(X_n)$ are finite.

Using the fact that, for any sets M_1, M_2, we have $M_1 = M_2$ iff $(M_1 \setminus M_2) \cup (M_2 \setminus M_1) = \varnothing$ and $M_1 = \varnothing = M_2$ iff $M_1 \cup M_2 = \varnothing$, we can transform a given finite system of language equations and disequations into an equivalent one (i.e., one with the same set of solutions) of the form

$$\varphi = \varnothing, \quad \psi_1 \neq \varnothing, \quad \ldots, \quad \psi_k \neq \varnothing. \tag{1}$$

In order to test such a system for (finite) solvability, we translate it into a looping tree automaton.

[3] Note that the concatenation is *one-sided* in the sense that constants ($a \in \Sigma$) are only concatenated from the right to expressions.

2.2 Translation into Looping Tree Automata

Given a ranked alphabet Γ, where every symbol has a nonzero rank, infinite trees over Γ are defined in the usual way, that is, every node in the tree is labeled with an element $f \in \Gamma$ and has as many successor nodes as is the rank of f. A *looping tree automaton* $\mathcal{A} = (Q, \Gamma, Q_0, \Delta)$ consists of a finite set of states Q, a ranked alphabet Γ, a set of initial states $Q_0 \subseteq Q$, and a transition function $\Delta : Q \times \Gamma \to 2^{Q^*}$ that maps each pair (q, f) to a subset of Q^k, where k is the rank of f. A *run* r of $\mathcal{A}$ on a tree t labels the nodes of t with elements of Q, such that the root is labeled with $q_0 \in Q_0$, and the labels respect the transition function, that is, if a node v has label $t(v)$ in t and label $r(v)$ in r, then the tuple $(q_1, \ldots, q_k)$ labeling the successors of v in r must belong to $\Delta(q, t(v))$. The tree t is *accepted* by $\mathcal{A}$ if there is a run of $\mathcal{A}$ on t. The *language accepted by the looping tree automaton* $\mathcal{A}$ is defined as

$$L(\mathcal{A}) := \{t \mid t \text{ is an infinite tree over } \Gamma \text{ that is accepted by } \mathcal{A}\}.$$

It is well-known that the *non-emptiness problem* for looping tree automata, that is, the question whether, given such an automaton $\mathcal{A}$, the accepted language $L(\mathcal{A})$ is non-empty, is decidable in linear time [7].

When reducing a finite system of language (dis)equations of the form (1) to a looping tree automaton, we actually consider a very restricted case of looping tree automata. Assume that the alphabet used in the system is $\Sigma = \{a_1, \ldots, a_m\}$. Then we restrict our attention to a ranked alphabet Γ containing a single symbol γ of rank m. Thus, there is only one infinite tree, and the labeling of its nodes by γ can basically be ignored. Every node in this tree can be uniquely represented by a word $w \in \Sigma^*$, where each symbol a_i selects the ith successor of a node. Consequently, any run on this tree of a looping tree automaton with set of states Q can be represented as a mapping from Σ^* to Q.

Given a finite system of language (dis)equations of the form (1), let Φ denote the set of all subexpressions of $\varphi, \psi_1, \ldots, \psi_k$. We assume that $\varepsilon, X_1, \ldots, X_n \in \Phi$ (otherwise, we simply add them). In [5] we have shown how to construct a looping tree automaton $\mathcal{A}$ with the set of states $Q := 2^{\Phi}$, and with a 1–1-correspondence between runs of $\mathcal{A}$ and substitutions. To be more precise, given a run $r : \Sigma^* \to Q$ of $\mathcal{A}$, the corresponding substitution $\theta^r = \{X_1 \mapsto L_1^r, \ldots, X_n \mapsto L_n^r\}$ is obtained by defining

$$L_i^r := \{w \in \Sigma^* \mid X_i \in r(w)\}.$$

Conversely, given a substitution $\theta = \{X_1 \mapsto L_1, \ldots, X_n \mapsto L_n\}$, the corresponding run r_θ is

$$r_\theta(w) := \{\xi \in \Phi \mid w \in \theta(\xi)\}.$$

Lemma 1 ([5]). *The mapping of runs to substitutions introduced above is a bijection, and the mapping of substitutions to runs is its inverse.*

How do runs that correspond to solutions look like? Given a substitution θ, the corresponding run r_θ satisfies

$$\xi \in r_\theta(w) \quad \text{iff} \quad w \in \theta(\xi)$$

for all $\xi \in \Phi$. Recall that our system is of the form (1) and that $\varphi, \psi_1, \ldots, \psi_k$ belong to Φ. Thus, θ solves the equation $\varphi = \varnothing$ iff $\varphi \notin r_\theta(w)$ for all $w \in \Sigma^*$, i.e., the run does not use any states containing φ. Consequently, if we remove from $\mathcal{A}$ all states containing φ, then we obtain an automaton whose runs are in a 1–1-correspondence with the solutions of $\varphi = \varnothing$. Let us call the resulting looping tree automaton $\mathcal{A}_\varphi$. Obviously, the size of $\mathcal{A}_\varphi$ is exponential in the size of the input system of language (dis)equations, and this automaton can be constructed in exponential time. To decide solvability of the equation $\varphi = \varnothing$ it is enough to test whether $\mathcal{A}_\varphi$ has a run, which can be done using the (linear-time) emptiness test for looping tree automata.

However, some of the runs of $\mathcal{A}_\varphi$ may correspond to substitutions that do not solve the disequations. If θ solves the disequation $\psi_i \neq \varnothing$, then there is a $w \in \Sigma^*$ such that $w \in \theta(\psi_i)$, which is equivalent to $\psi_i \in r_\theta(w)$.

Lemma 2. *A run r of $\mathcal{A}_\varphi$ corresponds to a solution of the whole system (1) iff for every $i, 1 \leq i \leq k$, there is a word $w \in \Sigma^*$ such that $\psi_i \in r_\theta(w)$.*

If we view the indices $1, \ldots, k$ as colors and assign to each state q of $\mathcal{A}_\varphi$ the color set $\kappa(q) := \{i \mid \psi_i \in q\}$, then the condition in the lemma can be reformulated as follows: we are looking for runs in which each color occurs in the color set of at least one state. We will show in the next section how one can check whether a run satisfying such an additional "color condition" exists.

Finiteness of a solution can also easily be expressed by a condition on runs. In fact, since we have $w \in \theta(X_i)$ iff $X_i \in r_\theta(w)$, we need to look for runs in which the variables X_i occur only finitely often. Let us call a state q of $\mathcal{A}_\varphi$ a *variable state* if $X_i \in q$ for some $i, 1 \leq i \leq n$.

Lemma 3. *A run r of $\mathcal{A}_\varphi$ corresponds to a* finite *solution of $\varphi = \varnothing$ iff it contains only finitely many variable states, i.e., the set $\{w \in \Sigma^* \mid r(w)$ is a variable state$\}$ is finite.*

3 Looping Tree Automata with Colors

In this section, we first introduce a new type of automata that can express the "color condition" caused by disequations, and then analyze the complexity of the non-emptiness problem for these automata.

Definition 1. *A* looping tree automaton with colors *is of the form $\mathcal{A} = (Q, \Gamma, Q_0, \Delta, K, \kappa)$, where $\mathcal{A} = (Q, \Gamma, Q_0, \Delta)$ is a looping tree automaton, K is a finite set (of colors), and $\kappa : Q \to 2^K$ assigns to every state q a set of colors $\kappa(q) \subseteq K$.*

A run of $\mathcal{A} = (Q, \Gamma, Q_0, \Delta, K, \kappa)$ on a tree t is a run of the underlying looping tree automaton (Q, Γ, Q_0, Δ) on t. The set $\kappa(r)$ of colors of the run *r is defined as*

$$\kappa(r) := \{\nu \in K \mid \text{there is a node } v \text{ in } t \text{ with } \nu \in \kappa(r(v))\}.$$

The run r satisfies the color condition *if $K = \kappa(r)$. The tree t is* accepted by the looping tree automaton with colors *$\mathcal{A}$ if there is a run of $\mathcal{A}$ on t that satisfies the color condition. The* language *$L(\mathcal{A})$* accepted by the looping tree automaton with colors *$\mathcal{A}$ is the set of all trees accepted by $\mathcal{A}$.*

3.1 Decidability of the Emptiness Problem

In order to show decidability of the non-emptiness problem for looping tree automata with colors, we reduce it to the non-emptiness problem for Büchi tree automata. A *Büchi tree automaton* $\mathcal{A} = (Q, \Gamma, Q_0, \Delta, F)$ is a looping tree automaton that additionally is equipped with a set F of final states. A run r of this automaton on a tree t satisfies the *Büchi acceptance condition* if, on every infinite path through the tree, infinitely many nodes are labeled with final states. The tree t is *accepted* by the Büchi tree automaton $\mathcal{A}$ if there is a run of $\mathcal{A}$ on t that satisfies the Büchi acceptance condition. Again, the *language* $L(\mathcal{A})$ *accepted by the Büchi tree automaton* $\mathcal{A}$ is the set of all trees accepted by $\mathcal{A}$. It is well-known that the emptiness problem for Büchi tree automata is decidable in quadratic time [18].

Let $\mathcal{A} = (Q, \Gamma, Q_0, \Delta, K, \kappa)$ be a looping tree automaton with colors. The corresponding Büchi tree automaton $\mathcal{B}_\mathcal{A} = (Q', \Gamma, Q_0', \Delta', F)$ is defined as follows:

- $Q' := Q \times 2^K$;
- $Q_0' := \{(q, K) \mid q \in Q_0\}$;
- for $q \in Q$, $L \subseteq K$, and $f \in \Gamma$ of arity k we define

$$\Delta'((q, L), f) := \{((q_1, L_1), \ldots, (q_k, L_k)) \mid (q_1, \ldots, q_k) \in \Delta(q, f), L \setminus \kappa(q) \text{ is the union of disjoint sets } L_1, \ldots, L_k\};$$

- $F := Q \times \{\varnothing\}$.

The automaton $\mathcal{B}_\mathcal{A}$ simulates $\mathcal{A}$ in the first components of its states. The second component guesses in which subtree the still required colors are to be found. The Büchi acceptance condition ensures that only runs where these guesses are correct are accepting runs.

Proposition 1. $L(\mathcal{A}) = L(\mathcal{B}_\mathcal{A})$.

Proof. First, assume that r is a run of $\mathcal{A}$ on t that satisfies the color condition, i.e., $\kappa(r) = K$. For each color $\nu \in K$, select a node v_ν of t such that $\nu \in \kappa(r(v_\nu))$ and v_ν has minimal distance from the root, i.e., no node u in t strictly above v_ν satisfies $\nu \in \kappa(r(u))$. We now construct a run of $\mathcal{B}_\mathcal{A}$ on t by adding to r the second components of the states of $\mathcal{B}_\mathcal{A}$. Consider an arbitrary node v in t. We assign to this node the color set

$$\lambda(v) := \{\nu \in K \mid v_\nu = v \text{ or } v_\nu \text{ lies below } v\}.$$

The mapping r' from the nodes of t to the states of $\mathcal{B}_\mathcal{A}$ is defined as $r'(v) = (r(v), \lambda(v))$. We claim that this mapping is a run of $\mathcal{B}_\mathcal{A}$ on t that satisfies the Büchi acceptance condition.

To show that r' is indeed a run of $\mathcal{B}_\mathcal{A}$, consider an arbitrary node v of t. Let $v_1, \ldots, v_k$ be the successor nodes of v. We must show that $((r(v), \lambda(v)), t(v)) \to ((r(v_1), \lambda(v_1)), \ldots, (r(v_k), \lambda(v_k)))$ is a valid transition of $\mathcal{B}_\mathcal{A}$. Since r is a run of $\mathcal{A}$, we have $(r(v_1), \ldots, r(v_k)) \in \Delta(r(v), t(v))$, and thus it is sufficient to show

that $\lambda(v) \setminus \kappa(r(v))$ is the disjoint union of $\lambda(v_1), \ldots, \lambda(v_k)$. Pairwise disjointness of the sets $\lambda(v_1), \ldots, \lambda(v_k)$ is an immediate consequence of the fact that we have chosen only one node v_ν for each color ν, and such a node can belong only to one of the successor subtrees of v. To show that

$$\lambda(v) \setminus \kappa(r(v)) = \lambda(v_1) \cup \ldots \cup \lambda(v_k),$$

first observe that $\nu \in \lambda(v_i)$ means that $v_\nu = v_i$ or v_ν lies below v_i. Thus, v_ν lies below v, which shows that $\nu \in \lambda(v)$. Since v_ν was chosen so that it has minimal distance from the root, $\nu \in \kappa(r(v))$ is not possible. Thus, we have shown that $\nu \in \lambda(v_i)$ implies $\nu \in \lambda(v) \setminus \kappa(r(v))$. Conversely, assume that $\nu \in \lambda(v) \setminus \kappa(r(v))$. Then $\nu \in \lambda(v)$ means that $v_\nu = v$ or v_ν lies below v. However, $\nu \notin \kappa(r(v))$ shows that the first option is not possible. Consequently, v_ν belongs to one of the subtrees below v, which yields $\nu \in \lambda(v_i)$ for some $i, 1 \leq i \leq k$.

To show that r' satisfies the Büchi acceptance condition, consider the maximal distance of the color nodes v_ν for $\nu \in K$ from the root. Since K is finite, this maximal distance is a well-defined natural number d. Any node v that has a larger distance from the root than d cannot be equal to or have below itself any of the color nodes. Consequently, $\lambda(v) = \varnothing$. This shows that, in any infinite path in t, infinitely many nodes are labeled by r' with a state of $\mathcal{B}_\mathcal{A}$ whose second component is $\varnothing$. Since these are exactly the final states of $\mathcal{B}_\mathcal{A}$, this shows that r' satisfies the Büchi acceptance condition. Thus, we have shown that any tree accepted by $\mathcal{A}$ is also accepted by $\mathcal{B}_\mathcal{A}$, i.e., $L(\mathcal{A}) \subseteq L(\mathcal{B}_\mathcal{A})$.

To show that the inclusion in the other direction also holds, assume that r' is a run of $\mathcal{B}_\mathcal{A}$ on t that satisfies the Büchi acceptance condition. Let r be the mapping from the nodes of t to Q that is obtained from r' by disregarding the second components of states, i.e., if $r'(v) = (q, L)$, then $r(v) = q$. Obviously, r is a run of $\mathcal{A}$. It remains to show that it satisfies the color condition. Assume that there is a color $\nu \in K$ that does not occur in $\kappa(r)$. We claim that this implies that there is an infinite path in t satisfying the following property: $(*)$ for any node v in this path, the second component of $r'(v)$ contains ν. Since this would imply that r' does not satisfy the Büchi acceptance condition, this then shows that such a color cannot exist, i.e., $K = \kappa(r)$.

To show the existence of an infinite path satisfying property $(*)$, it is sufficient to show the following: if v is a node in t such that the second component L of $r'(v)$ contains ν, then there is a successor node v_i of v such that the second component L_i of $r'(v_i)$ contains ν. The existence of such a successor node is an immediate consequence of the definition of the transition relation of $\mathcal{B}_\mathcal{A}$ and the fact that ν cannot be an element of $\kappa(r(v))$ since we have assumed $\nu \notin \kappa(r)$. □

As an immediate consequence of this proposition we have that the non-emptiness problem for looping tree automata with colors is decidable: given a looping tree automaton with colors $\mathcal{A}$, we can construct $\mathcal{B}_\mathcal{A}$, and then use the quadratic non-emptiness test for Büchi automata. Regarding the complexity of this decision procedure, we can observe that the size of $\mathcal{B}_\mathcal{A}$ is polynomial in the number of states of $\mathcal{A}$, but exponential in the number of colors.

Theorem 1. *The non-emptiness problem for looping tree automata with colors can be decided in time polynomial in the number of states, but exponential in the number of colors.*

The non-emptiness for looping tree automata with colors can actually also be reduced to the one for looping tree automata without colors. However, this reduction is not language-preserving, but only emptiness-preserving. In fact, it is easy to show that looping tree automata with colors are more expressive than looping tree automata (see [6] for proofs of these results).

3.2 The Exact Complexity of the Emptiness Problem

If we consider the complexity of the emptiness test described in the previous subsection w.r.t. the overall size of the input automaton, then the test yields an ExpTime upper bound for the emptiness problem. In this section, we show that the problem is actually NP-complete.

We show *NP-hardness of the non-emptiness problem for looping tree automata with colors* by a simple reduction from SAT, the satisfiability problem for sets of clauses in propositional logic. Let $\mathcal{P} = \{p_1, \ldots, p_n\}$ be a set of propositional variables, and $\mathcal{L} = \mathcal{P} \cup \{\neg p_1, \ldots, \neg p_n\}$ the corresponding set of literals. Recall that a clause c is a set of literals $\{\ell_1, \ldots, \ell_m\}$, which stands for the disjunction $\ell_1 \vee \ldots \vee \ell_m$ of these literals. A set of clauses $\mathcal{C} = \{c_1, \ldots, c_p\}$ is read conjunctively, i.e., a propositional valuation satisfies $\mathcal{C}$ iff it satisfies all clauses in $\mathcal{C}$. Given a set of clauses $\mathcal{C} = \{c_1, \ldots, c_p\}$ built using literals from $\mathcal{L} = \mathcal{P} \cup \{\neg p_1, \ldots, \neg p_n\}$, we define the corresponding looping tree automaton with colors $\mathcal{A}_\mathcal{C} = (Q, \Gamma, Q_0, \Delta, K, \kappa)$ as follows:

- $\Gamma := \{f\}$ where f has arity 1;
- $Q := \mathcal{L} \cup \{q_{\text{loop}}\}$;
- $Q_0 := \{p_1, \neg p_1\}$;
- for $1 \leq i < n$ and $\ell \in \{p_i, \neg p_i\}$ we define $\Delta(\ell, f) := \{p_{i+1}, \neg p_{i+1}\}$;
- for $\ell \in \{p_n, \neg p_n, q_{\text{loop}}\}$ we define $\Delta(\ell, f) := \{q_{\text{loop}}\}$;
- $K := \mathcal{C}$;
- $\kappa(\ell) := \{c \in \mathcal{C} \mid \ell \in c\}$ for $\ell \in \mathcal{L}$ and $\kappa(q_{\text{loop}}) := \varnothing$.

Obviously, the size of $\mathcal{A}_\mathcal{C}$ is polynomial in the size of $\mathcal{L}$ and $\mathcal{C}$.

A run r of $\mathcal{A}_\mathcal{C}$ on the unique infinite tree over Γ contains, for every $i, 1 \leq i \leq n$, either p_i or $\neg p_i$, i.e., it determines a propositional valuation. If this run satisfies the color condition, then every clause c belongs to $\kappa(r)$, i.e., there is a literal ℓ that occurs in r (i.e., ℓ is true in the valuation determined by r) and that is contained in c. This shows that runs satisfying the color condition determine valuations that satisfy all clauses in $\mathcal{C}$. Conversely, a propositional valuation determines a unique run r, by choosing for every i the literal that is true in this valuation. If the valuation satisfies $\mathcal{C}$, then for each clause c one of its literals is true, and thus occurs in r. Consequently, each clause occurs in the color set $\kappa(r)$, which shows that r satisfies the color condition. Therefore, the clause set $\mathcal{C}$ is satisfiable iff $L(\mathcal{A}_\mathcal{C}) \neq \varnothing$.

Since the satisfiability problem for sets of propositional clauses is NP-hard, this shows that the same is true for the non-emptiness problem for looping tree automata with colors.

Proposition 2. *The non-emptiness problem for looping tree automata with colors is NP-hard.*

To show that the *non-emptiness problem for looping tree automata with colors is in NP* we consider the Büchi tree automaton constructed in the previous subsection. But first, we eliminate all states in the given automaton that do not occur in any run: these states can be identified in polynomial time using the emptiness test for looping tree automata [7]. The resulting automaton has the same set of runs on any tree, and thus also accepts the same language.

Let us now assume that all states of the looping tree automaton with colors $\mathcal{A} = (Q, \Gamma, Q_0, \Delta, K, \kappa)$ occur in some run, and that the set of colors K is non-empty.[4] Let $\mathcal{B}_\mathcal{A} = (Q', \Gamma, Q'_0, \Delta', F)$ be the Büchi automaton constructed from $\mathcal{A}$ in the previous section. Call a transition $((q, L), f) \to ((q_1, L_1), \ldots, (q_k, L_k))$ *decreasing* if $|L| > |L_i|$ holds for all $i, 1 \leq i \leq k$. Otherwise, the transition is called *non-decreasing*. The following lemma is an easy consequence of the definition of Δ'.

Lemma 4. *If* $((q, L), f) \to ((q_1, L_1), \ldots, (q_k, L_k))$ *is non-decreasing, then* $\kappa(q) \cap L = \varnothing$ *and there is an* $i, 1 \leq i \leq k$, *such that* $L_i = L$ *and* $L_j = \varnothing$ *for all* $j \neq i$.

Now, assume that r is a run of $\mathcal{B}_\mathcal{A}$ satisfying the Büchi acceptance condition. This run starts with an initial state $(q_0, K) \in Q'_0 = Q_0 \times \{K\}$. If the first transition that is applied is a non-decreasing transition, then there is exactly one successor node n_1 of the root to which r assigns a state with $K \neq \varnothing$ as second component, whereas all the other nodes are assigned states with empty second components (i.e., final states). If another non-decreasing transition is applied to n_1, then there is exactly one successor node of n_1 to which r assigns a state with $K \neq \varnothing$ as second component, etc. Since r satisfies the Büchi acceptance condition, after a finite number of non-decreasing steps we reach a node v to which a decreasing transition is applied. Let this decreasing transition be of the form $((q, K), \_) \to ((q_1, L_1), \ldots, (q_k, L_k))$ (where here and in the following, the alphabet symbol from Γ is irrelevant). Since the transition is decreasing, we have $|K| > |L_i|$ for all $i, 1 \leq i \leq k$. Let $v_1, \ldots, v_k$ be the successor nodes of v, and consider all v_i such that $L_i \neq \varnothing$. We can now apply the same analysis as for the root and K to the nodes v_i and $L_i \neq \varnothing$, i.e., we follow a chain of non-decreasing transitions that reproduce L_i until we find the next decreasing transition. This can be done until all color sets are empty. Basically, this construction yields a finite tree of decreasing transitions satisfying certain easy to check properties (see Definition 2 below). Our NP-algorithm guesses such a tree and checks whether the required properties are satisfied. Before we can formally define the relevant properties of this tree, we need to introduce one more notation.

[4] If $K = \varnothing$, then $\mathcal{A}$ is a normal looping tree automaton, for which the non-emptiness problem is decidable in polynomial time.

Let $L \subseteq K$ be a non-empty set of colors and let q, q' be states in Q. We say that q' is *directly L-reachable* from q if there is a transition $(q, \_) \to (q_1, \ldots, q_k)$ in Δ such that $q' = q_i$ for some $i, 1 \leq i \leq k$, and $L \cap \kappa(q) = \varnothing$. Note that this implies that there is a non-decreasing transition $((q, L), \_) \to ((q_1, L_1), \ldots, (q_k, L_k))$ with $L_i = L$ and $L_j = \varnothing$ for $j \neq i$ in the transition relation Δ' of $\mathcal{B}_\mathcal{A}$. We say that q' is *L-reachable* from q if there is a sequence of states $p_0, \ldots, p_\ell$ ($\ell \geq 0$) such that $q = p_0$, $q' = p_\ell$, and p_{i+1} is directly L-reachable from p_i for all $i, 0 \leq i < \ell$.

Definition 2. *Given a looping tree automaton with colors $\mathcal{A}$ and the corresponding Büchi tree automaton $\mathcal{B}_\mathcal{A}$, a* dt-tree *for $\mathcal{B}_\mathcal{A}$ is a finite tree T whose nodes are decreasing transitions of $\mathcal{B}_\mathcal{A}$ such that the following properties are satisfied:*

- *the root of T is of the form $((q, K), \_) \to ((q_1, L_1), \ldots, (q_k, L_k))$ such that q is K-reachable from some initial state of $\mathcal{A}$;*
- *if $((q, L), \_) \to ((q_1, L_1), \ldots, (q_k, L_k))$ is a node in T and $i_1, \ldots, i_\ell$ are all the indices i with $L_i \neq \varnothing$, then this node has ℓ successor nodes of the form $((q'_{i_j}, L_{i_j}), \_) \to \cdots$ such that q'_{i_j} is L_{i_j}-reachable from q_{i_j} for $j = 1, \ldots, \ell$.*

Note that the leaves of a dt-tree are labeled with transitions $((q, L), \_) \to ((q_1, L_1), \ldots, (q_k, L_k))$ for which $L_1 = \ldots = L_k = \varnothing$.

Lemma 5. *We have $L(\mathcal{B}_\mathcal{A}) \neq \varnothing$ iff there exists a dt-tree for $\mathcal{B}_\mathcal{A}$.*

The lemma, whose proof can be found in [6], shows that it is enough to design an algorithm that checks for the existence of a dt-tree. For this to be possible in non-deterministic polynomial time, we need to know that the size of dt-trees is polynomial in the size of $\mathcal{A}$. We can actually show the following linear bound in the number of colors.

Lemma 6. *The number of nodes of a dt-tree is bounded by $2 \cdot |K|$.*

Proof. We call a decreasing transition $((q, L), \_) \to ((q_1, L_1), \ldots, (q_k, L_k))$ *removing* if $L \cap \kappa(q) \neq \varnothing$ and *branching* otherwise. Note that, for a branching transition $((q, L), \_) \to ((q_1, L_1), \ldots, (q_k, L_k))$, there must be indices $i \neq j$ such that L_i and L_j are non-empty.

In a dt-tree, for every color there is exactly one transition removing it, and every removing transition removes at least one color. Consequently, a dt-tree can contain at most $|K|$ removing transitions. Since decreasing transitions that are leaves in a dt-tree are necessarily removing, this also shows that the number of leaves of a dt-tree is bounded by $|K|$.

Any branching transition increases the number of leaves by at least one, which shows that a dt-tree can contain at most $|K| - 1$ branching transitions. Since every decreasing transition is either removing or branching, this completes the proof of the lemma. □

Together with Lemma 5, this lemma yields the desired NP upper bound (see [6] for more details). Given the NP-hardness result of Proposition 2, we thus have determined the exact worst-case complexity of the non-emptiness problem.

Theorem 2. *The non-emptiness problem for looping tree automata with colors is NP-complete.*

4 Applying the Results

We will first show that the results obtained so far allow us to determine the exact complexity of (finite) solvability of finite systems of language (dis)equations with one-sided concatenation.

Proposition 3. *For a given finite system of language (dis)equations of the form (1), solvability and finite solvability are decidable in ExpTime.*

Proof. Let $\mathcal{A}_\phi = (Q, \Gamma, Q_0, \Delta)$ be the looping tree automaton constructed from the system (1) in Section 2.2, and define $K := \{1, \ldots, k\}$ and $\kappa(q) := \{i \in K \mid \psi_i \in q\}$ for all $q \in Q$. According to Lemma 2, the system (1) has a solution iff the looping tree automaton with colors $\mathcal{A} = (Q, \Gamma, Q_0, \Delta, K, \kappa)$ has a run satisfying the color condition, i.e., accepts a non-empty language. As shown in the previous section, from $\mathcal{A}$ we can construct a Büchi automaton $\mathcal{B}_\mathcal{A}$ such that $L(\mathcal{A}) = L(\mathcal{B}_\mathcal{A})$ and the size of $\mathcal{B}_\mathcal{A}$ is polynomial in the number of states, but exponential in the number of colors of $\mathcal{A}$. Since the number of states of $\mathcal{A}$ is exponential in the size of the system (1), but the number of colors is linear in that size, the size of $\mathcal{B}_\mathcal{A}$ is exponential in the size of the system (1). As the emptiness problem for Büchi automata can be solved in polynomial time, this yields the desired ExpTime upper bound for solvability.

For finite solvability, we also must take the condition formulated in Lemma 3 into account, i.e., we are looking for runs of $\mathcal{B}_\mathcal{A}$ such that states of $\mathcal{B}_\mathcal{A}$ whose first components are variable states of $\mathcal{A}$ occur only finitely often. This condition can easily be expressed by modifying the Büchi automaton $\mathcal{B}_\mathcal{A}$, as described in a more general setting in the proof of the next lemma. Since the new Büchi automaton constructed in that proof is linear in the size of the original automaton, this yields the desired ExpTime upper bound for finite solvability. □

Lemma 7. *Let $\mathcal{B} = (Q, \Gamma, Q_0, \Delta, F)$ be a Büchi automaton and $P \subseteq Q$. Then we can construct in linear time a Büchi automaton $\mathcal{B}' = (Q', \Gamma, Q'_0, \Delta', F')$ such that $L(\mathcal{B}') = \{t \mid$ there is a run of $\mathcal{B}$ on t that contains only finitely many states from $P\}$.*

Proof. We define $Q' := Q \times \{1\} \cup (Q \setminus P) \times \{0\}$, $Q'_0 = Q_0 \times \{1\}$, $F' := (F \setminus P) \times \{0\}$, and

$$\Delta'((q,1),\gamma) := \{((q_1,i_1),\ldots,(q_k,i_k)) \mid (q_1,\ldots,q_k) \in \Delta(q,\gamma), \; i_j = 1 \text{ if } q_j \in P, \; i_j \in \{0,1\} \text{ if } q_j \in Q \setminus P\},$$

$$\Delta'((q,0),\gamma) := \{((q_1,0),\ldots,(q_k,0)) \mid (q_1,\ldots,q_k) \in \Delta(q,\gamma), \; q_1,\ldots,q_k \notin P\}.$$

Basically, this Büchi automaton guesses (by decreasing the second component of a state to 0) that from now on only states from $Q \setminus P$ will be seen. In fact, once the second component is 0, it stays 0 in all successor states, and only states from $Q \setminus P$ are paired with 0. Since F' contains only states with second component 0, this enforces that on every path eventually only states with second component 0

(and thus first component in $Q \setminus P$) occur. By König's lemma, this implies that a run of $\mathcal{B}'$ satisfying the Büchi acceptance condition contains only finitely many states with second component 1, and thus only finitely many states whose first component belongs to P. □

Since (finite) solvability of language equations that are simpler than the ones considered here are ExpTime-hard [4,3], we thus have determined the exact complexity of (finite) solvability of our systems of language (dis)equations.

Theorem 3. *The problems of deciding solvability and finite solvability of finite systems of language (dis)equations of the form (1) are ExpTime-complete.*

4.1 Disunification in $\mathcal{FL}_0$

Unification in the description logic $\mathcal{FL}_0$ has been investigated in detail in [4]. In particular, it is shown there that solvability of $\mathcal{FL}_0$-unification problems is an ExpTime-complete problem. The ExpTime upper bound is based on a reduction to finite solvability of a restricted form of language equations with one-sided concatenation. In this subsection, we use Theorem 3 to show that this upper bound also holds for $\mathcal{FL}_0$-disunification problems.

Due to the space restriction, we cannot recall syntax and semantics of the description logic (DL) $\mathcal{FL}_0$ and the exact definition of unification in $\mathcal{FL}_0$ here (they can be found in [4] and in [6]). For our purposes, it is enough to recall on an abstract level how such unification problems are translated into language equations. The syntax of $\mathcal{FL}_0$ determines what kind of *concept terms* one can build from given finite sets N_C of concept names and N_R of role names, and the semantics is based on interpretations $\mathcal{I}$, which assign sets $C^{\mathcal{I}}$ to concept terms C. Two concept terms C, D are equivalent ($C \equiv D$) iff $C^{\mathcal{I}} = D^{\mathcal{I}}$ for every interpretation $\mathcal{I}$. An $\mathcal{FL}_0$-unification problem is a finite set of equivalences $C \equiv^? D$, where C, D are $\mathcal{FL}_0$-concept patterns, i.e., $\mathcal{FL}_0$-concept terms with variables. Substitutions replace concept variables by concept terms. A *unifier* σ of a given unification problem is a substitution that solves all its equivalences, i.e., satisfies $\sigma(C) \equiv \sigma(D)$ for all equivalences $C \equiv^? D$ in the problem.

As shown in [4], every unification problem can be transformed in linear time into an equivalent one consisting of a single equation $C_0 \equiv^? D_0$. This equation can then be transformed into a system of language equations, with one language equation $E_{C_0,D_0}(A)$ for every concept name $A \in N_C$.[5] The alphabet of these language equations is the set N_R of role names, and the variables occurring in $E_{C_0,D_0}(A)$ are renamed copies X_A of the variables X occurring in the patterns C_0, D_0. In particular, this implies that the equations $E_{C_0,D_0}(A)$ do not share variables, and thus can be solved independently from each other.

[5] These equations are basically language equations with one-sided concatenation, as introduced in the present paper, but with concatenation of constants from the left rather than from the right. However, one can transform them into equations with concatenation of constants from the right, by reversing all concatenations [4]. We assume from now on that the equations $E_{C_0,D_0}(A)$ are already of this form.

Lemma 8 ([4]). *The equivalence $C_0 \equiv^? D_0$ has a unifier iff for all concept names $A \in N_C$, the language equations $E_{C_0,D_0}(A)$ have finite solutions.*

For disunification, we additionally consider finitely many disequivalences $C_i \not\equiv^? D_i$ for $i = 1, \ldots, k$. A substitution σ solves such a disequivalence iff $\sigma(C_i) \not\equiv \sigma(D_i)$. Disequivalences can now be translated into language disequations $D_{C_i,D_i}(A)$, which are defined like $E_{C_i,D_i}(A)$, with the only difference that equality $=$ is replaced by inequality $\neq$. For a disequivalence it is enough to solve one of the associated language disequations. The following can be shown by a simple adaptation of the proof of Lemma 8 in [4].

Lemma 9. *The disunification problem $\{C_0 \equiv^? D_0, C_1 \not\equiv^? D_1, \ldots, C_k \not\equiv^? D_k\}$ has a solution iff for every $A \in N_C$, there is a substitution θ_A such that*

- *$\theta_A(X_A)$ is finite for all $A \in N_C$ and all variables X occurring in the problem;*
- *θ_A solves the language equation $E_{C_0,D_0}(A)$ for all $A \in N_C$;*
- *for every index $i \in \{1, \ldots, k\}$ there is a concept name $A \in N_C$ such that θ_A solves the language disequation $D_{C_i,D_i}(A)$.*

In order to take care of the last condition of the lemma, we consider functions $f : \{1, \ldots, k\} \to N_C$. Given such a function f, we define, for each $A \in N_C$, the system of language (dis)equations $DE_f(A)$ as

$$DE_f(A) := \{E_{C_0,D_0}(A)\} \cup \{D_{C_i,D_i}(A) \mid f(i) = A\}.$$

The following theorem is then an immediate consequence of Lemma 9.

Theorem 4. *The disunification problem $\{C_0 \equiv^? D_0, C_1 \not\equiv^? D_1, \ldots, C_k \not\equiv^? D_k\}$ has a solution iff there is a function $f : \{1, \ldots, k\} \to N_C$ such that, for every concept names $A \in N_C$, the system of language (dis)equations $DE_f(A)$ has a finite solution.*

Since there are exponentially many functions $f : \{1, \ldots, k\} \to N_C$ and finite solvability of each system of language (dis)equations $DE_f(A)$ can be tested in exponential time by Theorem 3, this yields an overall exponential time complexity. ExpTime-hardness already holds for the special case of unification.

Corollary 1. *Solvability of $\mathcal{FL}_0$-disunification problems is ExpTime-complete.*

4.2 Monadic Set Constraints

As already mentioned in [3] and [5], there is a close connection between language equations with one-sided concatenation and monadic set constraints, i.e., set constraints where all function symbols are unary or nullary. For the case of set constraints without negation (i.e., where only inclusions between sets are allowed), it has been known for a long time [1] that the unrestricted case is NExpTime-complete and the monadic one (with at least two unary symbols and at least one nullary symbol) is ExpTime-complete. For the case of set constraints with negation (i.e., where inclusions and negated inclusions between sets are allowed),

NExpTime-completeness for the unrestricted case has been shown by several authors [8,17,2], but to the best of our knowledge, the monadic case has not been investigated.

Because of the space constraints, we cannot formally introduce monadic set constraints and their translation into language equations here, but it should be noted that this translation is quite obvious (see [6] for details). In fact, nullary and unary function symbols correspond to the elements of the alphabet and application of unary functions to concatenation. To be more precise, using postfix notation, the term $f_1(f_2(\cdots f_k(a)\cdots))$ can be written as a word $af_k \dots f_1$. This way, sets of terms can be translated into sets of words, where each word starts with a constant and is followed by a (possibly empty) sequence of unary function symbols. Since they basically have the same syntax rules, positive set constraints can be translated into language equations and negative set constraints into language disequations, so that solutions of the set constraints translate into solutions of the language (dis)equations, as sketched above. In order to translate solutions of the languages (dis)equations back to solutions of the sets constraints, one must make sure that every word occurring in such a solution starts with a constant and is followed by a sequence of unary function symbols. This restriction can easily be enforced by adding appropriate equations. This shows that solvability of finite systems of monadic set constraints with negation can be reduced in polynomial time to solvability of finite systems of language (dis)equations. Since Theorem 3 states an ExpTime upper bound also for solvability, this yields an ExpTime upper bound for solvability of monadic set constraints with negation. ExpTime-hardness already holds for the special case of monadic set constraints without negation [1].

Corollary 2. *Solvability of monadic set constraints with negation is ExpTime-complete.*

5 Conclusion

We have shown that solvability and finite solvability of systems of language (dis)equations are ExpTime-complete, in contrast to their undecidability (Σ_2^0-completeness) in the case of unrestricted concatenation [13]. We have used these results to obtain new complexity results for solving monadic set constraints with negation, and for disunification problems in the DL $\mathcal{FL}_0$. As a tool, we have introduced looping tree automata with colors. Though the results of Section 3 show that a direct reduction to the emptiness problem for Büchi tree automata would be possible, using looping tree automata with colors as intermediate formalism makes the presentation much clearer and easier to comprehend. In addition, we believe that these automata may be of interest also for other applications in logic.

References

1. Aiken, A., Kozen, D., Vardi, M.Y., Wimmers, E.L.: The Complexity of Set Constraints. In: Meinke, K., Börger, E., Gurevich, Y. (eds.) CSL 1993. LNCS, vol. 832, pp. 1–17. Springer, Heidelberg (1994)

2. Aiken, A., Kozen, D., Wimmers, E.L.: Decidability of systems of set constraints with negative constraints. Information and Computation 122(1), 30–44 (1995)
3. Baader, F., Küsters, R.: Unification in a Description Logic with Transitive Closure of Roles. In: Nieuwenhuis, R., Voronkov, A. (eds.) LPAR 2001. LNCS (LNAI), vol. 2250, pp. 217–232. Springer, Heidelberg (2001)
4. Baader, F., Narendran, P.: Unification of concept terms in description logic. Journal of Symbolic Computation 31, 277–305 (2001)
5. Baader, F., Okhotin, A.: On Language Equations with One-sided Concatenation., LTCS-Report LTCS-06-01, Chair for Automata Theory, Institute for Theoretical Computer Science, TU Dresden, A short version has been published in the Proceedings of the 20th International Workshop on Unification, UNIF 2006 (2006), http://lat.inf.tu-dresden.de/research/reports.html
6. Baader, F., Okhotin, A.: Solving Language Equations and Disequations Using Looping Tree Automata with Colors, LTCS-Report LTCS-12-01, Chair for Automata Theory, Institute for Theoretical Computer Science, TU Dresden (2012), http://lat.inf.tu-dresden.de/research/reports.html
7. Baader, F., Tobies, S.: The Inverse Method Implements the Automata Approach for Modal Satisfiability. In: Goré, R.P., Leitsch, A., Nipkow, T. (eds.) IJCAR 2001. LNCS (LNAI), vol. 2083, pp. 92–106. Springer, Heidelberg (2001)
8. Charatonik, W., Pacholski, L.: Negative set constraints with equality. In: Logic in Computer Science, LICS 1994, Paris, France, pp. 128–136 (1994)
9. Ginsburg, S., Rice, H.G.: Two families of languages related to ALGOL. J. of the ACM 9, 350–371 (1962)
10. Jeż, A., Okhotin, A.: On the Computational Completeness of Equations over Sets of Natural Numbers. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part II. LNCS, vol. 5126, pp. 63–74. Springer, Heidelberg (2008)
11. Kunc, M.: What Do We Know About Language Equations? In: Harju, T., Karhumäki, J., Lepistö, A. (eds.) DLT 2007. LNCS, vol. 4588, pp. 23–27. Springer, Heidelberg (2007)
12. Lehtinen, T., Okhotin, A.: On Language Equations $XXK = XXL$ and $XM = N$ over a Unary Alphabet. In: Gao, Y., Lu, H., Seki, S., Yu, S. (eds.) DLT 2010. LNCS, vol. 6224, pp. 291–302. Springer, Heidelberg (2010)
13. Okhotin, A.: Strict Language Inequalities and Their Decision Problems. In: Jedrzejowicz, J., Szepietowski, A. (eds.) MFCS 2005. LNCS, vol. 3618, pp. 708–719. Springer, Heidelberg (2005)
14. Pacholski, L., Podelski, A.: Set Constraints: A Pearl in Research on Constraints. In: Smolka, G. (ed.) CP 1997. LNCS, vol. 1330, pp. 549–562. Springer, Heidelberg (1997)
15. Parikh, R., Chandra, A., Halpern, J., Meyer, A.: Equations between regular terms and an application to process logic. SIAM Journal on Computing 14(4), 935–942 (1985)
16. Rabin, M.O.: Decidability of second-order theories and automata on infinite trees. Transactions of the American Mathematical Society 141, 1–35 (1969)
17. Stefánsson, K.: Systems of set constraints with negative constraints are NEXP-TIME-complete. In: Logic in Computer Science, LICS 1994, Paris, France, pp. 137–141 (1994)
18. Vardi, M.Y., Wolper, P.: Automata-theoretic techniques for modal logics of programs. Journal of Computer and System Sciences 32, 183–221 (1986)

Dual-Priced Modal Transition Systems with Time Durations*

Nikola Beneš[2,**], Jan Křetínský[2,3,***], Kim Guldstrand Larsen[1], Mikael H. Møller[1], and Jiří Srba[1]

[1] Aalborg University, Denmark
[2] Masaryk University, Czech Republic
[3] Technical University München, Germany

Abstract. Modal transition systems are a well-established specification formalism for a high-level modelling of component-based software systems. We present a novel extension of the formalism called modal transition systems with durations where time durations are modelled as controllable or uncontrollable intervals. We further equip the model with two kinds of quantitative aspects: each action has its own running cost per time unit, and actions may require several hardware components of different costs. We ask the question, given a fixed budget for the hardware components, what is the implementation with the cheapest long-run average reward. We give an algorithm for computing such optimal implementations via a reduction to a new extension of mean payoff games with time durations and analyse the complexity of the algorithm.

1 Introduction and Motivating Example

Modal Transition Systems (MTS) is a specification formalism [16,2] that aims at providing a flexible and easy-to-use compositional development methodology for reactive systems. The formalism can be viewed as a fragment of a temporal logic [1,9] that at the same time offers a behavioural compositional semantics with an intuitive notion of process refinement. The formalism of MTS is essentially a labelled transition system that distinguishes two types of labelled transitions: *must* transitions which are required in any refinement of the system, and *may* transitions that are allowed to appear in a refined system but are not required. The refinement of an MTS now essentially consists of iteratively resolving the presence or absence of may transitions in the refined process.

In a recent line of work [15,3], the MTS framework has been extended to allow for the specification of additional constraints on quantitative aspects (e.g. time, power or memory), which are highly relevant in the area of embedded systems. In this paper we continue the pursuit of quantitative extensions of MTS by

* Supported by VKR Center of Excellence MT-LAB.
** The author has been supported by Czech Grant Agency, grant no. GAP202/11/0312.
*** The author is a holder of Brno PhD Talent Financial Aid and is supported by the Czech Science Foundation, grant No. P202/10/1469.

N. Bjørner and A. Voronkov (Eds.): LPAR-18, LNCS 7180, pp. 122–137, 2012.

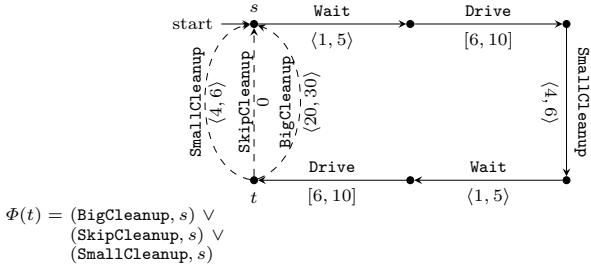
presenting a novel extension of MTS with time durations being modelled as controllable or uncontrollable intervals. We further equip the model with two kinds of quantitative aspects: each action has its own running cost per time unit, and actions may require several hardware components of different costs. Thus, we ask the question, given a fixed budget for the investment into the hardware components, what is the implementation with the cheapest long-run average reward.

Before we give a formal definition of modal transition systems with durations (MTSD) and the dual-price scheme, and provide algorithms for computing optimal implementations, we present a small motivating example.

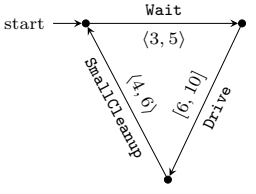
Consider the specification $\mathcal{S}$ in Figure 1a describing the work of a shuttle bus driver. He drives a bus between a hotel and the airport. First, the driver has to `Wait` for the passengers at the hotel. This can take one to five minutes. Since this behaviour is required to be present in all the implementations of this specification, it is drawn as a solid arrow and called a *must* transition. Then the driver has to `Drive` the bus to the airport (this takes six to ten minutes) where he has to do a `SmallCleanup`, then `Wait` before he can `Drive` the bus back to the hotel. When he returns he can do either a `SmallCleanup`, `BigCleanup` or `SkipCleanup` of the bus before he continues. Here we do not require a particular option to be realised in the implementations, hence we only draw the transitions as dashed arrows. As these transitions may or may not be present in the implementations, they are called *may* transitions. However, here the intention is to require at least one option be realised. Hence, we specify this using a propositional formula Φ assigned to the state t over its outgoing transitions as described in [5,6]. After performing one of the actions, the driver starts over again. Note that next time the choice in t may differ.

Observe that there are three types of durations on the transitions. First, there are *controllable* intervals, written in angle brackets. The meaning of e.g. $\langle 1, 5\rangle$ is that in the implementation we can instruct the driver to wait for a fixed number of minutes in the range. Second, there are *uncontrollable* intervals, written in square brackets. The interval $[6, 10]$ on the `Drive` transition means that in the implementation we cannot fix any particular time and the time can vary, say, depending on the traffic and it is chosen nondeterministically by the environment. Third, the degenerated case of a single number, e.g. 0, denotes that the time taken is always constant and given by this number. In particular, a zero duration means that the transition happens instantaneously.

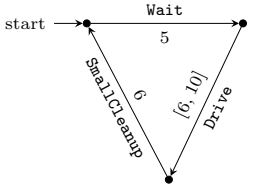
The system $\mathcal{S}_1$ is another specification, a *refinement* of $\mathcal{S}$, where we additionally specify that the driver must do a `SmallCleanup` after each `Drive`. Note that the `Wait` interval has been narrowed. The system $\mathcal{I}_1$ is an implementation of $\mathcal{S}_1$ (and actually also of $\mathcal{S}$) where all controllable time intervals have already been fully resolved to their final single values: the driver must `Wait` for 5 minutes and do the `SmallCleanup` for 6 minutes. Note that uncontrollable intervals remain unresolved in the implementations and the time is chosen by the environment each time the action is performed. This reflects the inherent uncontrollable uncertainty of the timing, e.g. of a traffic.



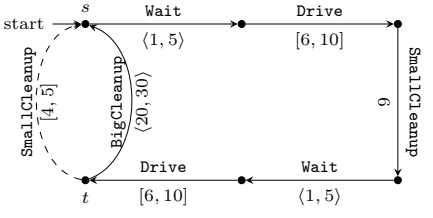
(a) Specification $\mathcal{S}$



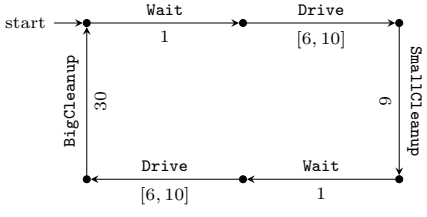
(b) Specification $\mathcal{S}_1$



(c) Implementation $\mathcal{I}_1$



(d) Specification $\mathcal{S}_2$



(e) Implementation $\mathcal{I}_2$

$a \in \Sigma$	$r(a)$
Wait	8
Drive	10
SmallCleanup	6
BigCleanup	7
SkipCleanup	0

$H = \{\texttt{VacuumCleaner}, \texttt{Sponge}\}$

$$\Psi(a) = \begin{cases} \texttt{VacuumCleaner} & \text{if } a = \texttt{BigCleanup} \\ \texttt{Sponge} \vee \texttt{VacuumCleaner} & \text{if } a = \texttt{SmallCleanup} \\ true & \text{otherwise} \end{cases}$$

$h \in H$	$i(h)$
VacuumCleaner	100
Sponge	5

(f) Price Scheme

Fig. 1. Example of Dual-Priced Modal Transition Systems with Time Durations

The system $\mathcal{S}_2$ is yet another specification and again a refinement of $\mathcal{S}$, where the driver can always do a `BigCleanup` in t and possibly there is also an alternative allowed here of a `SmallCleanup`. Notice that both `SmallCleanup` intervals have been restricted and changed to uncontrollable. This means that we give up the control over the duration of this action and if this transition is implemented, its duration will be every time chosen nondeterministically in that range. Finally, $\mathcal{I}_2$ is then an implementation of $\mathcal{S}_2$ and $\mathcal{S}$.

Furthermore, we develop a way to model cost of resources. Each action is assigned a *running price* it costs per time unit, e.g. `Drive` costs 10 each time unit it is being performed as it can be seen in the left table of Figure 1f. In addition, in order to perform an action, some hardware may be needed, e.g. a `VacuumCleaner` for the `BigCleanup` and its price is 100 as can be seen on the right. This *investment price* is paid once only.

Let us now consider the problem of finding an optimal implementation, so that we spend the least possible amount of money (e.g. the pay to the driver) per time unit while conforming to the specification $\mathcal{S}$. We call this problem *the cheapest implementation problem*. The optimal implementation is to buy a vacuum cleaner if one can afford an investment of 100 and do the `BigCleanup` every time as long as possible and `Wait` as shortly as possible. (Note that `BigCleanup` is more costly per time unit than `SmallCleanup` but lasts longer.) This is precisely implemented in $\mathcal{I}_2$ and the (worst-case) average cost per time unit is ≈ 7.97. If one cannot afford the vacuum cleaner but only a sponge, the optimal worst case long run average is then a bit worse and is implemented by doing the `SmallCleanup` as long as possible and `Wait` now as *long* as possible. This is depicted in $\mathcal{I}_1$ and the respective average cost per time unit is ≈ 8.10.

The most related work is [12] where prices are introduced into a class of interface theories and long-run average objectives are discussed. Our work omits the issue of distinguishing input and output actions. Nevertheless, compared to [12], this paper deals with the time durations, the one-shot hardware investment and, most importantly, refinement of specifications. Further, timed automata have also been extended with prices [4] and the long-run average reward has been computed in [10]. However, priced timed automata lack the hardware and any notion of refinement, too.

The paper is organized as follows. We introduce the MTS with the time durations in Section 2 and the dual-price scheme in Section 3. Section 4 presents the main results on the complexity of the cheapest implementation problem. First, we state the complexity of this problem in general and in an important special case and prove the hardness part. The algorithms proving the complexity upper bounds are presented only after introducing an extension of mean payoff games with time durations. These are needed to establish the results but are also interesting on their own as discussed in Section 4.1. Due to space limitations, some of the proofs are in the full version of the paper [7]. We conclude and give some more account on related and future work in Section 5.

2 Modal Transition Systems with Durations

In order to define MTS with durations, we first introduce the notion of *controllable* and *uncontrollable* duration intervals. A controllable interval is a pair $\langle m, n\rangle$ where $m, n \in \mathbb{N}_0$ and $m \leq n$. Similarly, an uncontrollable interval is a pair $[m, n]$ where $m, n \in \mathbb{N}_0$ and $m \leq n$. We denote the set of all controllable intervals by $\mathfrak{I}_\mathsf{c}$, the set of all uncontrollable intervals by $\mathfrak{I}_\mathsf{u}$, and the set of all intervals by $\mathfrak{I} = \mathfrak{I}_\mathsf{c} \cup \mathfrak{I}_\mathsf{u}$. We also write only m to denote the singleton interval $[m, m]$.

Singleton controllable intervals need not be handled separately as there is no semantic difference to the uncontrollable counterpart.

We can now formally define modal transition systems with durations. In what follows, $\mathcal{B}(X)$ denotes the set of propositional logic formulae over the set X of atomic propositions, where we assume the standard connectives $\wedge, \vee, \neg$.

Definition 1 (MTSD). *A* Modal Transition System with Durations *(MTSD) is a tuple* $\mathcal{S} = (S, T, D, \Phi, s_0)$ *where* S *is a set of* states *with the* initial state s_0, $T \subseteq S \times \Sigma \times S$ *is a set of* transitions, $D : T \to \mathfrak{I}$ *is a* duration interval function, *and* $\Phi : S \to \mathcal{B}(\Sigma \times S)$ *is an* obligation function. *We assume that whenever the atomic proposition* (a, t) *occurs in the Boolean formula* $\Phi(s)$ *then also* $(s, a, t) \in T$.

We moreover require that there is no cycle of transitions that allows for zero accumulated duration, i.e. there is no path $s_1 a_1 s_2 a_2 \cdots s_n$ *where* $(s_i, a_i, s_{i+1}) \in T$ *and* $s_n = s_1$ *such that for all* i, *the interval* $D((s_i, a_i, s_{i+1}))$ *is of the form either* $\langle 0, m\rangle$ *or* $[0, m]$ *for some* m.

Note that instead of the basic may and must modalities known from the classical modal transition systems (see e.g. [2]), we use arbitrary boolean formulae over the outgoing transitions of each state in the system as introduced in [6]. This provides a higher generality as the formalism is capable to describe, apart from standard modal transition systems, also more expressive formalisms like disjunctive modal transition systems [17] and transition systems with obligations [5]. See [6] for a more thorough discussion of this formalism.

In the rest of the paper, we adapt the following convention when drawing MTSDs. Whenever a state s is connected with a solid arrow labelled by a to a state s', this means that in any satisfying assignment of the Boolean formula $\Phi(s)$, the atomic proposition (a, s') is always set to true (the transition *must* be present in any refinement of the system). Should this not be the case, we use a dashed arrow instead (meaning that the corresponding transition *may* be present in a refinement of the system but it can be also left out). For example the solid edges in Figure 1a correspond to an implicitly assumed $\Phi(s) = (a, s')$ where (s, a, s') is the (only) outgoing edge from s; in this case we do not explicitly write the obligation function. The three dashed transitions in the figure are optional, though at least one of them has to be preserved during any refinement (a feature that can be modelled for example in disjunctive MTS [17]).

Remark 2. The standard notion of modal transition systems (see e.g. [2]) is obtained under the restriction that the formulae $\Phi(s)$ in any state $s \in S$ have the form $(a_1, s_1) \wedge \ldots \wedge (a_n, s_n)$ where $(s, a_1, s_1), \ldots, (s, a_n, s_n) \in T$. The edges mentioned in such formulae are exactly all must transitions; may transitions are not listed in the formula and hence can be arbitrarily set to true or false.

Let by $T(s) = \{(a, t) \mid (s, a, t) \in T\}$ denote the set of all outgoing transitions from the state $s \in S$. A modal transition system with durations is called an *implementation* if $\Phi(s) = \bigwedge T(s)$ for all $s \in S$ (every allowed transition is also required), and $D(s, a, s') \in \mathcal{I}_{\mathsf{u}}$ for all $(s, a, s') \in T$, i.e. all intervals are uncontrollable, often singletons. Figure 1c shows an example of an implementation,

while Figure 1b is not yet an implementation as it still contains the controllable intervals $\langle 3, 5\rangle$ and $\langle 4, 6\rangle$.

We now define a notion of modal refinement. In order to do that, we first need to define refinement of intervals as a binary relation $\leq \subseteq \mathfrak{I} \times \mathfrak{I}$ such that

- $\langle m', n'\rangle \leq \langle m, n\rangle$ whenever $m' \geq m$ and $n' \leq n$, and
- $[m', n'] \leq \langle m, n\rangle$ whenever $m' \geq m$ and $n' \leq n$.

Thus controllable intervals can be refined by narrowing them, at most until they become singleton intervals, or until they are changed to uncontrollable intervals. Let us denote the collection of all possible sets of outgoing transitions from a state s by $\mathrm{Tran}(s) := \{E \subseteq T(s) \mid E \models \Phi(s)\}$ where $\models$ is the classical satisfaction relation on propositional formulae assuming that E lists all true propositions.

Definition 3 (Modal Refinement). *Let $\mathcal{S}_1 = (S_1, T_1, D_1, \Phi_1, s_1)$ and $\mathcal{S}_2 = (S_2, T_2, D_2, \Phi_2, s_2)$ be two MTSDs. A binary relation $R \subseteq S_1 \times S_2$ is a* modal refinement *if for every $(s, t) \in R$ the following holds:*

$$\begin{aligned}
&\forall M \in \mathrm{Tran}(s) : \exists N \in \mathrm{Tran}(t) : \\
&\qquad \forall (a, s') \in M : \exists (a, t') \in N : D_1(s, a, s') \leq D_2(t, a, t') \ \wedge \ (s', t') \in R \text{ and} \\
&\qquad \forall (a, t') \in N : \exists (a, s') \in M : D_1(s, a, s') \leq D_2(t, a, t') \ \wedge \ (s', t') \in R \ .
\end{aligned}$$

We say that $s \in S_1$ modally refines $s' \in S_2$, denoted by $s \leq_{\mathrm{m}} s'$, if there exists a modal refinement R such that $(s, s') \in R$. We also write $\mathcal{S}_1 \leq_{\mathrm{m}} \mathcal{S}_2$ if $s_1 \leq_{\mathrm{m}} s_2$.

Intuitively, the pair (s, t) can be in the relation R if for any satisfiable instantiation of outgoing edges from s there is a satisfiable instantiation of outgoing edges from t so that they can be mutually matched, possibly with s having more refined intervals, and the resulting states are again in the relation R.

Observe that in our running example the following systems are in modal refinement: $\mathcal{I}_1 \leq_{\mathrm{m}} \mathcal{S}_1 \leq_{\mathrm{m}} \mathcal{S}$ and thus also $\mathcal{I}_1 \leq_{\mathrm{m}} \mathcal{S}$, and similarly $\mathcal{I}_2 \leq_{\mathrm{m}} \mathcal{S}_2 \leq_{\mathrm{m}} \mathcal{S}$ and thus also $\mathcal{I}_2 \leq_{\mathrm{m}} \mathcal{S}$.

The reader can verify that on the standard modal transition systems (see Remark 2) the modal refinement relation corresponds to the classical modal refinement as introduced in [16].

3 Dual-Price Scheme

In this section, we formally introduce a dual-price scheme on top of MTSD in order to model the *investment cost* (cost of hardware necessary to perform the implemented actions) and the *running cost* (weighted long-run average of running costs of actions). We therefore consider only deadlock-free implementations (every state has at least one outgoing transition) so that the long-run average reward is well defined.

Definition 4 (Dual-Price Scheme). *A* dual-price scheme *over an alphabet Σ is a tuple $\mathcal{P} = (r, H, \Psi, i)$ where*

– $r : \Sigma \to \mathbb{Z}$ *is a* running cost *function of actions per time unit,*
– H *is a finite set of available* hardware*,*
– $\Psi : \Sigma \to \mathcal{B}(H)$ *is a* hardware requirement *function, and*
– $i : H \to \mathbb{N}_0$ *is a hardware* investment cost *function.*

Hence every action is assigned its unit cost and every action can have different hardware requirements (specified as a Boolean combination of hardware components) on which it can be executed. This allows for much more variability than a possible alternative of a simple investment cost $\Sigma \to \mathbb{N}_0$. Further, observe that the running cost may be negative, meaning that execution of such an action actually gains rather than spends resources.

Let $\mathcal{I}$ be an implementation with an initial state s_0. A set $G \subseteq H$ of hardware is *sufficient* for an implementation $\mathcal{I}$, written $G \models \mathcal{I}$, if $G \models \Psi(a)$ for every action a reachable from s_0. The *investment cost* of $\mathcal{I}$ is then defined as

$$\mathsf{ic}(\mathcal{I}) = \min_{G \models \mathcal{I}} \sum_{g \in G} i(g) \ .$$

Further, a *run* of $\mathcal{I}$ is an infinite sequence $s_0 a_0 t_0 s_1 a_1 t_1 \cdots$ with $(s_i, a_i, s_{i+1}) \in T$ and $t_i \in D(s_i, a_i, s_{i+1})$. Hence, in such a run, a concrete time duration in each uncontrollable interval is selected. We denote the set of all runs of $\mathcal{I}$ by $\mathcal{R}(\mathcal{I})$. The *running cost* of an implementation $\mathcal{I}$ is the worst-case long-run average

$$\mathsf{rc}(\mathcal{I}) = \sup_{s_0 a_0 t_0 s_1 a_1 t_1 \cdots \in \mathcal{R}(\mathcal{I})} \limsup_{n \to \infty} \frac{\sum_{i=0}^{n} r(a_i) \cdot t_i}{\sum_{i=0}^{n} t_i} \ .$$

Our *cheapest-implementation problem* is now defined as follows: given an MTSD specification $\mathcal{S}$ together with a dual-price scheme over the same alphabet, and given an upper-bound max_{ic} for the investment cost, find an implementation $\mathcal{I}$ of $\mathcal{S}$ (i.e. $\mathcal{I} \leq_{\mathrm{m}} \mathcal{S}$) such that $\mathsf{ic}(\mathcal{I}) \leq max_{\mathsf{ic}}$ and for every implementation $\mathcal{I}'$ of $\mathcal{S}$ with $\mathsf{ic}(\mathcal{I}') \leq max_{\mathsf{ic}}$, we have $\mathsf{rc}(\mathcal{I}) \leq \mathsf{rc}(\mathcal{I}')$.

Further, we introduce the respective decision problem, the *implementation problem*, as follows: given an MTSD specification $\mathcal{S}$ together with a dual-price scheme, and given an upper-bound max_{ic} for the investment cost and an upper bound max_{rc} on the running cost, decide whether there is an implementation $\mathcal{I}$ of $\mathcal{S}$ such that both $\mathsf{ic}(\mathcal{I}) \leq max_{\mathsf{ic}}$ and $\mathsf{rc}(\mathcal{I}) \leq max_{\mathsf{rc}}$.

Example 5. Figure 1f depicts a dual-price scheme over the same alphabet $\Sigma = \{\texttt{Wait}, \texttt{Drive}, \texttt{SmallCleanup}, \texttt{BigCleanup}, \texttt{SkipCleanup}\}$ as of our motivating specification $\mathcal{S}$. The running cost of the implementation $\mathcal{I}_2$ is $(1 \cdot 8 + 10 \cdot 10 + 6 \cdot 6 + 1 \cdot 8 + 10 \cdot 10 + 30 \cdot 7)/(1 + 10 + 6 + 1 + 10 + 30) \approx 7.97$ as the maximum value is achieved when `Drive` (with running cost 10) takes 10 minutes. On the one hand, this is optimal for $\mathcal{S}$ and a maximum investment cost at least 100. On the other hand, if the maximum investment cost is 99 or less then the optimal implementation is depicted in $\mathcal{I}_1$ and its cost is $(5 \cdot 8 + 10 \cdot 10 + 6 \cdot 5)/(5 + 10 + 6) \approx 8.10$.

Remark 6. Note that the definition of the dual-price scheme only relies on having durations on the labelled transition systems. Hence, one could easily apply this in various other settings like in the special case of traditional MTS (with may and must transitions instead of the obligation function) or in the more general case of parametric MTS (see [6]) when equipped with durations as described above.

4 Complexity Results

In this section, we give an overview of the complexity of our problem both in general and in an important special case. We start with establishing the hardness results. The matching upper bounds and the outline of their proofs follow. When referring to the size of MTSDs and the dual-price scheme, we implicitly assume binary encoding of numbers. We start by observing that the implementation problem is NP-hard even if no hardware is involved.

Proposition 7. *The implementation problem is NP-hard even for the hardware requirement function Ψ that is constantly true for all actions.*

Proof. We shall reduce the satisfiability problem of Boolean formulae (SAT) to our problem. Let φ be a Boolean formula over the variables $x_1, \ldots, x_n$. We define a MTSD $\mathcal{S}$ over the set of actions $\Sigma = \{x_1, \ldots, x_n, *\}$ such that the running cost is $r(x_j) = 1$ for all $1 \leq j \leq n$ and $r(*) = 2$ and the duration of all actions is 1. The specification $\mathcal{S}$ has one state s and a self-loop under all elements of Σ with the obligation function $\Phi(s) = \varphi \vee (*, s)$. The reason for adding the action $*$ is to make sure that in case φ is not satisfiable then we can still have a deadlock-free, but more running-cost-expensive implementation. Now we set the hardware to $H = \emptyset$ and the hardware requirement function $\Psi(a)$ constantly true for all $a \in \Sigma$. It is easy to observe that the formula φ is satisfiable iff $\mathcal{S}$ has an implementation $\mathcal{I}$ with $\mathsf{rc}(\mathcal{I}) \leq 1$ (and $\mathsf{ic}(\mathcal{I}) = 0$). □

Note that in the proof we required Φ to be a general Boolean formula. If, for instance, we considered Φ in *positive form* (i.e. only containing $\wedge$ and $\vee$ operators and not $\neg$), the hardness would not hold. Thus on the one hand, one source of hardness is the complexity of Φ. On the other hand, even if Φ corresponds to the simplest case of an implementation (Φ is a conjunction of atomic propositions), the problem remains hard due to the hardware.

Proposition 8. *The implementation problem is NP-hard even for specifications that are already implementations.*

Proof. We reduce the NP-complete problem of vertex cover to our problem. Let (V, E) where $E \subseteq V \times V$ be a graph and $k \in \mathbb{N}$ be an integer. We ask whether there is a subset of vertices $V_k \subseteq V$ of cardinality k such that for every $(v_1, v_2) \in E$ at least $v_1 \in V_k$ or $v_2 \in V_k$. Let us construct an MTSD specification $\mathcal{S}$ with hardware $H = V$ and the investment function $i(v) = 1$ for all $v \in H$, such that $\mathcal{S}$ has only one state s and a self-loop under a single action

a that is required ($\Phi(s) = (a, s)$) and where the hardware requirement function is $\Psi(a) = \bigwedge_{(u,v)\in E}(u \vee v)$. There is now a vertex cover in (V, E) of size k iff $\mathcal{S}$ has an implementation $\mathcal{I}$ with $\mathsf{ic}(\mathcal{I}) \leq k$. Setting e.g. $D(s, a, s) = 1$ and the running cost $r(a) = 0$ establishes NP-hardness of the implementation problem where we ask for the existence of an implementation of $\mathcal{S}$ with maximum running cost 0 and maximum investment cost k.

Alternatively, we may introduce a self-loop with a new action name $a_{(u,v)}$ for every edge (u, v) in the graph such that $\Psi(a_{(u,v)}) = u \vee v$, showing NP-hardness even for the case where the hardware requirement function is a simple disjunction of hardware components. □

In the subsequent sections, we obtain the following matching upper bound which yields the following theorem.

Theorem 9. *The implementation problem is NP-complete.*

By analysing the proof of Proposition 8, it is clear that we have to restrict the hardware requirement function before we can obtain a more efficient algorithm for the implementation problem. We do so by assuming a constant number of hardware components (not part of the input). If we at the same time require the obligation function in positive form, we obtain a simpler problem as stated in the following theorem.

Theorem 10. *The implementation problem with positive obligation function and a constant number of hardware components is polynomially equivalent to mean payoff games and thus it is in NP∩coNP and solvable in pseudo-polynomial time.*

The subsequent sections are devoted to proving Theorems 9 and 10. The algorithm to solve the implementation problem first reduces the dual-priced MTSD into a mean payoff game extended with time durations and then solves this game. This new extension of mean payoff games and an algorithm to solve them is presented in Section 4.1. The translation follows in Section 4.2. Since this translation is exponential in general, Section 4.3 then shows how to translate in polynomial time with only local exponential blow-ups where negations occur. Section 4.4 then concludes and establishes the complexity bounds.

4.1 Weighted Mean Payoff Games

We extend the standard model of mean payoff games (MPG) [14] with time durations. Not only is this extension needed for our algorithm, but it is also useful for modelling by itself. Consider, for instance, energy consumption of 2kW for 10 hours and 10kW for 2 hour, both followed by 10 hours of inactivity. Obviously, although both consumptions are 20kWh per cycle, the average consumption differs: 1kW in the former case and 20/12kW in the latter one. We also allow zero durations in order to model e.g. discrete changes of states, an essential part of our algorithm. Another extension of MPGs with dual-cost was studied in [8].

Definition 11. *A* weighted mean payoff game *is* $G = (V, V_{min}, V_{max}, E, r, d)$ *where* V *is a set of vertices partitioned into* V_{min} *and* V_{max}*,* $E \subseteq V \times V$ *is a set of edges,* $r : E \to \mathbb{Z}$ *is a rate function,* $d : E \to \mathbb{N}_0$ *is a duration function.*

It is assumed that there are no deadlocks (vertices with out-degree 0) and that there are no zero-duration cycles. The game is played by two players, *min* and *max*. The play is an infinite path such that each player picks successors in his/her vertices. The value of a play $v_0 v_1 v_2 \cdots$ is defined as:

$$\nu(v_0 v_1 v_2 \cdots) = \limsup_{n \to \infty} \frac{\sum_{i=0}^{n} r(v_i, v_{i+1}) \cdot d(v_i, v_{i+1})}{\sum_{i=0}^{n} d(v_i, v_{i+1})} \ . \qquad (*)$$

Player *min* tries to minimize this value, while *max* aims at the opposite. Let $v(s)$ denote the infimum of the values *min* can guarantee if the play begins in the vertex s, no matter what the player *max* does.

Note that the standard MPGs where edges are assigned only integer weights can be seen as weighted MPGs with rates equal to weights and durations equal to 1 on all edges.

We now show how to solve weighted MPGs by reduction to standard MPGs. We first focus on the problem whether $v(s) \geq 0$ for a given vertex s. As the durations are nonnegative and there are no zero-duration cycles, the denominator of the fraction in $(*)$ will be positive starting from some n. Therefore, the following holds for every play $v_0 v_1 v_2 \ldots$ and every (large enough) n:

$$\frac{\sum_{i=0}^{n} r(v_i, v_{i+1}) \cdot d(v_i, v_{i+1})}{\sum_{i=0}^{n} d(v_i, v_{i+1})} \geq 0 \iff \frac{1}{n} \sum_{i=0}^{n} r(v_i, v_{i+1}) \cdot d(v_i, v_{i+1}) \geq 0 \ .$$

We may thus solve the question whether $v(s) \geq 0$ by transforming the weighted MPG into a standard MPG, leaving the set of vertices and edges the same and taking $w(u, v) = r(u, v) \cdot d(u, v)$ as the edge weight function. Although the value $v(s)$ may change in this reduction, its (non)negativeness does not.

Further, we may transform any problem of the form $v(s) \geq \lambda$ for any fixed constant λ into the above problem. Let us modify the weighted MPG as follows. Let $r'(u, v) = r(u, v) - \lambda$ and leave everything else the same. The value of a play $v_0 v_1 v_2 \cdots$ is thus changed as follows.

$$\nu'(v_0 v_1 v_2 \cdots) = \limsup_{n \to \infty} \frac{\sum_{i=0}^{n} (r(v_i, v_{i+1}) - \lambda) \cdot d(v_i, v_{i+1})}{\sum_{i=0}^{n} d(v_i, v_{i+1})} = \nu(v_0 v_1 v_2 \cdots) - \lambda$$

It is now clear that $v(s) \geq \lambda$ in the original game if and only if $v'(s) \geq 0$ in the modified game.

Furthermore, there is a one-to-one correspondence between the strategies in the original weighted MPG and the constructed MPG. Due to the two equivalences above, this correspondence preserves optimality. Therefore, there are optimal positional strategies in weighted MPGs since the same holds for standard MPGs [14]. (A strategy is positional if its decision does not depend on the current history of the play but only on the current vertex, i.e. can be described as a function $V \to V$.)

4.2 Translating Dual-Priced MTSD into Weighted MPG

We first focus on the implementation problem without considering the hardware ($H = \emptyset$). We show how the implementation problem can be solved by reduction to the weighted MPGs. The first translation we present is exponential, however, we provide methods for making it smaller in the subsequent section.

We are given an MTSD $\mathcal{S} = (S, T, D, \Phi, s_0)$ and a dual-price scheme (r, H, Ψ, i) and assume that there is no state s with $\emptyset \in \mathrm{Tran}(s)$. Let us define the following auxiliary vertices that will be used to simulate the more complicated transitions of MTSD in the simpler setting of weighted MPG (by convention all singleton intervals are treated as uncontrollable).

$$\begin{aligned} T_{\mathsf{u}} &= \{(s,a,t) \mid (s,a,t) \in T;\ D(s,a,t) \in \mathcal{I}_{\mathsf{u}}\} \\ T_{\mathsf{c}} &= \{(s,a,t) \mid (s,a,t) \in T;\ D(s,a,t) \in \mathcal{I}_{\mathsf{c}}\} \\ T_{*} &= \{(s,a,j,t) \mid (s,a,t) \in T;\ j \in D(s,a,t)\} \end{aligned}$$

We construct the weighted mean-payoff game with $V_{min} = S \cup T_{\mathsf{c}} \cup T_{*}$, $V_{max} = 2^T \cup T_{\mathsf{u}}$ and E defined as follows:

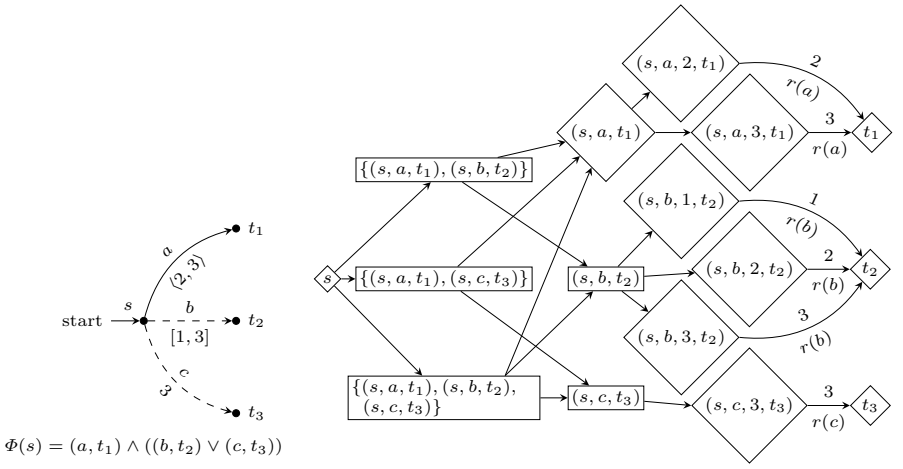
$$\begin{aligned} (s, X) \in E \quad &\iff \exists V \in \mathrm{Tran}(s) : X = \{(s,a,t) \mid (a,t) \in V\} \\ (X, (s,a,t)) \in E \quad &\iff (s,a,t) \in X \\ ((s,a,t),(s,a,j,t)) \in E \quad &\iff j \in D(s,a,t) \\ ((s,a,j,t),t) \in E \quad &\text{(always)} \end{aligned}$$

Further, $r((s,a,j,t),t) = r(a)$, $d((s,a,j,t),t) = j$ and $r(-,-) = d(-,-) = 0$ otherwise.

Example 12. In Figure 2 we show an example of how this translation to weighted MPG works. For simplicity we only translate a part of the MTSD $\mathcal{S}$ shown in Figure 2a. The resulting weighted MPG is shown in Figure 2b. The diamond shaped states belong to *min* and the squared states belong to *max*. In the vertex s, *min* chooses which outgoing transition are implemented. Only the choices satisfying $\Phi(s)$ are present in the game. Afterwards, *max* decides which transition to take. The chosen transition is then assigned by one of the players a time that it is going to take.

Notice that (s, a, t_1) is the only transition controlled by *min*, because it has a controllable interval $\langle 2, 3\rangle$. The remaining transitions with uncontrollable intervals are operated by *max* who chooses the time from these intervals. All the "auxiliary" transitions are displayed without any labels meaning their duration (and rate) is zero. Thus, only the transitions corresponding to "real" transitions in MTSDs are taken into account in the value of every play.

A strategy for *min* can now be translated into an implementation of the original MTSD in a straightforward way. The implemented transitions in s are given by $\sigma(s)$, similarly the durations of a transition (s, a, t) with a controllable interval are given by the third component of $\sigma((s, a, t))$.



(a) A part of an MTSD S (b) Resulting weighted MPG from translation of S

Fig. 2. Translating MTSD to weighted MPG

4.3 Optimizations

We now simplify the construction. The first simplification is summarized by the observation that the strategies of both players only need to choose the extremal points of the interval in vertices of the form (s, a, t).

Lemma 13. *There are optimal positional strategies for both* min *and* max *such that the choice in vertices of the form* (s, a, t) *is always one of the two extremal points of the interval* $D(s, a, t)$.

We may thus simplify the construction according to the previous lemma so that there are at most two outgoing edges for each state of the form (s, a, t) are as follows: $((s, a, t), (s, a, j, t)) \in E$ iff j is an extremal point of $D(s, a, t)$.

We can also optimize the expansion of $\mathrm{Tran}(s)$. So far, we have built an exponentially larger weighted MPG graph as the size of $\mathrm{Tran}(s)$ is exponential in the out-degree of s. However, we can do better if we restrict ourselves to the class of MTSD where all $\Phi(s)$ are positive boolean formulae, i.e. the only connectives are $\wedge$ and $\vee$. Instead of enumerating all valuations, we can use the syntactic tree of the formula to build a weighted MPG of polynomial size.

Let $sf(\varphi)$ denote the set of all sub-formulae of φ (including φ). Let further $S_* = \{(s, \varphi) \mid s \in S;\ \varphi \in sf(\Phi(s))\}$. The weighted MPG is constructed with

- $V_{min} = \{(s, \varphi) \in S_* \mid \varphi = \varphi_1 \vee \varphi_2$ or $(\varphi = (a, t)$ and $D(s, a, t) \in \mathcal{I}_\mathsf{c})\} \cup T_*$
- $V_{max} = \{(s, \varphi) \in S_* \mid \varphi = \varphi_1 \wedge \varphi_2$ or $(\varphi = (a, t)$ and $D(s, a, t) \in \mathcal{I}_\mathsf{u})\}$

– E is defined as follows:

$$\begin{aligned}
((s, \varphi_1 \wedge \varphi_2), (s, \varphi_i)) &\in E \quad i \in \{1, 2\} \\
((s, \varphi_1 \vee \varphi_2), (s, \varphi_i)) &\in E \quad i \in \{1, 2\} \\
((s, (a, t)), (s, a, j, t)) &\in E \quad \iff j \text{ is an extremal point of } D(s, a, t) \\
((s, a, j, t), (t, \Phi(t))) &\in E \quad \text{(always)}
\end{aligned}$$

– $r((s, a, j, t), (t, \Phi(t))) = r(a)$ and $r(-, -) = 0$ otherwise
– $d((s, a, j, t), (t, \Phi(t))) = j$ and $d(-, -) = 0$ otherwise.

Example 14. In Figure 3 we show the result of translating the part of an MTSD from Figure 2a. This weighted MPG is similar to the one in Figure 2b, but instead of having a vertex for each satisfying set of outgoing transitions, we now have the syntactic tree of the obligation formula for each state. Further the vertex $(s, b, 2, t_2)$ is left out, due to Lemma 13. Note that the vertices $(t_1, \Phi(t_1))$, $(t_2, \Phi(t_2))$ and $(t_3, \Phi(t_3))$ are drawn as circles, because the player of these states depends on the obligation formula and the outgoing transitions.

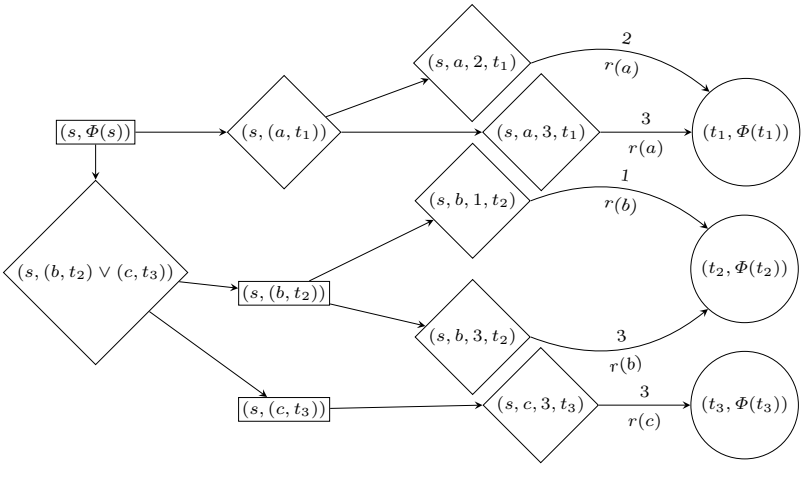


Fig. 3. Result of the improved translation of S in Figure 2a

Remark 15. Observe that one can perform this optimization even in the general case. Indeed, for those s where $\Phi(s)$ is positive we locally perform this transformation; for s with $\Phi(s)$ containing negations we stick to the original expansion. Thus, the exponential (in out-degree) blow-up occurs only locally.

Lemma 16. *Both optimized translations are correct and on MTSDs where the obligation function is positive they run in polynomial time.*

4.4 The Algorithm and Its Complexity

The algorithm for our problem, given a specification $\mathcal{S}$, works as follows.

1. Nondeterministically choose hardware with the total price at most max_{ic}.
2. Create the weighted MPG out of $\mathcal{S}$.
3. Solve the weighted MPG using the reduction to MPG and any standard algorithm for MPG that finds an optimal strategy for player *min* and computes the value $v(s_0)$.
4. Transform the strategy to an implementation $\mathcal{I}$.
5. In the case of the cheapest-implementation problem return $\mathcal{I}$;
   in the case of the implementation (decision) problem return $v(s_0) \leq max_{\mathsf{rc}}$.

We can now prove the following result, finishing the proof of Theorem 9.

Proposition 17. *The implementation problem is in NP.*

Proof. We first nondeterministically guess the hardware assignment. Due to Section 4.2, we know that the desired implementation has the same states as the original MTSD and its transitions are a subset of the transitions of the original MTSD as the corresponding optimal strategies are positional. The first optimization (Section 4.3) guarantees that durations can be chosen as the extremal points of the intervals. Thus we can nondeterministically guess an optimal implementation and its durations, and verify that it satisfies the price inequality. □

Proposition 18. *The implementation problem for MTSD with positive obligation function and a constant number of hardware components is in NP ∩ coNP and solvable in pseudo-polynomial time.*

Proof. With the constant number of hardware components, we get a constant number of possible hardware configurations and we can check each configuration separately one by one. Further, by the first and the second optimization in Section 4.3, the MPG graph is of size $\mathcal{O}(|T| + |\Phi|)$. Therefore, we polynomially reduce the implementation problem to the problem of solving constantly many mean payoff games. The result follows by the existence of pseudo-polynomial algorithms for MPGs [18]. □

Further, our problem is at least as hard as solving MPGs that are clearly a special case of our problem. Hence, Theorem 10 follows.

5 Conclusion and Future Work

We have introduced a new extension of modal transition systems. The extension consists in introducing (1) variable time durations of actions and (2) pricing of actions, where we combine one-shot investment price for the hardware and cost for running it per each time unit it is active. We believe that this formalism is appropriate to modelling many types of embedded systems, where safety comes along with economical requirements.

We have solved the problem of finding the cheapest implementation w.r.t. the running cost given a maximum hardware investment we can afford, and we established the complexity of the decision problem in the general setting and in a practically relevant subcase revealing a close connection with mean payoff games.

As for the future work, apart from implementing the algorithm, one may consider two types of extensions. First, one can extend the formalism to cover the distinction between input, output and internal actions as it is usual in interface theories [12], and include even more time features, such as clocks in priced timed automata [4,10]. Second, one may extend the criteria for synthesis of the cheapest implementation by an additional requirement that the partial sums stay within given bounds as done in [11], or requiring the satisfaction of a temporal property as suggested in [12,13].

References

1. Aceto, L., Fábregas, I., de Frutos-Escrig, D., Ingólfsdóttir, A., Palomino, M.: Graphical representation of covariant-contravariant modal formulae. In: EXPRESS. EPTCS, vol. 64, pp. 1–15 (2011)
2. Antonik, A., Huth, M., Larsen, K.G., Nyman, U., Wasowski, A.: 20 years of modal and mixed specifications. Bulletin of the EATCS (95), 94–129 (2008)
3. Bauer, S.S., Fahrenberg, U., Juhl, L., Larsen, K.G., Legay, A., Thrane, C.R.: Quantitative Refinement for Weighted Modal Transition Systems. In: Murlak, F., Sankowski, P. (eds.) MFCS 2011. LNCS, vol. 6907, pp. 60–71. Springer, Heidelberg (2011)
4. Behrmann, G., Larsen, K.G., Rasmussen, J.I.: Priced Timed Automata: Algorithms and Applications. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2004. LNCS, vol. 3657, pp. 162–182. Springer, Heidelberg (2005)
5. Beneš, N., Křetínský, J.: Process algebra for modal transition systemses. In: MEMICS. OASICS, vol. 16, pp. 9–18. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany (2010)
6. Beneš, N., Křetínský, J., Larsen, K., Møller, M., Srba, J.: Parametric Modal Transition Systems. In: Bultan, T., Hsiung, P.-A. (eds.) ATVA 2011. LNCS, vol. 6996, pp. 275–289. Springer, Heidelberg (2011)
7. Beneš, N., Křetínský, J., Larsen, K., Møller, M., Srba, J.: Dual-priced modal transition systems with time durations. Tech. Rep. FIMU-RS-2012-01, Faculty of Informatics MU (2012)
8. Bloem, R., Greimel, K., Henzinger, T.A., Jobstmann, B.: Synthesizing robust systems. In: Proc. of FMCAD 2009, pp. 85–92. IEEE (2009)
9. Boudol, G., Larsen, K.G.: Graphical versus logical specifications. Theor. Comput. Sci. 106(1), 3–20 (1992)
10. Bouyer, P., Brinksma, E., Larsen, K.G.: Optimal infinite scheduling for multi-priced timed automata. Formal Methods in System Design 32(1), 3–23 (2008)
11. Bouyer, P., Fahrenberg, U., Larsen, K.G., Markey, N., Srba, J.: Infinite Runs in Weighted Timed Automata with Energy Constraints. In: Cassez, F., Jard, C. (eds.) FORMATS 2008. LNCS, vol. 5215, pp. 33–47. Springer, Heidelberg (2008)
12. Chakrabarti, A., de Alfaro, L., Henzinger, T.A., Stoelinga, M.: Resource Interfaces. In: Alur, R., Lee, I. (eds.) EMSOFT 2003. LNCS, vol. 2855, pp. 117–133. Springer, Heidelberg (2003)

13. Chatterjee, K., Doyen, L.: Energy Parity Games. In: Abramsky, S., Gavoille, C., Kirchner, C., Meyer auf der Heide, F., Spirakis, P.G. (eds.) ICALP 2010, Part II. LNCS, vol. 6199, pp. 599–610. Springer, Heidelberg (2010)
14. Ehrenfeucht, A., Mycielski, J.: Positional strategies for mean payoff games. International Journal of Game Theory 8, 109–113 (1979), doi:10.1007/BF01768705
15. Juhl, L., Larsen, K.G., Srba, J.: Introducing modal transition systems with weight intervals. Journal of Logic and Algebraic Programming (2011)
16. Larsen, K.G., Thomsen, B.: A modal process logic. In: LICS, pp. 203–210. IEEE Computer Society (1988)
17. Larsen, K.G., Xinxin, L.: Equation solving using modal transition systems. In: LICS, pp. 108–117. IEEE Computer Society (1990)
18. Zwick, U., Paterson, M.: The complexity of mean payoff games on graphs. Theoretical Computer Science 158, 343–359 (1996)

Finding Finite Herbrand Models

Stefan Borgwardt and Barbara Morawska*

Theoretical Computer Science, TU Dresden, Germany
{stefborg,morawska}@tcs.inf.tu-dresden.de

Abstract. We show that finding finite Herbrand models for a restricted class of first-order clauses is ExpTime-complete. A Herbrand model is called finite if it interprets all predicates by finite subsets of the Herbrand universe. The restricted class of clauses consists of anti-Horn clauses with monadic predicates and terms constructed over unary function symbols and constants. The decision procedure can be used as a new goal-oriented algorithm to solve linear language equations and unification problems in the description logic $\mathcal{FL}_0$. The new algorithm has only worst-case exponential runtime, in contrast to the previous one which was even best-case exponential.

1 Introduction

Satisfiability of formulas in First Order Logic (FOL) has always been of interest for computer science and is an active field of research. The main problem is that satisfiability of such formulas is not even semi-decidable. Thus, the focus lies on finding algorithms that decide satisfiability for restricted classes. A possible approach is to use restrictions on the resolution or superposition calculi to obtain decision procedures [8,10].

Related to this is the problem of *model building* that asks for an actual model witnessing the satisfiability of the given clauses. Additionally, one usually asks for a finite representation of such a model. For example, the completeness proofs of resolution-style inference systems sometimes explicitly construct (counter-)models, but there are also other approaches [2,11,16].

Here, we want to study the related problem of finding finite Herbrand models. We call a Herbrand model *finite* if each predicate is interpreted by a finite subset of the Herbrand universe. This problem is semi-decidable since the finite Herbrand interpretations over a fixed signature can be recursively enumerated. It has not been studied before and it is unknown whether it is decidable for arbitrary first-order formulae. The existence of finite Herbrand models implies the existence of finite models in the usual sense, where the domain is required to be finite, but the other implication does not hold in general.

We restrict ourselves to finite sets of *propagation rules*, which are anti-Horn clauses that use only monadic predicates and function symbols, one constant symbol, and one variable. In particular, we do not allow the equality predicate.

* The authors are supported by DFG under grant BA 1122/14-1.

N. Bjørner and A. Voronkov (Eds.): LPAR-18, LNCS 7180, pp. 138–152, 2012.

These sets of clauses can be seen as skolemized versions of Ackermann formulas, for which satisfiability is known to be decidable [7,10]. This class of clause sets is also similar to the decidable Bernays-Schönfinkel class [10], but neither is actually included in the other.

In this paper, we show that the problem of deciding the existence of a finite Herbrand model for a finite set of propagation rules is ExpTime-complete. Our decision procedure is aided by a new computational model that we call *propagation nets*. The process of building a model is simulated by the process of saturating the net with terms. This process terminates iff a finite Herbrand model exists. We decide this by analyzing the structure of the net.

The problem of finding finite Herbrand models for a set of propagation rules occurred while designing a new unification procedure for the description logic $\mathcal{FL}_0$. The unification problem in this logic was shown to be ExpTime-complete in [1]. There, solving unification in $\mathcal{FL}_0$ is shown to be equivalent to solving linear language equations. The problem of solving these equations reduces in a natural way to the problem of finding finite Herbrand models for propagation rules. In this reduction, variables become predicates and their finite interpretation in the Herbrand universe defines a solution to the original language equation.

Our decision procedure thus provides a new way to solve linear language equations. It is worst-case exponential, but there are cases in which our algorithm runs in polynomial time. Thus, it has advantages over the previous algorithm [1], which is always exponential.

We think that this method of finding finite Herbrand models can be generalized to larger classes of clauses. As detailed above, it has an immediate application to unification and solving formal language equations.

This paper does not include the formal proofs of our results. These and more detailed explanations can be found in the technical report [4].

2 Propagation Rules

We first introduce *propagation rules*, which are clauses over a signature of finitely many unary predicates $\mathcal{P}$, finitely many unary function symbols $\mathcal{F}$, one constant a, and one variable x. Every ground term over this signature is of the form $f_1(\dots f_n(a)\dots)$, which we will abbreviate as $f_1 \dots f_n(a)$. A *propagation rule* is a clause of the form $\top \to P_1(a) \vee \dots \vee P_n(a)$ *(positive clause)*, $P_0(a) \to P_1(a) \vee \dots \vee P_n(a)$, or $P_0(t_0) \to P_1(t_1) \vee \dots \vee P_n(t_n)$ for $P_0, \dots, P_n \in \mathcal{P}$ and non-ground terms $t_0, \dots, t_n$ over $\mathcal{F}$ and x.[1]

We assume that the reader is familiar with Herbrand interpretations (see, e.g., [10]). We call a Herbrand interpretation $\mathcal{H}$ over the above signature *finite* if it interprets every predicate $P \in \mathcal{P}$ by a finite set $P^{\mathcal{H}}$. The task we are interested in is to decide the existence of finite Herbrand models for finite sets of propagation rules. As a first step, we will flatten the propagation rules to get rid of most terms

[1] Note that n might be 0, in which case the right-hand side of the clause is $\bot$. Positive clauses must be ground since otherwise no finite Herbrand model could exist.

of depth larger than 0. A finite set $\mathcal{C}$ of propagation rules is called *normalized* if there is a set $\mathcal{D}(\mathcal{C}) \subseteq \mathcal{P} \times \mathcal{F}$ such that

- For every $(P, f) \in \mathcal{D}(\mathcal{C})$, we have $P^f \in \mathcal{P}$ and the clauses $P^f(x) \to P(f(x))$ (*increasing clause*) and $P(f(x)) \to P^f(x)$ (*decreasing clause*) in $\mathcal{C}$.
- All other clauses in $\mathcal{C}$ must be *flat*, i.e., of the form $\top \to P_1(a) \vee \cdots \vee P_n(a)$, $P_0(a) \to P_1(a) \vee \cdots \vee P_n(a)$, or $P_0(x) \to P_1(x) \vee \cdots \vee P_n(x)$.

For $f \in \mathcal{F}$, we denote by $\mathcal{D}^f(\mathcal{C})$ the set $\{P \in \mathcal{P} \mid (P, f) \in \mathcal{D}(\mathcal{C})\}$.

The interesting property of such sets is that in order to check whether a flat clause $P_0(x) \to P_1(x) \vee \cdots \vee P_n(x)$ is satisfied by a ground term, one only needs to consider this term. Different terms can only occur in the same instance of a clause if it is an increasing or a decreasing clause, which only allows a very limited connection between the terms, i.e., adding and removing the leading function symbol. The set $\mathcal{D}(\mathcal{C})$ acts as an "interface" between terms of different lengths: A clause can only contain different terms if a predicate P^f with $(P, f) \in \mathcal{D}(\mathcal{C})$ is involved. The special predicate P^f represents those terms in P that have the prefix f: For any Herbrand model $\mathcal{H}$ and any word $w \in \mathcal{F}^*$, the term $f(w(a))$ is in $P^{\mathcal{H}}$ iff $w(a)$ is in ${P^f}^{\mathcal{H}}$.

To transform a finite set $\mathcal{C}$ of propagation rules into a normalized set $\mathcal{C}'$, we introduce auxiliary predicates that allow us to replace arbitrary atoms by flat ones. For example, the atom $P(fg(x))$ can be replaced by the equivalent atom $P^{fg}(x)$ if (P, f) and (P^f, g) are added to $\mathcal{D}(\mathcal{C})$. In contrast to common flattening procedures for first-order clauses, we do not use new variables or equality [2].

Lemma 1. *For every finite set $\mathcal{C}$ of propagation rules, we can construct in polynomial time a normalized set $\mathcal{C}'$ of propagation rules such that $\mathcal{C}$ has a finite Herbrand model iff $\mathcal{C}'$ does.*

Example 2. Consider the propagation rules

$$\begin{aligned}\mathcal{C}_1 := \{&\top \to P_0(a),\ P_0(f(x)) \to \bot,\ P_0(g(x)) \to \bot,\ P_3(a) \to \bot,\ P_3(f(x)) \to \bot,\\ &P_3(g(x)) \to P_0(x),\ P_0(x) \to P_3(g(x)),\ P_0(a) \to P_1(a),\ P_1(a) \to P_0(a),\\ &P_2(x) \to P_3(x) \vee P_1(f(x)),\ P_3(x) \to P_2(x),\ P_1(f(x)) \to P_2(x)\\ &P_1(x) \to P_2(x) \vee P_1(g(x)),\ P_2(x) \to P_1(x),\ P_1(g(x)) \to P_1(x)\}.\end{aligned}$$

To construct the normalized set $\mathcal{C}_1'$, we first rename P_0 to P_3^g and add the pair (P_3, g) to $\mathcal{D}(\mathcal{C}_1')$. Afterwards, the pairs (P_3, f), (P_1, f), (P_1, g), (P_3^f, g), and (P_3^g, g) are added, together with the corresponding increasing and decreasing clauses. The resulting flat clauses are the following:

$$\begin{aligned}&\top \to P_3^g(a),\ P_3^{gf}(x) \to \bot,\ P_3^{gg}(x) \to \bot,\ P_3(a) \to \bot,\ P_3^f(x) \to \bot,\\ &P_3^g(a) \to P_1(a),\ P_1(a) \to P_3^g(a),\\ &P_2(x) \to P_3(x) \vee P_1^f(x),\ P_3(x) \to P_2(x),\ P_1^f(x) \to P_2(x),\\ &P_1(x) \to P_2(x) \vee P_1^g(x),\ P_2(x) \to P_1(x),\ P_1^g(x) \to P_1(x).\end{aligned}$$

We will use $\mathcal{C}_1'$ throughout this paper to illustrate the presented algorithms.

For a flat clause c, the set $\texttt{possibilities}(c)$ contains all predicates occurring on the right-hand side of c. For a set $\mathcal{C} = \{c_1, \ldots, c_n\}$ of flat clauses, we define $\texttt{possibilities}(\mathcal{C}) := \{\{P_1, \ldots, P_n\} \mid \forall i \in \{1, \ldots, n\} : P_i \in \texttt{possibilities}(c_i)\}$. For example, $P_1(a) \rightarrow P_2(a) \vee P_3(a)$ has the possibilities P_2 and P_3, while $\{P_1(x) \rightarrow P_2(x) \vee P_3(x), \top \rightarrow P_0(a)\}$ has the possibilities $\{P_2, P_0\}$ and $\{P_3, P_0\}$.

In the following, we assume that any normalized set $\mathcal{C}$ of propagation rules contains at most one positive clause, which is of the form $\top \rightarrow A(a)$, and that the predicate A otherwise only occurs on the left-hand side of other ground clauses. If this is not the case, we introduce a new predicate A, add the clause $\top \rightarrow A(a)$ to $\mathcal{C}$, and replace $\top$ by $A(a)$ in every other positive clause. It is easy to see that this modification does not affect the existence of a finite Herbrand model for $\mathcal{C}$.

For the set $\mathcal{C}_1'$ from Example 2, we simply add $\top \rightarrow A(a)$ to $\mathcal{C}_1'$ and replace the propagation rule $\top \rightarrow P_3^g(a)$ by $A(a) \rightarrow P_3^g(a)$.

3 Propagation Nets

We now introduce a new computational model, called *propagation net*, that will be used to decide the existence of finite Herbrand models for finite sets of propagation rules. We use notions borrowed from the theory of Petri nets [12,13].

A propagation net consists of places and transitions which are connected by directed arcs. A computation moves words from places to other places using the transitions between them. If a place has several outgoing arcs to transitions, it can choose one of them to *fire*. This means that a word from this place is transported to the transition and then distributed to all places reachable from this transition. An arc from a place to a transition can also change the word by adding a letter or removing the first letter. An arc from a transition to a place can filter out words that should not be transported to the place. The firing of a transition does not remove the word from the place but just deactivates it. The goal is to find a computation that starts with a given distribution of words among places and *terminates* in the sense that all words are deactivated.

Definition 3. *A* propagation net $\mathcal{N} = (P, T, \Sigma, E, I, \pi, \tau)$ *consists of*

- *a finite set* P *of* places,
- *a finite set* T *of* transitions,
- *a finite alphabet* Σ,
- *a set* $E \subseteq (P \times T) \cup (T \times P)$ *of* arcs,
- *an* initial marking $I : (P \cup T) \rightarrow \mathcal{P}(\Sigma^*)$ *and* $I_a : P \rightarrow \mathcal{P}(\Sigma^*)$,
- *a partial* filter function $\pi : \big(E \cap (T \times P)\big) \rightarrow \Sigma \cup \{\varepsilon\}$, *and*
- *a* successor function $\tau : \big(E \cap (P \times T)\big) \rightarrow \Sigma \cup \{f^{-1} | f \in \Sigma\} \cup \{\varepsilon\}$.

A *token* in $\mathcal{N}$ is a word over Σ. A *marking* M of $\mathcal{N}$ is a pair of mappings $M : (P \cup T) \rightarrow \mathcal{P}(\Sigma^*)$ and $M_a : P \rightarrow \mathcal{P}(\Sigma^*)$ assigning to each place and each transition finite sets of tokens such that $M_a(p) \subseteq M(p)$ for every $p \in P$. $M(p)$ contains the tokens of a place $p \in P$, while $M(t)$ contains the tokens of a transition $t \in T$ in the marking M. The set $M_a(p)$ contains the *active* tokens of p in M. We assume that I is a proper marking in the above sense.

We say that a token w *matches* the filter $\pi(t,p)$ of an arc $(t,p) \in E \cap (T \times P)$ if either (i) $\pi(t,p)$ is undefined (no restriction on w), (ii) $\pi(t,p) = \varepsilon$ and then $w = \varepsilon$, or (iii) $\pi(t,p) = f \in \Sigma$ and then w starts with f.

There are two elementary operations on markings. A token w is *deactivated* at $p \in P$ by removing it from $M_a(p)$, if it is in $M_a(p)$, and adding it to $M(p)$, if it is not already in $M(p)$. Note that w need not be in $M(p)$ to be deactivated.

A token w is *produced* at a transition $t \in T$ by adding it to $M(t)$. This operation has the side effect of also *producing* the token at all places $p \in P$ with $(t,p) \in E$. This secondary operation is executed only if w matches the filter $\pi(t,p)$. If this is the case and $w \notin M(p)$, then w is added to $M(p)$ and $M_a(p)$. Otherwise, the token w is not added to the marking at p.

A *firing* in $\mathcal{N}$ is a triple $\mathfrak{f} = (p,w,t) \in P \times \Sigma^* \times T$ such that $(p,t) \in E$ and the concatenation $\tau(p,t)w$ is defined, i.e., if $\tau(p,t) = f^{-1}$, then w begins with f. The result of firing $\mathfrak{f}$ in a marking M is a new marking M' as follows:

1. Initialize $M' := M$ and $M'_a := M_a$.
2. Deactivate the token w at p in M'.
3. Compute the *successor token* $w' := \tau(p,t)w$.
4. Produce w' at t in M', thereby also producing w' at every place reachable from t by an outgoing arc whose filter matches w'.

If M' is the result of the firing $\mathfrak{f}$ in M, then we write $M \xrightarrow{\mathfrak{f}} M'$. If $M(p) = M'(p)$ for all $p \in P$, this firing is called *unproductive* in M; otherwise, it is called *productive*. An unproductive firing only removes an active token from the marking, while a productive firing also introduces new active tokens.

Given a marking M_0, a *firing sequence (starting in M_0)* is a finite sequence $M_0 \xrightarrow{\mathfrak{f}_1} \dots \xrightarrow{\mathfrak{f}_m} M_m$ of firings. If the initial marking is not important, we denote this sequence by $\mathfrak{f}_1, \dots, \mathfrak{f}_m$. M_m is called the *final marking* of this sequence. The sequence is called *terminating* if M_m is *stable*, i.e., $M_{m,a}(p) = \emptyset$ for all $p \in P$. We say that $\mathcal{N}$ *terminates* if it has a terminating firing sequence that starts in I. Note that such a firing sequence has to end with a nonproductive firing since otherwise new active tokens would be created. Figures 1 and 2 depict a simple propagation net and the effect of a firing on the initial marking.

Other Computational Models. There are several differences between propagation nets and Petri nets. In propagation nets, tokens are not atomic objects, but words over an alphabet Σ. Additionally, transitions do not need to be synchronized, i.e., do not require the input token to be present at every input place.

Propagation nets behave much more like two-way alternating automata on finite words [5,9,3] or trees [14,6], where places are existential states and transitions are universal states. Contrary to word automata, however, propagation nets do not read an input word, but rather write several words, i.e., the tokens that are produced. In finite trees, one can represent all these words simultaneously. But then propagation nets would represent automata on finite trees that can also accept with infinite computations, contrary to the standard definition.

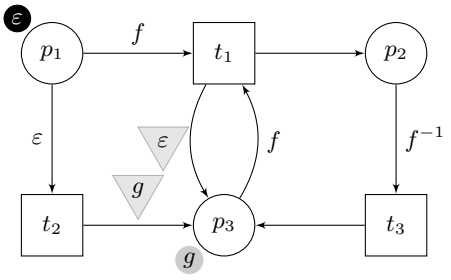


Fig. 1. A simple propagation net with $P = \{p_1, p_2, p_3\}$ and $T = \{t_1, t_2, t_3\}$. Edge labels denote the functions π and τ, where filters are depicted as triangles. Filled circles are the tokens of the initial marking; active tokens have a black background.

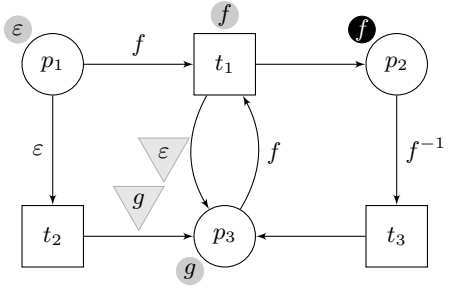


Fig. 2. The propagation net from Fig. 1 after firing (p_1, ε, t_1). The token f is produced at t_1 and p_2, but not at p_3 since f does not match the filter $\tau(t_1, p_3) = \varepsilon$.

From Clauses to Propagation Nets. We will now translate any normalized set $\mathcal{C}$ of propagation rules into a propagation net $\mathcal{N}_\mathcal{C}$. The goal is to express the finite Herbrand models of $\mathcal{C}$ by stable markings of $\mathcal{N}_\mathcal{C}$. We will represent terms by tokens, clauses by places, and predicates by transitions. From a clause, a token can be transferred to any of its possibilities. From a predicate, a token is then distributed to all clauses with this predicate on their left-hand side. The filter function allows to discard those terms (tokens) that are irrelevant for satisfying the clause. The successor function expresses increasing and decreasing clauses by adding or removing letters, respectively. For a flat clause, the successor function is ε, i.e., it leaves the term as it is. The initial marking simply consists of the active token ε at $\top \to A(a)$ since this is the only clause without precondition.

Definition 4. *Let $\mathcal{C}$ be a normalized set of propagation rules. The propagation net $\mathcal{N}_\mathcal{C} := (\mathcal{C}, \mathcal{P}, \mathcal{F}, E_\mathcal{C}, I_\mathcal{C}, \pi_\mathcal{C}, \tau_\mathcal{C})$ has the following components:*

- $E_\mathcal{C} := \big\{(c, P_i) \mid c = \ldots \to P_1(t_1) \vee \cdots \vee P_n(t_n) \in \mathcal{C} \text{ and } i \in \{1, \ldots, n\}\big\}$
  $\quad \cup \big\{(P_0, c) \mid c = P_0(t_0) \to \cdots \in \mathcal{C}\big\}$
- $I_{\mathcal{C},a}(c) := I_\mathcal{C}(c) := \begin{cases} \{\varepsilon\} & \text{if } c = \top \to A(a) \\ \emptyset & \text{otherwise} \end{cases}$

- $\pi_{\mathcal{C}}\big(P_0, P_0(t_0) \to \dots\big) := \begin{cases} \varepsilon & \text{if } t_0 = a \\ \text{undefined} & \text{if } t_0 = x \\ f & \text{if } t_0 = f(x) \end{cases}$
- $\tau_{\mathcal{C}}\big(P_0(t_0) \to P_1(t_1) \vee \dots \vee P_n(t_n), P_i\big) := \begin{cases} f & \text{if } t_0 = x,\ t_i = f(x) \\ f^{-1} & \text{if } t_0 = f(x),\ t_i = x \\ \varepsilon & \text{otherwise} \end{cases}$
- $\tau_{\mathcal{C}}(\top \to A(a), A) := \varepsilon$

In this propagation net, every firing (c, w, P) represents a possibility of c. Firing sequences can thus be seen as sequences of applying possibilities to tokens on the left-hand side of clauses: If $w(a)$ is a term in $P^{\mathcal{H}}$ for a Herbrand interpretation $\mathcal{H}$ and we want $\mathcal{H}$ to satisfy a clause $P(x) \to P_1(x) \vee \dots \vee P_n(x)$, then we have to find a possibility P_i for which to put $w(a)$ into $P_i^{\mathcal{H}}$. If this process of satisfying clauses stops, we have found a finite Herbrand model of $\mathcal{C}$.

Lemma 5. *$\mathcal{C}$ has a finite Herbrand model iff $\mathcal{N}_{\mathcal{C}}$ terminates.*

Example 6. Consider the propagation net $\mathcal{N}_{\mathcal{C}_1'}$ for the rules from Example 2. Ignoring unproductive firings, the following is a terminating firing sequence:

$$(\top \to A(a), \varepsilon, A),\ (A(a) \to P_3^g(a), \varepsilon, P_3^g),\ (P_3^g(x) \to P_3(g(x)), \varepsilon, P_3),$$
$$(P_3(x) \to P_2(x), g, P_2),\ (P_2(x) \to P_1(x), g, P_1),\ (P_1(g(x)) \to P_1^g(x), g, P_1^g),$$
$$(P_1^g(x) \to P_1(x), \varepsilon, P_1)$$

If we abbreviate firings like $(P_1(x) \to P_2(x) \vee P_1^g(x), g, P_2)$ by $P_1(g) \to P_2(g)$ and join "adjacent" firings, the structure of this sequence becomes apparent:

$$\begin{array}{ccccc} & & P_3(g) & \longrightarrow & P_2(g) \longrightarrow P_1(g) \\ & \nearrow & & & \searrow \\ \top & \longrightarrow A(\varepsilon) & \longrightarrow & P_3^g(\varepsilon) & \qquad\qquad P_1^g(\varepsilon) \to P_1(\varepsilon) \end{array}$$

It is easy to read off the corresponding finite Herbrand model $\mathcal{H}$ of $\mathcal{C}_1'$:

$$A^{\mathcal{H}} = P_1^{g\mathcal{H}} = P_3^{g\mathcal{H}} = \{a\},\ P_1^{\mathcal{H}} = \{a, g(a)\},\ P_2^{\mathcal{H}} = P_3^{\mathcal{H}} = \{g(a)\},$$
$$P_1^{f\mathcal{H}} = P_3^{f\mathcal{H}} = P_3^{gf\mathcal{H}} = P_3^{gg\mathcal{H}} = \emptyset.$$

3.1 Behavior of Propagation Nets

Our goal is to decide termination of propagation nets $\mathcal{N}_{\mathcal{C}}$ obtained from normalized sets of propagation rules $\mathcal{C}$. We will use these propagation nets to formulate the ideas behind a decision procedure for the existence of finite Herbrand models for the clause sets.

Termination of Propagation Nets. We first analyze what it means for $\mathcal{N}_{\mathcal{C}}$ to have a terminating firing sequence starting in $I_{\mathcal{C}}$. Any such sequence will start with the token ε at A and gradually distribute it to other predicates, while

sometimes increasing it. There are two reasons why this might not be possible. First, it may be impossible to avoid a contradiction, i.e., a clause with $\bot$ on the right-hand side, in any firing sequence starting in $I_{\mathcal{C}}$. The other possibility is that every firing sequence that avoids all contradictions is forced into a cycle of creating ever longer tokens. Thus, in order for the sequence to terminate, the length of the produced tokens has to be bounded. To analyze the detailed structure of terminating firing sequences, we introduce the following notions.

Definition 7. *Let $P \in \mathcal{X} \subseteq \mathcal{P}$ and $w = fw' \in \mathcal{F}^+$. A $(P, \mathcal{X}, w)$-*replacement sequence *is a firing sequence of $\mathcal{N}_{\mathcal{C}}$ starting in M_0 and ending in M_m such that*

- *M_0 only contains the token w at P and the active token w at all clauses with $P(x)$ or $P(f(x))$ on the left-hand side,*
- *M_m only contains tokens with the suffix w,*
- *$w \in M_m(Q)$ iff $Q \in \mathcal{X}$, and*
- *if $w' \in M_{m,a}(c)$, then $w' = w$ and $c = Q(f(x)) \to Q^f(x)$.*

*A (P, ε)-*replacement sequence *is a firing sequence starting in M_0 and ending in M_m such that*

- *M_0 only contains the token ε at P and the active token ε at all clauses with $P(x)$ or $P(a)$ on the left-hand side, and*
- *M_m is stable.*

The height *of a replacement sequence is the maximal number $|w'| - |w|$ for any token w' in M_m.*

Every terminating firing sequence starting in $I_{\mathcal{C}}$ consists of the firing $(\top \to A(a), \varepsilon, A)$ and an (A, ε)-replacement sequence. Thus, our goal is to decide the existence of such replacement sequences. If there is an (A, ε)-replacement sequence of height 0, then only the token ε is produced in this sequence. Deciding the existence of such sequences is easy (see Alg. 2). If the height of an (A, ε)-replacement sequence is larger than 0, it contains other replacement sequences of smaller height, as explained in the following.

The sequence has to produce a token $w = fw' \neq \varepsilon$ at a predicate P, and then w is contained in the final marking at all clauses with $P(x)$ or $P(f(x))$ on the left-hand side. We can extract a $(P, \mathcal{X}, w)$-replacement sequence as follows: Starting from the token w at all clauses with $P(x)$ or $P(f(x))$ on the left-hand side, we extract all firings that deactivate these tokens and the tokens produced from these firings, except firings of the form $(Q(f(x)) \to Q^f(x), w, Q^f)$. The extracted firings form the replacement sequence and the set $\mathcal{X}$ consists of all predicates Q at which w was produced in this sequence.

Example 8. The terminating firing sequence from Example 6 mainly consists of an (A, ε)-replacement sequence. The firing $(P_3^g(x) \to P_3(g(x)), \varepsilon, P_3)$ produces the token g at all clauses with $P_3(x)$ or $P_3(g(x))$ on the left-hand side, which is the

starting point of a replacement sequence. The corresponding $(P_3, \{P_3, P_2, P_1\}, g)$-replacement sequence is

$$(P_3(x) \to P_2(x), g, P_2),\ (P_2(x) \to P_1(x), g, P_1),$$
$$(P_2(x) \to P_3(x) \vee P_1^f(x), g, P_3),\ (P_1(x) \to P_2(x) \vee P_1^g(x), g, P_2).$$

If a longer token w' is produced in such a sequence at $Q \in \mathcal{P}$, we can use the same procedure to extract a $(Q, \mathcal{Y}, w')$-replacement sequence of smaller height. We continue this until the height of the replacement sequences is 0. Thus, every terminating firing sequence is decomposed into nested replacement sequences.

To decide termination of $\mathcal{N}_\mathcal{C}$, we construct all possible replacement sequences, starting with height 0. These can be used to build replacement sequences of increasing heights, until we can construct an (A, ε)-replacement sequence.

Replacement Sequences of Height 0. To construct replacement sequences of height 0 for a predicate P, we define the set $\texttt{possibilities}(P)$ to contain all possibilities of the set of all flat clauses with $P(x)$ on the left-hand side. Such a possibility $\{Q_1, \ldots, Q_n\}$ represents one way of firing all these flat clauses. Afterwards, we have to consider the possibilities of the reached predicates $Q_1, \ldots, Q_n$ and repeat this process until no new predicates are reached.

Since we want to find replacement sequences of height 0, we must prevent this process to reach predicates of the form P^f with $(P, f) \in \mathcal{D}(\mathcal{C})$. Thus, we define $\texttt{possibilities}(P^f(x) \to P(f(x))) := \emptyset$ and extend the set $\texttt{possibilities}(P^f)$ to also consider this increasing clause. Thus, $\texttt{possibilities}(P^f) = \emptyset$, which indicates that we have no way of dealing with the token w at P^f.

Example 9. The $(P_3, \{P_3, P_2, P_1\}, g)$-replacement sequence from Example 8 can be constructed as follows: For P_3, we have the possibility $\{P_2\}$, i.e., the firing $(P_3(x) \to P_2(x), g, P_2)$. P_2 has the possibilities $\{P_1, P_3\}$ and $\{P_1, P_1^f\}$. The first one yields $(P_2(x) \to P_1(x), g, P_1)$ and $(P_2(x) \to P_3(x) \vee P_1^f(x), g, P_3)$. The second possibility would lead to the active token g at P_1^f, which we disallow. Finally, for P_1 we choose the unproductive firing $(P_1(x) \to P_2(x) \vee P_1^g(x), g, P_2)$.

It is easy to see that a $(P, \mathcal{X}, w)$-replacement sequence can be changed into a $(P, \mathcal{X}, w')$-replacement sequence by substituting the suffix w by w' in every token in the sequence. Thus, the token w is not necessary to describe the replacement sequence. Similarly, it is not important which firings are used to deactivate tokens, only which predicates are reached. We are thus only interested in so-called *shortcuts* $(P, \mathcal{X})$ with $P \in \mathcal{X} \subseteq \mathcal{P}$ for which a $(P, \mathcal{X}, w)$-replacement sequence exists. There may be several possibilities for P, and thus several replacement sequences and several shortcuts $(P, \mathcal{X}_1)$, $(P, \mathcal{X}_2)$, ... representing them.

Example 10. The $(P_3, \{P_3, P_2, P_1\}, g)$-replacement sequence shown in Example 8 yields the shortcut $(P_3, \{P_3, P_2, P_1\})$. We can also find replacement sequences for P_1 and P_2, represented by the shortcuts $(P_1, \{P_1, P_2, P_3\})$ and $(P_2, \{P_1, P_2, P_3\})$.

Replacement Sequences of Larger Height. If we have shortcuts for all replacement sequences of height 0, we can construct replacement sequences of height 1 as follows. Such a sequence will contain firings of increasing clauses $P^f(x) \to P(f(x))$ w.r.t. some token w. This firing produces the token fw at all clauses having $P(x)$ or $P(f(x))$ on the left-hand side. This is a possible starting point for a $(P, \mathcal{X}, fw)$-replacement sequence of height 0.

If we have already computed a shortcut $(P, \mathcal{X})$, there is a firing sequence that deactivates the token fw and distributes it to all predicates of $\mathcal{X}$. This leaves us to consider the tokens that were created at decreasing clauses. These clauses must be of the form $Q(f(x)) \to Q^f(x)$ for $Q \in \mathcal{X}$ since the token begins with f and is distributed only to predicates in $\mathcal{X}$. We then simply fire these decreasing clauses, which gets us back to the original token w.

Thus, when looking for replacement sequences of height 1, we can use shortcuts as possibilities for the predicates P^f. Each shortcut $(P, \mathcal{X})$ yields a possibility $\{Q^f \mid Q \in \mathcal{X} \cap \mathcal{D}^f(\mathcal{C})\}$ for the increasing clause $P^f(x) \to P(f(x))$. If there is at least one shortcut $(P, \mathcal{X})$, then $\texttt{possibilities}(P^f)$ can now be non-empty. With this new definition of `possibilities`, we can compute shortcuts for replacement sequences of height 1, similar to the construction of replacement sequences of height 0. These yield more possibilities, which lead to shortcuts for replacement sequences of height 2, and so on.

The following procedure implements the computation of all possibilities for a predicate P w.r.t. a set $\mathcal{R}$ of previously computed shortcuts.

Algorithm 1 ($\texttt{possibilities}(\mathcal{C}, \mathcal{R}, P)$)**.**

Input: a normalized set $\mathcal{C}$ of propagation rules, a set $\mathcal{R}$ of shortcuts, and a predicate P
Output: the set of possibilities for P w.r.t. $\mathcal{C}$ and $\mathcal{R}$
  if $P = Q^f$ with $(Q, f) \in \mathcal{D}(\mathcal{C})$ **then**
    $\mathcal{L} \leftarrow \{\{Q_1^f, \ldots, Q_n^f\} \mid (Q, \mathcal{X}) \in \mathcal{R},\ \{Q_1, \ldots, Q_n\} = \mathcal{X} \cap \mathcal{D}^f(\mathcal{C})\}$
  else $\mathcal{L} \leftarrow \{\emptyset\}$
  for all $P(x) \to P_1(x) \vee \cdots \vee P_n(x) \in \mathcal{C}$ **do**
    $\mathcal{L} \leftarrow \{\mathcal{Y} \cup \{P_l\} \mid \mathcal{Y} \in \mathcal{L},\ l \in \{1, \ldots, n\}\}$
  return $\mathcal{L}$

For example, if we have the shortcut $(P_1, \{P_1, P_2, P_3\})$ from Example 10, then $\texttt{possibilities}(\mathcal{C}_1', \mathcal{R}, P_1^f)$ is $\{\{P_1^f, P_3^f, P_2\}\}$ instead of $\emptyset$.

Replacement Sequences for ε. To construct a replacement sequence for ε, we can use the same approach as above, but we also have to consider the ground clauses of $\mathcal{C}$. Since we only want to decide the existence of such a replacement sequence, we need not compute any shortcuts.

We call a predicate $P \in \mathcal{P}$ *good* if there is a (P, ε)-replacement sequence. All other predicates are *bad*. To decide whether A is good, we construct the set $\mathcal{B}$ of all bad predicates using the following procedure. The idea is that a predicate is bad whenever all its possibilities contain a bad predicate. This is similar to the emptiness test for looping automata on infinite trees [15].

Algorithm 2 (`isTerminating`$(\mathcal{C}, \mathcal{R})$)**.**

Input: a normalized set $\mathcal{C}$ of propagation rules and a set $\mathcal{R}$ of shortcuts
Output: `true` iff A is good w.r.t. $\mathcal{R}$
  $\mathcal{B}_0 \leftarrow \emptyset$, $k \leftarrow 0$
  repeat
    $\mathcal{B}_{k+1} \leftarrow \mathcal{B}_k$
      $\cup \{P \in \mathcal{P} \mid \exists P(x) \rightarrow P_1(x) \vee \cdots \vee P_n(x) \in \mathcal{C} : \{P_1, \ldots, P_n\} \subseteq \mathcal{B}_k\}$
      $\cup \{P \in \mathcal{P} \mid \exists P(a) \rightarrow P_1(a) \vee \cdots \vee P_n(a) \in \mathcal{C} : \{P_1, \ldots, P_n\} \subseteq \mathcal{B}_k\}$
      $\cup \{P^f \in \mathcal{P} \mid (P, f) \in \mathcal{D}(\mathcal{C}),\ \forall (P, \mathcal{X}) \in \mathcal{R}\ \exists Q \in \mathcal{X} \cap \mathcal{D}^f(\mathcal{C}) : Q^f \in \mathcal{B}_k\}$
    $k \leftarrow k + 1$
  until $\mathcal{B}_k = \mathcal{B}_{k-1}$
  return $A \notin \mathcal{B}_k$

Example 11. Consider the set $\mathcal{C}'_1$ from Example 2 and assume that no shortcuts are available. The predicates P_1^f, P_1^g, P_3^f, P_3^g, P_3^{gf}, and P_3^{gg} are immediately bad. Because of the clause $A(a) \rightarrow P_3^g(a)$, A is also bad. With the shortcuts computed in Example 10, the predicates P_3^g and A are no longer bad. This means that there is an (A, ε)-replacement sequence of height 1, as already seen in Example 6.

4 Deciding Termination

We can now formulate our main algorithm that decides whether $\mathcal{N}_\mathcal{C}$ terminates. It computes shortcuts representing replacement sequences of increasing height. The sets $\mathcal{R}_i$ are used to store all shortcuts computed so far. In each iteration, the algorithm checks whether these shortcuts already suffice to prove termination of $\mathcal{N}_\mathcal{C}$ using `isTerminating`$(\mathcal{C}, \mathcal{R}_i)$ (Alg. 2). If not, shortcuts for the next height are computed. If there are no new shortcuts, the algorithm stops and returns `false`, indicating that $\mathcal{N}_\mathcal{C}$ does not terminate.

Algorithm 3 (Main algorithm).

Input: a normalized set $\mathcal{C}$ of propagation rules
Output: `true` iff $\mathcal{N}_\mathcal{C}$ terminates
  $\mathcal{R}_0 \leftarrow \emptyset$, $i \leftarrow 0$
  repeat
    if `isTerminating`$(\mathcal{C}, \mathcal{R}_i)$ **then return** `true`
    $\mathcal{R}_{i+1} \leftarrow$ `nextShortcuts`$(\mathcal{C}, \mathcal{R}_i)$
    $i \leftarrow i + 1$
  until $\mathcal{R}_i = \mathcal{R}_{i-1}$
  return `false`

The procedure $\texttt{nextShortcuts}(\mathcal{C}, \mathcal{R})$ implements the computation of the shortcuts representing replacement sequences of the next height. It uses a set $\mathcal{T}$ of triples of the form (P, R_P, V_P), where R_P is the set of predicates reached so far starting from P, and $V_P \subseteq R_P$ contains the predicates that were already *visited*, i.e., for which all possibilities have been considered. Visiting Q corresponds to firing all clauses starting with $Q(x)$.

The computation of shortcuts for P starts with the triple $(P, \{P\}, \emptyset)$. In each step, we choose a triple $(P, R_P, V_P) \in \mathcal{T}$ that still contains an unvisited predicate $Q \in R_P \setminus V_P$ and consider its possibilities. For each $\mathcal{Y} \in \texttt{possibilities}(\mathcal{C}, \mathcal{R}, Q)$, we add $(P, R_P \cup \mathcal{Y}, V_P \cup \{Q\})$ to $\mathcal{T}$ since the predicates from $\mathcal{Y}$ have been reached and Q has just been visited. The original triple is removed from $\mathcal{T}$.

We continue this process until there are no more unvisited predicates. A triple (P, R_P, R_P) then yields the shortcut (P, R_P). We restrict the starting triples $(P, \{P\}, \emptyset)$ to satisfy $(P, f) \in \mathcal{D}(\mathcal{C})$ for some $f \in \mathcal{F}$ since only such predicates can be reached by an increasing clause.

Algorithm 4 ($\texttt{nextShortcuts}(\mathcal{C}, \mathcal{R})$)**.**

Input: a normalized set $\mathcal{C}$ of propagation rules and a set $\mathcal{R}$ of shortcuts
Output: a set $\mathcal{R}'$ of shortcuts for the next height
  $\mathcal{T} \leftarrow \{(P, \{P\}, \emptyset) \mid r \in \mathcal{F},\ (P, r) \in \mathcal{D}(\mathcal{C})\}$
  while there is $(P, R_P, V_P) \in \mathcal{T}$ with $R_P \setminus V_P \neq \emptyset$ **do**
    $\mathcal{T} \leftarrow \mathcal{T} \setminus \{(P, R_P, V_P)\}$
    choose Q from $R_P \setminus V_P$
    for all $\mathcal{Y} \in \texttt{possibilities}(\mathcal{C}, \mathcal{R}, Q)$ **do**
      $\mathcal{T} \leftarrow \mathcal{T} \cup \{(P, R_P \cup \mathcal{Y}, V_P \cup \{Q\})\}$
  return $\{(P, R_P) \mid (P, R_P, R_P) \in \mathcal{T}\}$

Example 12. Consider the set $\mathcal{C}_1'$ from Example 2. We describe the computation of $\texttt{nextShortcuts}(\mathcal{C}_1', \emptyset)$, which was already illustrated in Example 9. It starts with the triples $(P_1, \{P_1\}, \emptyset)$, $(P_3, \{P_3\}, \emptyset)$, $(P_3^f, \{P_3^f\}, \emptyset)$, and $(P_3^g, \{P_3^g\}, \emptyset)$, but we consider here only the first one.

The possibilities $\{P_2\}$ and $\{P_1^g\}$ for P_1 yield the triples $(P_1, \{P_1, P_2\}, \{P_1\})$ and $(P_1, \{P_1, P_1^g\}, \{P_1\})$. Since there is no shortcut $(P_1, \mathcal{X})$, the set of possibilities for P_1^g is empty and the second triple is removed. P_2 has the possibilities $\{P_3, P_1\}$ and $\{P_1^f, P_1\}$. One of the resulting triples is simply removed, leaving us with $(P_1, \{P_1, P_2, P_3\}, \{P_1, P_2\})$. Finally, P_3 is visited, resulting in $(P_1, \{P_1, P_2, P_3\}, \{P_1, P_2, P_3\})$, and thus in the shortcut $(P_1, \{P_1, P_2, P_3\})$.

In the following, we show that the computed shortcuts actually represent replacement sequences. More precisely, the shortcuts computed in the i-th iteration of the main loop of Alg. 3 represent all replacement sequences of height at most $i - 1$.

Lemma 13. *Let $i \geq 1$ be such that $\mathcal{R}_i$ was computed by Alg. 3, $(P, \mathcal{X}) \in \mathcal{R}_i$, and $w \in \mathcal{F}^+$. Then there is a $(P, \mathcal{X}, w)$-replacement sequence of height $\leq i - 1$.*

On the other hand, every replacement sequence of $\mathcal{N}_\mathcal{C}$ of height at most i corresponds to a shortcut computed in the $i+1$-th iteration of the algorithm. However, this shortcut does not need to have the same set $\mathcal{X}$ of reached predicates, but only a subset of it. The reason for this is that firings can always be applied, regardless of whether they are necessary to deactivate some token or not. This means that replacement sequences might contain irrelevant firings. However, Alg. 3 computes shortcuts in such a way that only necessary firings are considered, i.e., only possibilities for predicates that were already reached.

Lemma 14. *Consider the variant of Alg. 3 that never returns, but simply computes the sets $\mathcal{R}_i$ for all $i \geq 0$. Let $P \in \mathcal{D}^f(\mathcal{C})$. If there is a $(P, \mathcal{X}, fw)$-replacement sequence of height $\leq i$, then $(P, \mathcal{X}') \in \mathcal{R}_{i+1}$ for some $\mathcal{X}' \subseteq \mathcal{X}$.*

These results can be used to show that the algorithm is correct. If Alg. 3 returns `true`, then Lemma 13 allows us to construct a terminating firing sequence from the computed shortcuts. On the other hand, if there is such a sequence, Lemma 14 shows that Alg. 3 computes enough shortcuts to detect its existence.

Theorem 15. *Termination of propagation nets of the form $\mathcal{N}_\mathcal{C}$ for normalized sets $\mathcal{C}$ of propagation rules can be decided in time exponential in the size of $\mathcal{C}$.*

Proof (Sketch). We have $\mathcal{R}_{i-1} \subseteq \mathcal{R}_i$ after every step of Alg. 3. Since there are only exponentially many possible shortcuts and $\texttt{nextShortcuts}(\mathcal{C}, \mathcal{R}_i)$ takes at most exponential time, the overall runtime is also exponential. □

Corollary 16. *The existence of finite Herbrand models for finite sets of propagation rules can be decided in* ExpTime.

Proof. This follows from Theorem 5 and the reductions of Sects. 2 and 3. □

If all the clauses of $\mathcal{C}$ are *deterministic*, i.e., have at most one possibility, the propagation net $\mathcal{N}_\mathcal{C}$ is called *deterministic*. Then all places of $\mathcal{N}_\mathcal{C}$ have at most one outgoing arc and the algorithm runs in time polynomial in the size of $\mathcal{C}$. For every additional nondeterministic clause in the set $\mathcal{C}$, the runtime of the algorithm increases by an exponential factor due to the computation of all possibilities and all shortcuts in $\texttt{possibilities}(\mathcal{C}, \mathcal{R}, P)$ and $\texttt{nextShortcuts}(\mathcal{C}, \mathcal{R})$.

5 Hardness

To conclude the complexity analysis, we present a reduction from linear language equations to finite sets of propagation rules. The equations are of the form

$$S_0 \cup S_1 X_1 \cup \cdots \cup S_n X_n = T_0 \cup T_1 X_1 \cup \cdots \cup T_n X_n$$

for finite sets $S_0, \ldots, S_n, T_0, \ldots, T_n$ of words over an alphabet Σ. A *solution* assigns finite sets of words to the variables X_i such that the equation holds. Deciding whether such an equation has a solution is ExpTime-complete [1].

We can transform such equations into *flat linear language inclusions*

$$L_0X_0 \subseteq L_1X_1 \cup \cdots \cup L_nX_n$$

for $L_0, \ldots, L_n \subseteq \Sigma \cup \{\varepsilon\}$. By *flat* we mean that all coefficients contain only words of length at most 1. This can be achieved in polynomial time.

Example 17. Consider the equation $\{rs\} \cup \{s\}Y \cup X = \{r\}Y \cup \{s\}X \cup \{\varepsilon\}$.[2] If we abbreviate $\{r\}$ by r and introduce a new variable Z, we can equivalently write this problem using the flat equations $rZ \cup sY \cup X = rY \cup sX \cup \varepsilon$ and $Z = s$. These are then split into the following flat linear language inclusions:

$$\begin{aligned}\mathcal{I}_1 := \{&rZ \subseteq rY \cup sX \cup \varepsilon,\ sY \subseteq rY \cup sX \cup \varepsilon,\ X \subseteq rY \cup sX \cup \varepsilon,\ Z \subseteq s\\ &rY \subseteq rZ \cup sY \cup X,\ sX \subseteq rZ \cup sY \cup X,\ \varepsilon \subseteq rZ \cup sY \cup X,\ s \subseteq Z\}.\end{aligned}$$

To solve a finite set $\mathcal{I}$ of such inclusions, we translate $\mathcal{I}$ into a finite set $\mathcal{C}_\mathcal{I}$ of propagation rules that express the same restrictions as the inclusions. We will treat each $r \in \Sigma$ as a unary function symbol, each variable X occurring in $\mathcal{I}$ as a unary predicate. The intention behind $\mathcal{C}_\mathcal{I}$ is that a finite Herbrand model $\mathcal{H}$ of $\mathcal{C}_\mathcal{I}$ represents a solution θ of $\mathcal{I}$ with $\theta(X) = \{w \mid w(a) \in X^\mathcal{H}\}$.

To express an inclusion $L_0X_0 \subseteq L_1X_1 \cup \cdots \cup L_nX_n$ by clauses, we use the following idea. The clauses have to restrict the interpretation of the variables such that every word $w \in \Sigma^*$ occurring on the left-hand side of the inclusion also occurs on the right-hand side. For each word w occurring in L_0X_0, we make a case analysis based on the first letter of w. We create one clause for the case $w = \varepsilon$, and one clause for every possible first letter of w.

Example 18. Consider the inclusion $rZ \subseteq rY \cup sX \cup \varepsilon$ from Example 17. Every word w on its left-hand side has to begin with r, so the case analysis can be narrowed to one case. The corresponding clause is $Z(x) \to Y(x)$. Note that the terms sX and ε can never be responsible for this inclusion to be satisfied, and thus they are not represented in the clause.

Consider now another inclusion $X \subseteq rY \cup sX \cup \varepsilon$, which has to be split according to s, r, and ε. For the case that a word w on the left-hand side begins with r, we introduce the clause $X(r(x)) \to Y(x)$. Similarly, for s we obtain $X(s(x)) \to X(x)$. The case $w = \varepsilon$ is expressed by the clause $X(a) \to A(a)$, where A is a special predicate that is always interpreted as $\{a\}$.

Theorem 19. *Deciding the existence of finite Herbrand models for finite sets of propagation rules is* ExpTime*-hard.*

6 Summary and Conclusions

Viewed from a different perspective, Alg. 3 and the reduction from Sect. 5 yield a new ExpTime-algorithm for deciding solvability of linear language equations.

[2] This equation is equivalent to the $\mathcal{FL}_0$-unification problem $\forall r.\forall s.A \sqcap \forall s.Y \sqcap X \equiv^? \forall r.Y \sqcap \forall s.X \sqcap A$, where A is a constant and X, Y are variables (see [4] for details).

While the original decision procedure [1] constructs a tree automaton of exponential size and uses a linear-time emptiness test, our algorithm constructs a polynomial-size propagation net and uses an algorithm that is worst-case exponential, but exhibits a better behavior if the constructed set of propagation rules contains few nondeterministic clauses.

In future work, we want to modify the algorithm to actually compute solutions to the language equations and analyze the usefulness of these solutions; it may be desirable to output minimal solutions w.r.t. some order. We also want to implement the algorithm and compare it with an implementation of the naive tree automaton construction. To this end, we will have to design optimizations to our algorithm.

Another interesting open question is whether the presented approach can be applied to finite sets of arbitrary clauses with unary predicates, unary function symbols and constants. The formalism of propagation nets is certainly powerful enough to reflect this change, but the decision procedure also has to be adapted.

Acknowledgement. We would like to thank Prof. Franz Baader for helpful discussions and comments.

References

1. Baader, F., Narendran, P.: Unification of concept terms in description logics. J. Symb. Comput. 31(3), 277–305 (2001)
2. Baumgartner, P., Fuchs, A., de Nivelle, H., Tinelli, C.: Computing finite models by reduction to function-free clause logic. J. Appl. Log. 7(1), 58–74 (2009)
3. Birget, J.: State-complexity of finite-state devices, state compressibility and incompressibility. Math. Syst. Theory 26(3), 237–269 (1993)
4. Borgwardt, S., Morawska, B.: Finding finite Herbrand models. LTCS-Report 11-04, TU Dresden (2011), see http://lat.inf.tu-dresden.de/research/reports.html.
5. Chandra, A.K., Kozen, D.C., Stockmeyer, L.J.: Alternation. J. ACM 28(1), 114–133 (1981)
6. Comon, H., Dauchet, M., Gilleron, R., Löding, C., Jacquemard, F., Lugiez, D., Tison, S., Tommasi, M.: Tree automata techniques and applications. Available on: http://www.grappa.univ-lille3.fr/tata (2007)
7. Dreben, B., Goldfarb, W.D.: The Decision Problem: Solvable Classes of Quantificational Formulas. Addison-Wesley (1979)
8. Joyner Jr., W.H.: Resolution strategies as decision procedures. J. ACM 23(3), 398–417 (1976)
9. Ladner, R.E., Lipton, R.J., Stockmeyer, L.J.: Alternating pushdown and stack automata. SIAM J. Comput. 13(1), 135–155 (1984)
10. Leitsch, A.: The Resolution Calculus. Springer (1997)
11. Peltier, N.: Model building with ordered resolution: Extracting models from saturated clause sets. J. Symb. Comput. 36(1-2), 5–48 (2003)
12. Petri, C.A.: Kommunikation mit Automaten. Ph.D. thesis, Uni Bonn (1962)
13. Reisig, W.: Petri Nets: An Introduction. Springer (1985)
14. Slutzki, G.: Alternating tree automata. Theor. Comput. Sci. 41, 305–318 (1985)
15. Vardi, M.Y., Wolper, P.: Automata theoretic techniques for modal logics of programs (extended abstract). In: Proc. STOC'84. pp. 446–456. ACM (1984)
16. Zhang, J.: Constructing finite algebras with FALCON. J. Autom. Reasoning 17, 1–22 (1996)

Smart Testing of Functional Programs in Isabelle

Lukas Bulwahn

Technische Universität München

Abstract. We present a novel counterexample generator for the interactive theorem prover Isabelle based on a compiler that synthesizes test data generators for functional programming languages (e.g. ML, Haskell) from specifications in Isabelle. In contrast to naive type-based test data generators, the smart generators take the preconditions into account and only generate tests that fulfill the preconditions.

The smart generators are constructed by a compiler that reformulates the preconditions as logic programs and analyzes them with an enriched mode inference. From this inference, the compiler can construct the desired generators in the functional programming language.

Applying these test data generators reduces the number of tests significantly and enables us to find errors in specifications where naive random and exhaustive testing fail.

1 Introduction

Writing programs and specifications is an error-prone business, and testing is common practice to find errors and validate software. Being aware that testing cannot prove the absence of errors, formal methods are applied for safety- and security-critical systems. To ensure the correctness of programs, critical properties are guaranteed by a formal proof. Proof assistants are used to develop a proof with trustworthy sound logical inferences. Once one has completed the formal proof, the proof assistant certifies that the program meets its specification. But in the process of proving, errors could still be revealed and tracking these down by failed proof attempts is a tedious task for the user. Undoubtedly, testing is still fruitful on the way to quickly detect errors in programs and specifications while the user attempts to prove them. Modern interactive theorem provers therefore do not only provide means to prove properties, but also to *disprove* properties in the form of counterexample generators.

Without specifications, it is common practice to write manual test suites to check properties. However, having a formal specification at hand, we can automatically generate test data and check if the program fulfills its specification. Such an automatic specification-based testing technique for functional Haskell programs was introduced by the popular tool QuickCheck [8], which is based on random testing. The tool SmallCheck [19] also tests Haskell programs against its specification, but is based on exhaustive testing.

The interactive theorem prover Isabelle [22] provides a counterexample generator [3], which currently incorporates the two approaches, random and exhaustive testing, similar to QuickCheck and SmallCheck. It works well on specifications that have weak

N. Bjørner and A. Voronkov (Eds.): LPAR-18, LNCS 7180, pp. 153–167, 2012.

preconditions and properties in a form that is directly executable in the functional language. If the property to be tested includes a precondition, both approaches generate test data that seldom fulfill the precondition, and so most of the execution time for testing is spent generating useless test values and rejecting them.

Our new approach aims to only generate test data that fulfill the precondition. The test data generator for a given precondition is produced by a compiler[1] that analyzes preconditions and synthesizes a purely functional program that serves as generator. For this purpose, the compiler reformulates the preconditions as logic programs by translating formulas in predicate logic with quantifiers and recursive functions to Horn clauses. The compiler then analyzes the Horn clauses with a data flow analysis, which determines which values can be computed from other values and which values must be generated. From this analysis, the compiler constructs the desired generators. This way, a much smaller number of test cases suffices to exhaustively test a program against its specification. Consequently, we can find errors in specifications where random and naive exhaustive testing fail to find a counterexample in a reasonable amount of time.

After discussing related work (§1.1), we show examples that motivate the work on our new counterexample generator (§2). In the main part, we then describe key ideas of this counterexample generator, the preprocessing, the data flow analysis and compilation (§3 to §6). In the end, we evaluate our counterexample generator compared with the existing approaches (§7).

1.1 Related Work

The aforementioned Haskell tool QuickCheck has many descendants in interactive theorem provers, e.g., Agda/Alfa, ACL2, ACL2 Sedan, Isabelle and PVS, and in a variety of programming languages. QuickCheck uses test data generators that create random values to test the propositions. Random testing can handle propositions with strong preconditions only very poorly. To circumvent this, the user must manually write a test data generator that only produces values that fulfill the precondition. SmallCheck tests the propositions exhaustively for small values. It also handles propositions with strong preconditions poorly, but in practice handles preconditions better than QuickCheck because it gives preference to small values, and they tend to fulfill the commonly occurring preconditions more often. Lazy SmallCheck [19] uses partially-instantiated values and a refinement algorithm to simulate narrowing in Haskell. This is closely related to the work of Lindblad [13] and EasyCheck [7], based on the narrowing strategy in the functional logic programming language Curry [11]. This approach can cut the search space of possible values to check, if partially instantiated values already violate the precondition. The three approaches, QuickCheck (without manual test data generators), SmallCheck and Lazy SmallCheck, are examples of *black-box testing*, i.e., they do not consider the description of the precondition – they generate (partial) values and test the precondition.

The counterexample generators in Isabelle translate the conjecture and related definitions to an ML program, exploiting Isabelle's code generation infrastructure [10].

[1] Throughout the presentation, we use the term *compilation* with a very specific meaning: to designate our translation of Horn specifications in Isabelle into programs written in a functional programming language.

Employing this translation yields a very efficient evaluation: The ML runtime environment can check millions of test cases within seconds, which is thousands of times faster than evaluating within the prover. Like the existing counterexample generators in Isabelle, the new one also builds upon this translation. Previous work [2] on code generation focused on the *verification* of the transformation of Horn clauses to functional programs, whereas the focus of this work is the extension and application of the transformation for *counterexample generation*. Our new counterexample generator is a glass-box testing approach, i.e., it considers the description of the precondition and compiles a purely functional program that generates values that fulfill the precondition. We reported about this work in an early stage in [5]. Closely related to our work is the glass-box testing by Fischer and Kuchen [9] for functional logic programs, but they take advantage of narrowing and nondeterministic execution in Curry.

Another approach to finding values that fulfill the preconditions is to use a CLP(FD) constraint solver, as done by Carlier et al. [6]. A completely different approach to finding counterexamples is translating the specification to propositional logic and invoking a SAT solver, as performed by the Isabelle tools Refute [21] and Nitpick [4].

2 Motivation

The previously existing counterexample generators in Isabelle, which test with random values or exhaustively with small values, perform well on conjectures without preconditions. For example, for the invalid conjecture about lists[2]

$$\mathit{reverse}\,(\mathit{append}\,\mathit{xs}\,\mathit{ys}) = \mathit{append}\,(\mathit{reverse}\,\mathit{xs})\,(\mathit{reverse}\,\mathit{ys}),$$

the counterexample generators provide the counterexample $\mathit{xs} = [a_1]$ and $\mathit{ys} = [a_2]$ (for atoms $a_1 \neq a_2$) instantaneously. For conjectures of this kind, random and exhaustive testing are perfectly suited thanks to their lightweight nature.

But random and exhaustive testing generate values without analyzing the conjecture. This can lead to many vacuous test cases, as in this simple example:

$$\mathit{length}\,\mathit{xs} = \mathit{length}\,\mathit{ys} \wedge \mathit{zip}\,\mathit{xs}\,\mathit{ys} = \mathit{zs} \Longrightarrow \mathit{map}\,\mathit{fst}\,\mathit{zs} = \mathit{xs} \wedge \mathit{map}\,\mathit{snd}\,\mathit{zs} = \mathit{ys}$$

The random and exhaustive strategies first generate values for *xs*, *ys*, and *zs* in an unconstrained fashion and then check the premises, namely that *xs* and *ys* are of equal length and that *zs* is the list obtained by zipping *xs* and *ys* together. For the vast majority of variable assignments, the premises are not fulfilled, and the conclusion is left untested. Clearly, it is desirable to take the premises into account when generating values. For further illustration, we focus on a simpler valid conjecture about distinct lists:

$$\mathit{distinct}\,\mathit{xs} \Longrightarrow \mathit{distinct}\,(\mathit{tl}\,\mathit{xs})$$

The previously existing counterexample generator, testing exhaustively, produces the following test program in Standard ML to check the validity of this conjecture:

[2] We use common notations from functional programming languages: $[]$ and $x \cdot xs$ denote the two list constructors *Nil* and *Cons x xs*, lists, such as $(x \cdot (y \cdot (z \cdot \mathit{Nil})))$, are conveniently written as $[x, y, z]$. The tail of a list *xs* is obtained with $\mathit{tl}\,\mathit{xs}$, where $\mathit{tl}\,[] = []$ and $\mathit{tl}\,(x \cdot xs) = xs$. Furthermore, free variables are implicitly universally quantified.

$$\begin{aligned}
&\textbf{val}\ \mathit{generate\text{-}nat}\ \mathit{size}\ \mathit{chk} = \mathit{if}\ \mathit{size} = 0\ \mathit{then}\ \mathit{None}\ \mathit{else}\ \mathit{case}\ \mathit{chk}\ 0\ \mathit{of}\\
&\quad \mathit{Some}\ \mathit{xs} \Rightarrow \mathit{Some}\ \mathit{xs}\\
&\quad \mid \mathit{None} \Rightarrow \mathit{generate\text{-}nat}\ (\mathit{size} - 1)\ (\lambda \mathit{n}.\ \mathit{chk}\ (\mathit{n} + 1))
\end{aligned}$$

$$\begin{aligned}
&\textbf{val}\ \mathit{generate\text{-}list}\ \mathit{size}\ \mathit{chk} = \mathit{if}\ \mathit{size} = 0\ \mathit{then}\ \mathit{None}\ \mathit{else}\ \mathit{case}\ \mathit{chk}\ []\ \mathit{of}\\
&\quad \mathit{Some}\ \mathit{xs} \Rightarrow \mathit{Some}\ \mathit{xs}\\
&\quad \mid \mathit{None} \Rightarrow \mathit{generate\text{-}nat}\ (\mathit{size} - 1)\ (\lambda \mathit{x}.\ \mathit{generate\text{-}list}\ (\mathit{size} - 1)\\
&\qquad\qquad (\lambda \mathit{xs}.\ \mathit{chk}\ (\mathit{x} \cdot \mathit{xs})))
\end{aligned}$$

$$\textbf{val}\ \mathit{test}\ \mathit{xs} = \mathit{if}\ \mathit{distinct}\ \mathit{xs}\ \wedge \neg\ \mathit{distinct}\ (\mathit{tl}\ \mathit{xs})\ \mathit{then}\ \mathit{Some}\ \mathit{xs}\ \mathit{else}\ \mathit{None}$$

$$\textbf{val}\ \mathit{check}\ \mathit{size} = \mathit{generate\text{-}list}\ \mathit{size}\ (\lambda \mathit{xs}.\ \mathit{test}\ \mathit{xs})$$

The *check* function implements a simple generate-and-test loop. It uses the function *generate-list* that generates all possible lists (of natural numbers) up to a given bound and iteratively calls the *test* function to test the property at hand. It returns the found counterexample as an optional value, i.e., if the property holds for all values up to the given bound, the *check* function returns *None*, otherwise the counterexample is returned as result with *Some*.

Our new approach interleaves generation and checking in a way that avoids generating lists that are not distinct. From the definition of the *distinct* predicate,

$$\begin{aligned}
&\mathit{distinct}\ [] = \mathit{True}\\
&\mathit{distinct}\ (\mathit{x} \cdot \mathit{xs}) = (\mathit{x} \notin \mathit{set}\ \mathit{xs} \wedge \mathit{distinct}\ \mathit{xs}),
\end{aligned}$$

we can derive how to construct distinct lists: First, the empty list is distinct; secondly, larger distinct lists can be constructed taking a (shorter) distinct list and appending an element which is not in the list already to its front. This insight is reflected in the following test data generator:

$$\begin{aligned}
&\textbf{val}\ \mathit{generate\text{-}distinct}\ \mathit{size}\ \mathit{chk} = \mathit{if}\ \mathit{size} = 0\ \mathit{then}\ \mathit{None}\ \mathit{else}\ \mathit{case}\ \mathit{chk}\ []\ \mathit{of}\\
&\quad \mathit{Some}\ \mathit{xs} \Rightarrow \mathit{Some}\ \mathit{xs}\\
&\quad \mid \mathit{None} \Rightarrow \mathit{generate\text{-}distinct}\ (\mathit{size} - 1)\ (\lambda \mathit{xs}.\ \mathit{generate\text{-}nat}\ (\mathit{size} - 1)\\
&\qquad\qquad (\lambda \mathit{x}.\ \mathit{if}\ \mathit{x} \notin \mathit{set}\ \mathit{xs}\ \mathit{then}\ \mathit{chk}\ (\mathit{x} \cdot \mathit{xs})\ \mathit{else}\ \mathit{None}))
\end{aligned}$$

The function *generate-distinct* only generates and tests the given property with distinct lists. It constructs lists by applying the two rules mentioned above. With this generator at hand, we can check the conclusion more efficiently by:

$$\begin{aligned}
&\textbf{val}\ \mathit{test}\ \mathit{xs} = \mathit{if}\ \neg\ \mathit{distinct}\ (\mathit{tl}\ \mathit{xs})\ \mathit{then}\ \mathit{Some}\ \mathit{xs}\ \mathit{else}\ \mathit{None}\\
&\textbf{val}\ \mathit{check}\ \mathit{size} = \mathit{generate\text{-}distinct}\ \mathit{size}\ (\lambda \mathit{xs}.\ \mathit{test}\ \mathit{xs})
\end{aligned}$$

Using these *smart* test data generators reduces the number of tests, and as our evaluation (§7) shows, this allows us to explore test values of larger sizes where exhaustive testing cannot cope with the explosion of useless test values. More precisely, in our simple example, naive exhaustive testing cannot check all lists of size 15 within one hour, where the smart generator can easily explore all the lists up to this size within 30 seconds.

In the following sections, we describe how we synthesize these test data generators automatically from the precondition's definition.

3 Overview of the Tool

In this section, we present the overall structure of our counterexample generator, and motivate the key features and design decisions. The detached presentation of individual components is then discussed in the following three sections.

3.1 Design Decisions

QuickCheck and SmallCheck execute the program with concrete values. Testing with concrete values has the clear advantage of being natively supported by the functional programming language, in our case ML, and hence can be executed very fast. But testing with concrete values has the drawback that a large set of test inputs may exhibit indistinguishable executions. E.g., in our example about distinct lists, the lists $[1,1,2]$, $[1,1,3]$, $[1,1,4]$, … are all non-distinct because of the non-distinct prefix $[1,1]$, and hence testing the conjecture with all these lists succeeds without even checking its conclusion.

An alternative to testing functional programs is executing the program by a *needed narrowing strategy* [1], which executes the program symbolically as far as possible. It avoids symmetric executions, i.e., a set of input values that result in the same execution. It checks the conjecture for a set of values with one symbolic execution, reducing the number of tests. In our example, all the lists $[1,1,2]$, $[1,1,3]$, $[1,1,4]$, … can be treated immediately with one symbolic execution $1 \cdot (1 \cdot xs)$, where xs is a free variable representing any list of natural numbers. Symbolic executions usually result in a non-deterministic computation, which is implemented with a backtracking mechanism, as known from Prolog. This execution principle requires some overhead, which then causes symbolic testing to be slower than testing with concrete values, if the number of eliminated symmetric executions is too low to compensate for the execution's overhead.

Two circumstances contribute to the fact that symbolic executions frequently do not pay off in practice: First, large parts of the program are purely functional executions; nevertheless one inherits some overhead even in those parts of the execution. Second, if the conclusion is *hyperstrict*, i.e., requires checking with all test values, it incurs the overhead of symbolic executions, but ends up doing all executions necessarily with concrete values anyway.

Our test data generators aim to find a balance between *fast execution with concrete values* and *avoiding symmetric executions*. The test data generators produce concrete values during the execution, so that it can be translated directly into the target functional programming language.

For conjectures without preconditions, we enumerate all possible concrete values. This is quite effective, because usually there are only very few symmetric executions in that case. When preconditions occur in conjectures, test data generators only produce values fulfilling the precondition, and then test the conclusion. We find values fulfilling the precondition by an implementation that queries the precondition's predicate for all possible values up to some bound. The generator will enumerate possible values, similar to a query in Prolog, but returning only ground solutions, i.e., not using logical variables. The query is integrated in a lightweight fashion into the test program by a

compilation. It retains purely functional evaluations, detects values that can be computed by inferring data flow in the program (between variables), and combines it with generation of values if data flow cannot be inferred.

In the end, this static analysis and the compilation lead to test data generators for the preconditions. They discard useless test inputs before generating them, and keep the execution mechanism simple to target functional programming languages. If large parts of symmetric executions are avoided by the data flow analysis, these generators can explore the space of test input faster than symbolic and concrete executions.

3.2 Architecture

The counterexample generator performs these steps: As the original specification can be defined using various definitional mechanisms, the specification is preprocessed by a few simple syntactic transformations (§4) to Horn clauses. The core component, which was previously described in [2], consists of a static data flow analysis, the mode analysis (§5) and the code generator (§6). This core component only works on a syntactic subset of the Isabelle language, namely Horn clauses of the following form:

$$Q_1\ \overline{u}_1 \Longrightarrow \cdots \Longrightarrow Q_n\ \overline{u}_n \Longrightarrow P\ \overline{t}$$

In a premise $Q_i\ \overline{u}_i$, Q_i must be a predicate defined by Horn clauses and the terms $\overline{u}_i$ must be constructor terms, i.e., only contain variables or datatype constructors. Furthermore, we allow negation of atoms, assuming the Horn clauses to be stratified. If a premise obeys these restrictions, the core compiler infers modes and compiles functional programs for the inferred modes. If a premise has a different form, e.g., the terms contain function symbols, or a predicate is not defined by Horn clauses, the core compiler will treat them as side conditions. For side conditions, the mode analysis does not infer modes, but requires all arguments as inputs. Enriching the mode analysis, we mark unconstrained values to be generated. Once we have inferred modes for the Horn clauses, these are turned into test data generators in ML using non-deterministic executions and type-based generators.

4 Preprocessing

In this section, we sketch how specifications in predicate logic and functions are preprocessed to Horn clauses. A definition in predicate logic is transformed to a system of Horn clauses, based on the fact that a formula of the form $P\ \overline{x} = \exists \overline{y}.\ Q_1\ u_1 \wedge \cdots \wedge Q_n\ u_n$ can be soundly underapproximated by a Horn clause $Q_1\ u_1 \Longrightarrow \cdots \Longrightarrow Q_n\ u_n \Longrightarrow P\ \overline{x}$. Predicate logic formulas in a different form are transformed into the form above by a few logical rewrite rules in predicate logic. We rewrite universal quantifiers to negation and existential quantifiers, put the formula in negation normal form, and distribute existential quantifiers over disjunctions. In the process of creating Horn clauses, it is necessary to introduce new predicates for subformulas, as our Horn clauses do not allow disjunctions within the premises or nested expressions under negations. Furthermore, we take special care of *if*, *case* and *let*-constructions.

Example 1. The *distinct* predicate on lists is defined by the two equations,

$$distinct\ [] = True$$
$$distinct\ (x \cdot xs) = (x \notin set\ xs \wedge distinct\ xs)$$

In the preprocessing step, these are made to fit the syntactic restrictions of the core component, yielding the two Horn clauses:

$$distinct\ []$$
$$x \notin set\ xs \Longrightarrow distinct\ xs \Longrightarrow distinct\ (x \cdot xs)$$

To enable inversion of functions, we preprocess n-ary functions to $(n+1)$-ary predicates defined by Horn clauses, which enables the core compilation to inspect the definition of the function and leads to better synthesized test data generators. This is achieved by *flattening* a nested functional expression to a flat relational expression, i.e., a conjunction of premises in a Horn clause.

Example 2. We present how the *length* function for lists and a precondition containing this function are turned into relational expressions by flattening. The *length* of a list is defined by $length\ [] = 0$, and $length\ (x \cdot xs) = Suc\ (length\ xs)$[3]. We derive a corresponding relation $length_P$ with two Horn clauses:

$$length_P\ []\ 0$$
$$length_P\ xs\ n \Longrightarrow length_P\ (x \cdot xs)\ (Suc\ n)$$

The precondition $length\ xs = length\ ys$ is then transformed into

$$length_P\ xs\ n \wedge length_P\ ys\ n$$

In the new formulation, the constraint of the two lists having the same length is expressed by their shared variable n. This relational description helps our mode analysis to find a more precise data flow.

This well-known technique of flattening is similarly described by Naish [15] and Rouveirol [18]. We also support flattening of higher-order functions, which allows inversion of higher-order functions if the function argument is invertible.

5 Mode Analysis

In order to execute a predicate P, its arguments are classified as *input* or *output*, made explicit by means of *modes*. Modes can be inferred using a static analysis on the Horn clauses. Our mode analysis is based on Mellish [14]. There are more sophisticated mode analysis approaches, e.g., by using abstract domains [20] or by translating to a boolean constraint system [17]. But for our purpose, we can apply the simple mode analysis, because if the analysis does not discover a dataflow due to its imprecision, the overall process still leads to a test data generator.

[3] Natural numbers are defined by constructors 0 and *Suc*.

Modes. For a predicate P with k arguments, a *mode* is a particular dataflow assignment which follows the type of the predicate and annotates all arguments as input (i) or output (o), e.g., for $length_P$, $o \Rightarrow i \Rightarrow bool$ denotes the mode where the first argument is output, the last argument is input.

A *mode assignment* for a given clause $Q_1\ \overline{u}_1 \Longrightarrow \cdots \Longrightarrow Q_n\ \overline{u}_n \Longrightarrow P\ \overline{t}$ is a list of modes $M, M_1, \ldots M_n$ for the predicates $P, Q_1, \ldots, Q_n$. Let $FV(t)$ denote the set of free variables in a term t. Given a vector of arguments $\overline{t}$ and a mode M, the projection expression $\overline{t}\langle M\rangle$ denotes the list of all arguments in $\overline{t}$ (in the order of their occurrence) which are input in M.

Mode Consistency. Given a clause $Q_1\ \overline{u}_1 \Longrightarrow \cdots \Longrightarrow Q_n\ \overline{u}_n \Longrightarrow P\ \overline{t}$ a corresponding mode assignment $M, M_1, \ldots M_n$ is *consistent* if the chain of sets of variables $v_0 \subseteq \cdots \subseteq v_n$ defined by **(1)** $v_0 = FV(\overline{t}\langle M\rangle)$ and **(2)** $v_j = v_{j-1} \cup FV(\overline{u}_j)$ obeys the conditions **(3)** $FV(\overline{u}_j\langle M_j\rangle) \subseteq v_{j-1}$ and **(4)** $FV(\overline{t}) \subseteq v_n$. Mode consistency guarantees the possibility of a sequential evaluation of premises in a given order, where v_j represents the known variables after the evaluation of the j-th premise. Without loss of generality, we can examine clauses under mode inference modulo reordering of premises. For side conditions R, condition 3 has to be replaced by $FV(R) \subseteq v_{j-1}$, i.e., all variables in R must be known when evaluating it. This definition yields a check whether a given clause is consistent with a particular mode assignment.

Generator Mode Analysis. To generate values that satisfy a predicate, we extend the mode analysis in a genuine way: If the mode analysis cannot detect a consistent mode assignment, i.e., the values of some variables are not *constrained* after the evaluation of the premises, we allow the use of *generators*, i.e., the values for these variables are constructed by an *unconstrained* enumeration. In other words, we combine two ways to enumerate values, either driven by the *computation* of a predicate or by *generation* based on its type.

Example 3. Given a unary predicate R with possible modes $i \Rightarrow bool$ and $o \Rightarrow bool$ and the Horn clause $R\ x \Longrightarrow P\ x\ y$, classical mode analysis fails to find a consistent mode assignment for P with mode $o \Rightarrow o \Rightarrow bool$. To generate values for x and y fulfilling P, we combine computation and generation of values as follows: the values for variable x are built using R with $o \Rightarrow bool$; values for y are built by a generator.

This extension gives rise to a number of possible modes, because we actually drop the conditions **(3)** and **(4)** for the mode analysis. Instead, we use a heuristic to find a considerably good dataflow by locally selecting the optimal premise Q_j and mode M_j with respect to the following criteria:

1. minimize missing values, i.e., have $\left|FV(\overline{u}_j\langle M_j\rangle) - v_{j-1}\right|$ to be minimal;
2. use functional predicates with their functional mode;
3. use predicates and modes that do not require generators themselves;
4. minimize number of output positions;
5. prefer recursive premises.

Next, we motivate and illustrate these five criteria. In general, we would like to avoid generation of values and computations that could fail, and to restrain ourselves from enumerating any values that could possibly be computed. Hence, the first priority is to

use modes where the number of missing values is minimal. This way, we partly recover conditions **(3)** and **(4)** from the mode analysis.

Example 3 (continued). For mode M_1 for $R\ x$, one has two alternatives: generating values for x and then testing R with mode $i \Rightarrow bool$, or only generating values for x using R with $o \Rightarrow bool$. The first choice generates values and rejects them by testing; the latter only generates fulfilling values and is preferable. The analysis favors $o \Rightarrow bool$ to $i \Rightarrow bool$ due to criterion 1: for $v_0 = \{\}, \overline{u}_1 = x$ and $M_1 = i \Rightarrow bool$, $FV(\overline{u}_1\langle M_1\rangle) - v_0 = \{x\}$; whereas for $M_1 = o \Rightarrow bool$, $FV(\overline{u}_1\langle M_1\rangle) - v_0 = \{\}$. $|FV(\overline{u}_1\langle M_1\rangle) - v_0|$ is minimal for $M_1 = o \Rightarrow bool$.

Example 4. Consider a clause $R\ x\ y \Longrightarrow F\ x\ y \Longrightarrow P\ x\ y$ where R is a one-to-many relation and F is functional. R and F both allow modes $i \Rightarrow o \Rightarrow bool$ and $i \Rightarrow i \Rightarrow bool$. For $M = i \Rightarrow o \Rightarrow bool$, $R\ x\ y$ and $F\ x\ y$ can be evaluated in either order. Our criterion 2 induces preference for computing y with the functional computation $F\ x\ y$ and checking $R\ x\ y$, i.e., whether the one value for y can fulfill $R\ x\ y$ or not.

Criterion 3 induces avoiding the generation of values in the predicate to be invoked. Furthermore, we minimize output positions, e.g., we prefer checking a predicate (no output position) before computing some solution (one output position) as we illustrate by the following example:

Example 5. In a clause $R\ x\ y \Longrightarrow Q\ x \Longrightarrow P\ x\ y$ with mode $i \Rightarrow o \Rightarrow bool$ for R and P, and $i \Rightarrow bool$ for Q, we prefer $Q\ x$ before $R\ x\ y$, since computing values for y would be useless if $Q\ x$ fails. This ordering is enforced by criterion 4.

Finally, we prefer recursive premises – this leads to a bottom-up generation of values. Generating larger values for predicates from smaller values for the predicate is commonly preferable because it takes advantage of the structure of the preconditions.

Example 6. In a clause $P\ xs \Longrightarrow C\ xs \Longrightarrow P\ (x \cdot xs)$, $P\ xs$ is favored for generation of xs and $C\ xs$ for checking. Generating values for P, we apply the generator for P recursively and check the condition $C\ xs$ afterwards.

This "aggressive" mode analysis results in moded Horn clauses with annotations for generators of values. In summary, it does not only *discover* an existing dataflow, but helps to *create* a dataflow by filling the gaps with value generators.

6 Generator Compilation

In this section, we discuss the translation of the compiler from moded Horn clauses to functional programs. First, we present the building blocks of the compiler, the execution mechanism and the generators. Then, we sketch the compilation scheme by applying it to the introductory examples.

Monads for Non-deterministic Computations. We use continuations with type $\alpha\ cps$ to enumerate the (potentially infinite) set of values fulfilling the involved predicates – in other words, the constructed continuations will hold the *enumerated solutions*. We define *plus monad* operations describing non-deterministic computations. Depending on

our enumeration scheme, we employ three different plus monads: one for unbounded computations, and two others for depth-limited computations within positive and negative contexts, respectively.

A plus monad supports four operations: *empty*, *single*, *plus* and *bind*. It provides executable versions of basic set operations: $\mathit{empty} = \emptyset$, $\mathit{single}\ x = \{x\}$, $\mathit{plus}\ A\ B = A \cup B$ and $\mathit{bind}\ A\ f = \bigcup_{x \in A} f\ x$. Employing these operations in SML results in a Prolog-like execution strategy, with a depth-first search. This strategy is fine for user-initiated evaluations, but for counterexample generation, automatically generated values cause infinite computations escaped from the control of the user. To avoid being stuck in such a computation, we also employ a plus monad with a different carrier that limits the computation by a depth-limit. Evaluating predicates with a depth-limited computation, we must take special care of negation. We implement different behaviors for queries in different contexts: for positive contexts, we compute an underapproximation; for negative contexts, an overapproximation.

For positive contexts, we implement a plus monad with the type $\mathit{int} \to \alpha\ \mathit{cps}$ as carrier. The bind^+ operation checks the depth-limit and if reached, returns *empty*, which yields a sound underapproximation; otherwise it passes a decreased depth-limit to its argument. It is defined by:

$$\mathit{bind}^+\ xq\ f = (\lambda i.\ \mathit{if}\ i = 0\ \mathit{then}\ \mathit{empty}\ \mathit{else}\ \mathit{bind}\ (xq\ (i-1))\ (\lambda a.\ f\ a\ i))$$

In negative contexts, we must explicitly distinguish failure (no solution found) from reaching the depth limit. To signal reaching the depth-limit, we include an explicit element to model an *unknown* value (as a third truth value), and continue the computation with this value. This makes the monad carrier type be $\mathit{int} \to \alpha\ \mathit{option}\ \mathit{cps}$ where the option value *None* stands for unknown. If one computation reaches the depth-limit and another computation fails, then the overall computation fails; in other words *failure absorbs the unknown value* (which is consistent with a three-valued logic interpretation).

Because negative and positive occurrences of predicates are intermixed, in actual enumeration we have to combine the positive and negative monads – the bridge between them is performed by executable *not*-operations that handle the unknown value depending on the context. For instance, when applied to a solution enumeration of a negated premise, *unknown* is mapped to *false* (computation failure); this reflects the intuition that if we were not able to prove a negated premise $\neg Q\ x$ within a given depth-limit for x, then all we can soundly assume is that $Q\ x$ may hold; hence the computation cannot proceed further.

The compilation scheme builds abstractly on the monad structure interface and hence is employed for all three monads. For the rest of the presentation, we write *plus* and *bind* infix as $\sqcup$ and $\gg\!=$.

Type-Based Generators. If values cannot be computed, we enumerate them up to a given depth. To generate values of a specific type, we make use of type classes in Isabelle. More specifically we require that the involved types τ come equipped with an operation $\mathit{gen}\ \tau$, the generator for type τ that enumerates all values. For inductive datatypes τ with n constructors $C_1\ \tau_1^1 \ldots \tau_1^{m_1} \mid \ldots \mid C_n\ \tau_n^1 \ldots \tau_n^{m_n}$ we construct generators that enumerate values exhaustively up to depth d by the following scheme:

$$
\begin{array}{l}
\mathit{gen}\ \tau\ d = \\
\quad \mathit{if}\ d = 0\ \mathit{then}\ \mathit{empty}\ \mathit{else} \\
\quad\quad (\mathit{gen}\ \tau_1^1\ (d-1) \gg\!= (\lambda x^1.\ \mathit{gen}\ \tau_1^2\ (d-1) \gg\!= \ldots \gg\!= (\lambda x^{m_1-1}. \\
\quad\quad\quad \mathit{gen}\ \tau_1^{m_1}\ (d-1) \gg\!= (\lambda x^{m_1}.\ \mathit{single}\ (C_1\ x^1 \ldots x^{m_1}))) \ldots)) \sqcup \ldots \sqcup \\
\quad\quad (\mathit{gen}\ \tau_n^1\ (d-1) \gg\!= (\lambda x^1.\ \mathit{gen}\ \tau_n^2\ (d-1) \gg\!= \ldots \gg\!= (\lambda x^{m_n-1}. \\
\quad\quad\quad \mathit{gen}\ \tau_n^{m_n}\ (d-1) \gg\!= (\lambda x^{m_n}.\ \mathit{single}\ (C_n\ x^1 \ldots x^{m_n}))) \ldots))
\end{array}
$$

We already have seen concrete instances of these generators for lists and natural numbers, *generate-list* and *generate-nat* in §2 – although there, the scheme is disguised by the fact that we inlined the plus monad operations.

Compilation of Moded Clauses. The central idea underlying the compilation of a predicate P is to generate a function P^M for each mode M of P that, given a list of input arguments, enumerates all tuples of output arguments. The functional equation for P^M is the union of the output values generated by the characterizing clauses. Employing the data flow from the mode inference, the expressions for the clauses are essentially constructed as chains of type-based generators and function calls for premises, connected through *bind* and *case* expressions. All functions P^M are executable in ML, because they only employ the monad operations and pattern matching. The function P^M for the mode M with all arguments as output serves as test data generator for predicate P.

Example 7. For the predicate *distinct*, we can infer the mode $o \Rightarrow \mathit{bool}$: The first clause *distinct* [] allows the mode $o \Rightarrow \mathit{bool}$, as the empty list is just a constant value. The second clause allows the mode $o \Rightarrow \mathit{bool}$ by choosing modes for its premises, i.e., *distinct xs* with mode $o \Rightarrow \mathit{bool}$ and $x \notin \mathit{set}\ \mathit{xs}$ with mode $i \Rightarrow i \Rightarrow \mathit{bool}$. This is then compiled to a test data generator $\mathit{distinct}^o$ for lists of type τ:

$$
\begin{array}{l}
\mathit{distinct}^o\ \tau = \mathit{single}\ [] \sqcup \\
(\mathit{distinct}^o \gg\!= (\lambda \mathit{xs}.\ \mathit{gen}\ \tau\ \gg\!= (\lambda x.\mathit{if}\ x \notin \mathit{set}\ \mathit{xs}\ \mathit{then}\ \mathit{single}\ (x \cdot \mathit{xs})\ \mathit{else}\ \mathit{empty}))
\end{array}
$$

Instantiating τ to the natural numbers and unfolding the plus monad operators, the definition of $\mathit{distinct}^o$ yields the test data generator *generate-distinct* from section 2.

Example 8. For the precondition $\mathit{length}\ \mathit{xs} = \mathit{length}\ \mathit{ys} \wedge \mathit{zip}\ \mathit{xs}\ \mathit{ys} = \mathit{zs}$, we obtain the following moded clause:

- $\mathit{length}_P\ \mathit{xs}\ n$ with mode $o \Rightarrow o \Rightarrow \mathit{bool}$,
- $\mathit{length}_P\ \mathit{ys}\ n$ with mode $o \Rightarrow i \Rightarrow \mathit{bool}$,
- $\mathit{zip}_P\ \mathit{xs}\ \mathit{ys}\ \mathit{zs}$ with its functional mode $i \Rightarrow i \Rightarrow o \Rightarrow \mathit{bool}$

In other words, we enumerate lists with their corresponding length, and as we know the length of *xs*, we only enumerate lists *ys* of equal length, and finally we obtain *zs* by executing *zip xs ys*. The generator for this precondition then is:

$$
\begin{array}{l}
{\mathit{length}_P}^{oo} \gg\!= (\lambda(\mathit{xs}, n).\ {\mathit{length}_P}^{oi}\ n \\
\gg\!= (\lambda \mathit{ys}.\ \mathit{single}\ (\mathit{zip}\ \mathit{xs}\ \mathit{ys}) \gg\!= (\lambda \mathit{zs}.\ \mathit{single}\ (\mathit{xs}, \mathit{ys}, \mathit{zs}))))
\end{array}
$$

Table 1. Number of test cases for given sizes and preconditions

	size									
predicate	5	6	7	8	9	10	11	12	13	14
–	24	89	425	2,373	16,072	125,673	1,112,083	10,976,184	119,481,296	1,421,542,641
distinct	16	39	105	315	1,048	3,829	15,207	65,071	297,840	1,449,755
sorted	15	31	63	127	255	511	1,023	2,047	4,095	8,191

Unfolding the definitions of the plus monad operators and reducing the syntactic clutter, this leads to

$$\begin{array}{l} \mathit{if}\ d = 0\ \mathit{then}\ \mathit{None} \\ \mathit{else}\ \mathit{length}_P{}^{oo}\ (d-1)\ (\lambda(xs, n).\ \mathit{length}_P{}^{oi}\ n\ (d-1)\ (\lambda ys.\ c\ xs\ ys\ (\mathit{zip}\ xs\ ys)) \end{array}$$

The arguments c and d make the continuation and the limit on the depth of the computation explicit. The monad operations implicitly pass around the values for c and d.

7 Evaluation

To evaluate our approach, we compared the performance of the new approach against the three other existing testing approaches in Isabelle: random, exhaustive and narrowing-based testing. Random and exhaustive testing employ concrete values, whereas narrowing-based testing employs symbolic values. The narrowing-based testing in Isabelle is a descendant of Lazy SmallCheck, employing the same evaluation mechanism.

First, we compare their performance validating conjectures with simple preconditions. Table 1 shows the number of test cases up to a given size, and the number of test cases (for that size) for which the preconditions *distinct* and *sorted* hold. In other words, we measured the density of the search space if restricted by some precondition, compared to the unrestricted search space. For example, testing the proposition $\mathit{distinct}\ xs \Longrightarrow \mathit{distinct}\ (\mathit{tl}\ xs)$, the table shows how many test cases are generated by the naive exhaustive testing and by the smart test generators. This already gives a rough estimate on the possible improvement avoiding useless tests. Table 2 shows the run time[4] to validate properties with values up to a given size on some representative conjectures from Isabelle's library with the precondition *distinct* (D_1, D_2, D_3) and *sorted* (S_1, S_2, S_3):

- D_1: $\mathit{distinct}\ xs \Longrightarrow \mathit{distinct}\ (\mathit{tl}\ xs)$
- D_2: $\mathit{distinct}\ xs \Longrightarrow \mathit{distinct}\ (\mathit{remove1}\ x\ xs)$
- D_3: $\mathit{distinct}\ xs \Longrightarrow \mathit{distinct}\ (\mathit{zip}\ xs\ ys)$
- S_1: $\mathit{sorted}\ xs \Longrightarrow \mathit{sorted}\ (\mathit{remdups}\ xs)$
- S_2: $\mathit{sorted}\ xs \Longrightarrow \mathit{sorted}\ (\mathit{insort\text{-}insert}\ x\ xs)$
- S_3: $\mathit{sorted}\ xs \wedge i \leq j \wedge j < \mathit{length}\ xs \Longrightarrow \mathit{nth}\ xs\ i \leq \mathit{nth}\ xs\ j$

The numbers of D_1 indicate the improvement using the smart test generators for *distinct*. In case of D_2, a more representative conjecture of the Isabelle's theory of lists, we observe a similar behaviour. In D_3, the exhaustive testing does not enumerate all pairs of

[4] All tests ran on a Pentium DualCore P9600 2.6GHz with 4GB RAM using Poly/ML 5.4.1 and Ubuntu GNU/Linux 11.04

Table 2. Run time in seconds for given sizes – E, N, S denote exhaustive testing, narrowing, and smart generators, resp.; 0 denotes time < 50 ms, empty cells denote timeout after 1h; bold numbers indicate the lowest run time

		size																	
		8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
D_1	E	0	0	0	0.3	3.2	38	509											
	N	0	0.1	0.4	3.5	32	364												
	S	0	0	0	**0**	**0.2**	**0.7**	**3.8**	**22**	**135**	**862**								
D_2	E	0	0	0	0.4	3.8	45	589											
	N	0	0.1	0.5	4.0	37	395												
	S	0	0	0	**0.1**	**0.4**	**2.5**	**16**	**98**	**671**									
D_3	E	0.1	4.3	**155**															
	N	0.9	17	446															
	S	0.1	4.3	157															
S_1	E	0	0	0	0.2	2.7	31	404											
	N	0	0	0	0.1	0.1	0.1	0.2	0.4	0.9	2.0	4.6	10	23	52	115	257	565	1238
	S	0	0	0	**0**	**0**	**0**	**0**	**0**	**0**	**0.1**	**0.2**	**0.3**	**0.8**	**1.7**	**3.6**	**7.8**	**17**	**36**
S_2	E	0	0	0	0.2	2.5	29	381											
	N	0	0.1	0.1	0.1	0.1	0.2	0.4	0.8	1.8	3.9	8.8	20	44	98	218	286	1063	
	S	0	0	0	**0**	**0**	**0**	**0.1**	**0.1**	**0.2**	**0.5**	**1.1**	**2.5**	**5.5**	**12**	**28**	**61**	**135**	**292**
S_3	E	0	0	0	0.2	2.3	27	337											
	N	0	0	0.1	0.1	0.2	0.5	1.3	2.9	6.9	16	38	87	204	467	1064			
	S	0	0	0	**0**	**0**	**0.1**	**0.1**	**0.2**	**0.4**	**0.9**	**2.2**	**5.1**	**12**	**26**	**59**	**136**	**311**	**708**

lists for *xs* and *ys*, but only generates lists *ys* if the generated list *xs* is distinct. This simple optimisation already reduces the number of tests dramatically, i.e., only 0.025 percent of all tests are rejected by the precondition. Due to this fact, using the smart generator does not add any further significant improvement in the run time behaviour. Hence our smart generators perform practically the same to the exhaustive testing. Symbolic execution with narrowing performs worst due to its overhead in the execution in all three cases. On the very sparse precondition, *sorted xs*, the improvements with smart test data generators are even more apparent. For example, in S_1, naive exhaustive testing times out at size 15 (with a time limit of one hour), where the smart generators can still enumerate lists up to size 20 within a second. Narrowing performs better than exhaustive testing, but is still slower than the smart generators. These numbers show that the test data generators outperform the naive exhaustive testing and the symbolic narrowing-based testing.

Second, to show that this performance improvement also results in a direct gain for our users, we apply the counterexample generators on faulty implementations of typical functional data structures. We injected faults by adding typos into the correct implementations of the delete operation of 2-3 trees, AVL trees, and red-black trees. By adding typos, we create 10 different (possibly incorrect) versions of the delete operation for each data structure. On 2-3 trees, we check two invariants of the delete operation, keeping the tree balanced and ordered, i.e., $\mathit{balanced}\ t \Longrightarrow \mathit{balanced}\ (\mathit{delete}\ k\ t)$, and $\mathit{ordered}\ t \Longrightarrow \mathit{ordered}\ (\mathit{delete}\ k\ t)$. With the 10 versions, this yields 20 tests, on which we apply the different counterexample generators. Random testing (with 2,000 iterations

for each size) finds errors in 5, and exhaustive testing in 7 of 20 tests within thirty seconds. The smart generator finds errors in five more cases, uncovering 12 errors in the 20 tests; the narrowing approach performs equally well. In principle, exhaustive testing should find the errors eventually: so, on the five more intrinsic cases where the generators perform well, we increased the time for naive exhaustive testing to finally discover the fault – even after one hour of testing, exhaustive testing was not able to detect them. Also increasing the iterations for random testing to 20,000 iterations, it still discovers only five faults. This shows that using the test data generators in this case is clearly superior to naive exhaustive testing. In the eight cases, where all approaches found no fault, even testing more thoroughly for an hour did not reveal any further errors – most probably the property still holds, as the randomly injected faults do not necessarily affect the invariant.

On AVL trees, we observe a similar behaviour. When checking the two invariants of its delete operation on 10 modified versions, random testing uncovers 5, exhaustive testing 6, the smart generators and narrowing-based approach 11 errors in 20 cases. On red-black trees, the invariant was formulated in a way by the user that our data flow analysis cannot discover a reasonable ground data flow and therefore the synthesized generators perform very poorly. Here, the narrowing-based testing clearly benefits from its usage of symbolic values.

Beyond data structures, we also check a hotel key card system in Isabelle by Nipkow [16] which itself was inspired by a model from Jackson [12]. The faulty system contains a tricky man-in-the-middle attack, which is only uncovered by a trace of length 6. The formalisation uses a restrictive predicate that describes in which order specific events can occurs. Using the smart generators, we can find the attack within a few seconds. Synthesizing a test data generator for these valid traces requires the preprocessing techniques (§4), i.e., we can eliminate existential quantifiers, which render it non-executable for the random and exhaustive testing. Even after manual refinements to obtain an executable reformulation, random and exhaustive testing fail to find the counterexample within ten minutes of testing. The narrowing-based testing can handle the existential quantifiers in principle, but practically it performs badly with the deeply nested existential quantifiers in the specification, rendering it impossible to find the counterexample. After manual rewriting to eliminate the existentials, we also can obtain a counterexample with this approach within a few seconds.

8 Conclusion

This counterexample generator described in this paper is included in the current Isabelle development version and can be invoked by Isabelle's users to validate their specifications before proving them correct. It complements the existing naive exhaustive and narrowing-based testing techniques by combining the strengths of both: it reduces the number of tests, as narrowing-based testing does, and it executes tests very fast, as the naive exhaustive testing does.

Acknowledgements. I would like to thank Andrei Popescu, Sascha Boehme, Tobias Nipkow, Alexander Krauss, Thomas Tuerk, Brian Huffman, Jasmin Blanchette and the anonymous referees for comments on earlier versions of this paper.

References

1. Antoy, S., Echahed, R., Hanus, M.: A needed narrowing strategy. J. ACM 47, 776–822 (2000)
2. Berghofer, S., Bulwahn, L., Haftmann, F.: Turning Inductive into Equational Specifications. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 131–146. Springer, Heidelberg (2009)
3. Berghofer, S., Nipkow, T.: Random Testing in Isabelle/HOL. In: SEFM 2004, pp. 230–239. IEEE Computer Society (2004)
4. Blanchette, J.C., Nipkow, T.: Nitpick: A Counterexample Generator for Higher-Order Logic Based on a Relational Model Finder. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 131–146. Springer, Heidelberg (2010)
5. Bulwahn, L.: Smart test data generators via logic programming. In: ICLP 2011 (Technical Communications). Leibniz Int. Proc. in Informatics, vol. 11, pp. 139–150. Schloss Dagstuhl, Leibniz-Zentrum für Informatik ((2011)
6. Carlier, M., Dubois, C., Gotlieb, A.: Constraint Reasoning in FocalTest. In: ICSOFT 2010 (2010)
7. Christiansen, J., Fischer, S.: EasyCheck — Test Data for Free. In: Garrigue, J., Hermenegildo, M.V. (eds.) FLOPS 2008. LNCS, vol. 4989, pp. 322–336. Springer, Heidelberg (2008)
8. Claessen, K., Hughes, J.: QuickCheck: A lightweight tool for random testing of Haskell programs. In: ICFP 2000, pp. 268–279. ACM SIGPLAN (2000)
9. Fischer, S., Kuchen, H.: Systematic generation of glass-box test cases for functional logic programs. In: PPDP 2007, pp. 63–74. ACM (2007)
10. Haftmann, F., Nipkow, T.: Code Generation via Higher-Order Rewrite Systems. In: Blume, M., Kobayashi, N., Vidal, G. (eds.) FLOPS 2010. LNCS, vol. 6009, pp. 103–117. Springer, Heidelberg (2010)
11. Hanus, M.: Multi-paradigm Declarative Languages. In: Dahl, V., Niemelä, I. (eds.) ICLP 2007. LNCS, vol. 4670, pp. 45–75. Springer, Heidelberg (2007)
12. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press (2006)
13. Lindblad, F.: Property directed generation of first-order test data. In: The Eigth Symposium on Trends in Functional Programming (2007)
14. Mellish, C.S.: The automatic generation of mode declarations for Prolog programs. Technical Report 163, Department of Artificial Intelligence (1981)
15. Naish, L.: Adding Equations to NU-Prolog. In: Małuszyński, J., Wirsing, M. (eds.) PLILP 1991. LNCS, vol. 528, pp. 15–26. Springer, Heidelberg (1991)
16. Nipkow, T.: Verifying a Hotel Key Card System. In: Barkaoui, K., Cavalcanti, A., Cerone, A. (eds.) ICTAC 2006. LNCS, vol. 4281, pp. 1–14. Springer, Heidelberg (2006)
17. Overton, D., Somogyi, Z., Stuckey, P.J.: Constraint-based mode analysis of mercury. In: PPDP 2002, pp. 109–120. ACM (2002)
18. Rouveirol, C.: Flattening and Saturation: Two Representation Changes for Generalization. Mach. Learn. 14(2), 219–232 (1994)
19. Runciman, C., Naylor, M., Lindblad, F.: SmallCheck and Lazy SmallCheck: automatic exhaustive testing for small values. In: Haskell 2008, pp. 37–48. ACM (2008)
20. Smaus, J.G., Hill, P.M., King, A.: Mode Analysis Domains for Typed Logic Programs. In: Bossi, A. (ed.) LOPSTR 1999. LNCS, vol. 1817, pp. 82–101. Springer, Heidelberg (2000)
21. Weber, T.: Bounded model generation for Isabelle/HOL. In: PDPAR 2004. Electronic Notes in Theoretical Computer Science, vol. 125(3), pp. 103–116. Elsevier (2005)
22. Wenzel, M., Paulson, L.C., Nipkow, T.: The Isabelle Framework. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 33–38. Springer, Heidelberg (2008)

Monitor-Based Statistical Model Checking for Weighted Metric Temporal Logic*

Peter Bulychev[1], Alexandre David[1], Kim Guldstrand Larsen[1], Axel Legay[2], Guangyuan Li[3], Danny Bøgsted Poulsen[1], and Amelie Stainer[4]

[1] Computer Science, Aalborg University, Denmark
[2] INRIA/IRISA, Rennes Cedex, France
[3] State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, P.R. of China
[4] University of Rennes 1, Rennes, France

Abstract. We present a novel approach and implementation for analysing weighted timed automata (WTA) with respect to the weighted metric temporal logic (WMTL$_{\leq}$). Based on a stochastic semantics of WTAs, we apply statistical model checking (SMC) to estimate and test probabilities of satisfaction with desired levels of confidence. Our approach consists in generation of deterministic monitors for formulas in WMTL$_{\leq}$, allowing for efficient SMC by run-time evaluation of a given formula. By necessity, the deterministic observers are in general approximate (over- or under-approximations), but are most often exact and experimentally tight. The technique is implemented in the new tool Casaal. that we seamlessly connect to Uppaal-smc. in a tool chain. We demonstrate the applicability of our technique and the efficiency of our implementation through a number of case-studies.

1 Introduction

Model checking (MC) [14] is a widely used approach to guarantee correctness of a system by checking that its model satisfies a given property. A typical model checking algorithm explores a state space of a model and tries to prove or disprove that the property holds on the model.

Despite a large and growing number of successful applications in industrial case studies, the MC approach still suffers from the so-called state explosion problem. This problem manifests itself in the form of unmanageably large state spaces of models with large number of components (i.e. number of variables, parallel components, etc). The situation is even worse when a system under analysis is hybrid (i.e. it possesses both continuous and discrete behaviors), because a state space of such models may lack finite representation [2]. Another challenge for MC is to analyze stochastic systems, i.e. systems with probabilistic assumptions for their behavior.

* The paper is supported by the Danish National Research Foundation, the National Natural Science Foundation of China (Grant No.61061130541) for the Danish-Chinese Center for Cyber Physical Systems and VKR Center of Excellence MT-LAB.

N. Bjørner and A. Voronkov (Eds.): LPAR-18, LNCS 7180, pp. 168–182, 2012.

One of the ways to avoid these complexity and undecidability issues is to use statistical model checking (SMC) approach [19]. The main idea of the latter is to observe a number of simulations of a model and then use results from statistics (e.g. sequential analysis) to get an overall estimate of a system behavior.

In the present paper we consider a problem of computing the probability that a random run of a given weighted timed automaton (WTA) satisfies a given weighted metric temporal logic formula ($\mathrm{WMTL}_{\leq}$). Solving this problem is of great practical interest since WTA are as expressive as general linear hybrid automata [2], a formalism which has proved to be very useful for modeling real-world hybrid and real-time systems. Moreover, $\mathrm{WMTL}_{\leq}$ [7] is not only a weighted extension of the well established LTL but can also be seen as an extension of MTL [15] to hybrid systems. However, the model checking problem for $\mathrm{WMTL}_{\leq}$ is known to be undecidable [7], and in our paper we propose an approximate approach that computes a confidence interval for the probability. In most of the cases this confidence interval can be made arbitrary small.

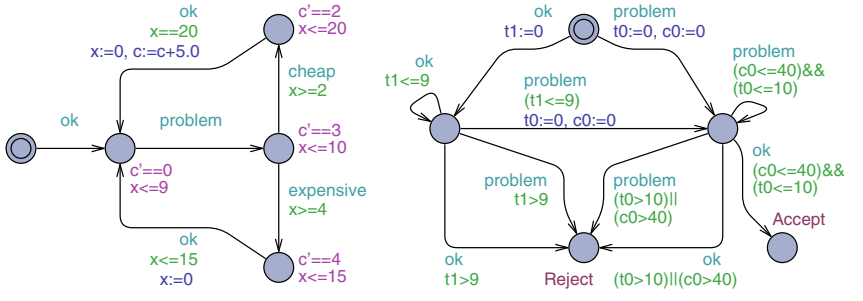


Fig. 1. A model (left) and deterministic monitor (right) for the repair problem

As an example consider a never-ending process of repairing problems [7], whose Weighted Timed Automata model is depicted at Fig. 1 (left). The repair of a problem has a certain cost, captured in the model by the clock c[1]. As soon as a problem occurs (modeled by the transition labeled by action `problem`) the value of `c` grows with rate 3, until actual `cheap` (rate 2) or `expensive` (rate 4) repair is taking place. Clock `x` grows with rate 1 (it's default behavior unless other rate is specified). Being a Weighted Timed Automaton, this model is equipped with a natural stochastic semantics [10] with a uniform choice on possible discrete transitions and uniformly selected delays in locations.

Now consider that we want to express the property that a path goes from `ok` back to itself in time less than 10 time units and cost less than 40. This can be formalized by the following $\mathrm{WMTL}_{\leq}$ formula:

$$\mathtt{ok}\, \mathsf{U}^{\tau}_{\leq 9}(\mathtt{problem} \wedge (\neg \mathtt{ok}\ \mathsf{U}^{\tau}_{\leq 10}\ \mathtt{ok}) \wedge (\neg \mathtt{ok}\ \mathsf{U}^{c}_{\leq 40}\ \mathtt{ok}))$$

[1] we will (mis)use the term "clock" from timed automata, though in the setting of WTAs the clocks are really general real-valued variables.

Here, the MITL$_{\leq}$-formula $\varphi_1 \mathsf{U}^{c}_{\leq d} \varphi_2$ is satisfied by a run if φ_1 is satisfied on the run until φ_2 is satisfied, and this will happen before the value of the clock c increases with more than d starting from the beginning of the run (τ is a special clock that always grows with rate 1).

In order to estimate the probability that a random run of a model satisfies a given property, our approach will first construct deterministic monitoring weighted timed automata for this property. In fact, it is not always possible to construct an *exact* deterministic observer for a property, thus our tool can result in deterministic under- and over-approximations. For our example, the tool constructed the exact deterministic monitor presented in Fig. 1 (right). Here rates of a monitoring automaton are defined by the rates of the automaton being monitored, i.e. the rate of `c0` is equal to the rate of `c`.

The constructed monitoring WTA permits the SMC engine of Uppaal to use run-time evaluation of the property in order to efficiently estimate the probability that runs of the models satisfy the given property. In our example the Uppaal-smc. returns the 95% confidence interval $[0.215, 0.225]$. If none of the under- and over-approximation monitors are exact, then we use both of them to compute the confidence interval.

Our contribution is twofold. First, we are the first to extend statistical model checking to the WMTL$_{\leq}$ logic. The closest logic that has been studied so far is the strictly less expressive MITL$_{\leq}$, that does not allow to use energy clocks in the U operator. Second, our monitor-based approach works on-the-fly and can terminate a simulation as soon as it may conclude that a formula will be satisfied (or violated) by the simulation. Other statistical model checking algorithms that deal with linear-time properties (cf. [1,18,19,20]) require a posterior (and expensive) check after a complete simulation of a fixed duration has been generated.

2 Weighted Timed Automata and Metric Temporal Logic

In this section we describe weighted timed automata (WTA) and weighted metric temporal logic (WMTL$_{\leq}$) as our modeling and specification formalisms. A notion of monitoring weighted timed automata (MWTA) is used to define automatically constructed (deterministic) observers for WMTL$_{\leq}$ properties.

2.1 Weighted Timed Automata

Let C be a set of clocks. A clock bound over C has the form $c \sim n$ where $c \in C$, $\sim \in \{<, \leq, \geq, >\}$ and $n \in \mathbb{Z}_{\geq 0}$. We denote the set of all possible clock bounds over C by $\mathcal{B}(C)$. A valuation over C is a function $v : C \to \mathbb{R}_{\geq 0}$, and a rate vector is a function $r : C \to \mathbb{Q}$. We let $\mathcal{V}(C)$ ($\mathcal{R}(C)$, respectively) to be all clock valuations (rates) over C.

Definition 1. *A* Weighted Timed Automaton[2] *(WTA) over alphabet* $\mathcal{A}$ *is a tuple* $(L, \ell_0, C_i, C_o, E, W, I, R)$ *where:*

- L *is a finite set of locations,*
- $\ell_0 \in L$ *is the initial location,*
- C_i *and* C_o *are finite set of real-valued variables called* internal *clocks and* observable *clocks, respectively,*
- $E \subseteq L \times \mathcal{A} \times 2^{\mathcal{B}(C_i \cup C_o)} \times 2^{C_i} \times L$ *is a finite set of edges,*
- $W : E \to \mathcal{R}(C_i \cup C_o)$ *assigns weights to edges, weights of observable clocks should be non-negative (i.e.* $W(e)(c) \geq 0$ *for any* $e \in E$ *and* $c \in C_o$*),*
- $I : L \to 2^{\mathcal{B}(C_i \cup C_o)}$ *assigns an invariant to each location,*
- $R : L \to \mathcal{R}(C_i \cup C_o)$ *assigns rates to the clocks in each location, rates of observable clocks should be non-negative.*

If $\delta \in \mathbb{R}_{\geq 0}$, then we define $v + \delta$ to be equal to the valuation v' such, that for all $c \in C$ we have $v'(c) = v(c) + \delta$. If r is a rate vector, then $v + r \cdot \delta$ is the valuation v' such that for all clocks c in C, $v'(c) = v(c) + r(c) \cdot \delta$. The valuation that assigns zero to all clocks is denoted by $\mathbf{0}$. Given $Y \subseteq C$, $v[Y = 0]$ is the valuation equal to $\mathbf{0}$ over Y and equal to v over $C \setminus Y$. We say, that a valuation v satisfies a clock bound $b = c \sim n$ (denoted $v \vDash b$), iff $v(c) \sim n$. A valuation satisfies a set of clock bounds if it satisfies all of them or this set is empty. A state (l, v) of a WTA consists of a location $l \in L$ and a valuation $v \in \mathcal{V}(C_i \cup C_o)$. In particular, the initial state of the WTA is $(\ell_0, \mathbf{0})$. From a state a WTA can either delay for some time δ or it can perform a discrete action a, the rules are given below:

- $(\ell, v) \xrightarrow{\delta} (\ell, v')$ if $v' = v + R(\ell) \cdot \delta$ and $v' \vDash I(\ell)$.
- $(\ell, v) \xrightarrow{a} (\ell', v')$ if $v \vDash g$ and there exists an edge $e \in E$ such that $e = (\ell, g, a, Y, r, \ell')$, $v' = v[Y = 0] + W(e) \cdot 1$ and $v' \vDash I(\ell')$.

An (infinite) weighted word over actions $\mathcal{A}$ and clocks C is a sequence $w = (a_0, v_0)(a_1, v_1) \ldots$ of pairs of actions $a_i \in \mathcal{A}$ and valuations $v_i \in \mathcal{V}(C)$. For $i \geq 0$, we denote by w^i the weighted word $w^i = (a_i, v_i)(a_{i+1}, v_{i+1}) \ldots$.

A WTA $A = (L, \ell_0, C_i, C_o, E, W, I, R)$ over $\mathcal{A}$ generates a weighted word $w = (a_0, v_0)(a_1, v_1) \ldots$ over actions $\mathcal{A}$ and *observable* clocks C_o, iff $v_0 = \mathbf{0}$ and there exists a sequence of transitions

$$(\ell_0, v'_0) \xrightarrow{\delta_0} (\ell_0, v''_0) \xrightarrow{a_0} (\ell_1, v'_1) \xrightarrow{\delta_1} \ldots \xrightarrow{a_n} (\ell_{n+1}, v'_{n+1}) \ldots,$$

and for any i the valuation v_i is a projection of v'_i to C_o, i.e. $v_i(c)$ is equal to $v'_i(c)$ for any observable clock $c \in C_o$.

Note, that since *observable* clocks are never reset and grow only with positive rates, the values of observable clocks can not decrease in a word generated by a

[2] In the classical notion of priced timed automata [6,4] cost-variables (e.g. clocks where the rate may differ from 1) may not be referenced in guards, invariants or in resets, thus making e.g. optimal reachability decidable. This is in contrast to our notion of WTA, which is as expressive as linear hybrid systems [8].

WTA. In fact, we restrict ourselves to WTAs that generate cost-divergent words (i.e. for any observable clock c and constant $k \in \mathbb{R}_{\geq 0}$ there is v_i such, that $v_i(c) > k$). If we consider that the WTA in Fig. 1(left) has only one observable clock $\mathtt{c}$, then this WTA can generate a weighted word $(\mathtt{ok}, \{\mathtt{c} \mapsto 2.0\})$, $(\mathtt{problem}, \{\mathtt{c} \mapsto 3.1\})$, $(\mathtt{cheap}, \{\mathtt{c} \mapsto 4.2\}), \ldots$.

We let $\mathcal{L}(A)$ denote the set of all weighted words generated by an WTA A and refer to it as the language of A.

A network of Weighted Timed Automata is a parallel composition of several WTA that have disjoint set of clocks and same set of actions $\mathcal{A}$. The automata are synchronized regarding discrete transitions such that if one automata performs a transition $\xrightarrow{a}$ all other also must perform an $\xrightarrow{a}$ transition. The notion of language recognized by WTA is naturally extended to the networks of Weighted Timed Automata.

In [10] we proposed a stochastic semantics for WTA, i.e. a probability measure over the set of accepted weighted words $\mathcal{L}(A)$. The non-determinism regarding discrete transitions for a single WTA is resolved using a uniform probabilistic choice among the possible transitions. Non-determinism regarding delays from a state (ℓ, v) of a single WTA is resolved using a density function $\mu_{(\ell,v)}$ over delays in $\mathbb{R}_{\geq 0}$ being either a uniform or an exponential distribution depending on whether the invariant of ℓ is empty or not.
The stochastic semantics for networks of WTA is then given in terms of repeated races between the component WTAs of the network: before a discrete transition each WTA chooses a delay according to its delay density function; then the WTA with a smallest delay wins the race and chooses probabilistically the action that the network must perform.

2.2 Monitoring Weighted Timed Automata

A monitoring weighted timed automaton (MWTA) A_M is a special kind of WTA used to define allowed behavior of a given WTA A (or a network of WTAs): a weighted word generated by A is fed as input to A_M for acceptance. For this, the actions of A and A_M coincide and there is a correspondence between the monitoring clocks of A_M and the observable clocks A ensuring that corresponding clocks grow with the same rate.

Definition 2. *A* Monitoring Weighted Timed Automaton *(MWTA) over the clocks C and the actions $\mathcal{A}$ is a tuple $(L, \ell_0, \ell_a, C_M, E, m)$ where:*

- *L is a finite set of locations,*
- *$\ell_0 \in L$ is the initial location,*
- *$\ell_a \in L$ is the accepting location,*
- *C_M is a finite set of* local *clocks,*
- *$E \subseteq L \times \mathcal{A} \times 2^{\mathcal{B}(C_M)} \times 2^{C_M} \times L$ is a finite set of edges,*
- *$m : C_M \to C$ gives the correspondence of local clocks and C.*

An MWTA is called deterministic if for any location $l \in L$, action $a \in \mathcal{A}$ and valuation $v \in \mathcal{V}(C_M)$ there exist not more than one edge $(l, a, g, Y, l') \in E$ such that $v \models g$.

An MWTA $A_M = (L, \ell_0, \ell_a, C_M, E, m)$ over clocks C and actions $\mathcal{A}$ accepts a weighted word $(a_0, v_0)(a_1, v_1)\ldots$ over the same C and $\mathcal{A}$, iff there exists a finite sequence $(l_0, v'_0), (l_1, v'_1), \ldots, (l_n, v'_n)$ of states of A_M such, that:

- $v'_0(c) = v_0(m(c))$ for any clock $c \in C_M$,
- for any i there exists an edge $(l_i, a_i, g_i, Y_i, l_{i+1}) \in E$ such, that:
  - $v'_i \models g_i$ and
  - for every clock $c \in C_M$, if $c \in Y_i$ then $v'_{i+1}(c) = 0$, and otherwise $v'_{i+1}(c) = v'_i(c) + (v_{i+1}(m(c)) - v_i(m(c)))$,
- $l_n = l_a$ is the accepting location of A.

Thus, after reading an element of an input weighted word, a *local* clock c of the MWTA is either reset, or it grows with the same rate as the corresponding clock $m(c)$ in the input word.

2.3 Weighted Metric Temporal Logic WMTL$_{\leq}$

Definition 3. *[7] A WMTL$_{\leq}$ formula φ over atomic propositions P and clocks C is defined by the grammar*

$$\varphi ::= p \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid O\varphi \mid \varphi_1 \mathsf{U}^c_{\leq d} \varphi_2$$

where $p \in P$, $d \in \mathbb{N}$, and $c \in C$.

Let *false* be an abbreviation for $(p \wedge \neg p)$, and *true* be an abbreviation for $\neg false$. The other commonly used operators in WMTL$_{\leq}$ can be defined by the following abbreviations: $(\varphi_1 \vee \varphi_2) = \neg(\neg\varphi_1 \wedge \neg\varphi_2)$, $(\varphi_1 \rightarrow \varphi_2) = (\neg\varphi_1) \vee \varphi_2$, $\Diamond^c_{\leq d}\varphi = true\, \mathsf{U}^c_{\leq d}\varphi$, $\Box^c_{\leq d}\varphi = \neg\Diamond^c_{\leq d}\neg\varphi$, and $\varphi_1 \mathsf{R}^c_{\leq d}\varphi_2 = \neg(\neg\varphi_1 \mathsf{U}^c_{\leq d}\neg\varphi_2)$, where R is the "release" operator. We also assume, that there always exists a special clock $\tau \in C$ (that grows with a rate 1 in an automaton being monitored).

Assuming that P are atomic propositions over actions $\mathcal{A}$, WMTL$_{\leq}$ formulas are interpreted over weighted words (we use the pointwise semantics). For a given weighted word $w = (a_0, v_0)(a_1, v_1)(a_2, v_2)\ldots$ over $\mathcal{A}$ and C and WMTL$_{\leq}$ formula φ over P and C, the satisfaction relation $w^i \models \varphi$ is defined inductively:

1. $w^i \models p$ iff $a_i \models p$
2. $w^i \models \neg\varphi$ iff $w^i \not\models \varphi$
3. $w^i \models O\varphi$ iff $w^{i+1} \models \varphi$
4. $w^i \models \varphi_1 \wedge \varphi_2$ iff $w^i \models \varphi_1$ and $w^i \models \varphi_2$
5. $w^i \models \varphi_1 \mathsf{U}^c_{\leq d}\varphi_2$ iff there exists j such that $j \geq i$, $w^j \models \varphi_2$, $v_j(c) - v_i(c) \leq d$, and $w^k \models \varphi_1$ for all k with $i \leq k < j$.

We say, that a weighted word w satisfies φ, iff $w^0 \models \varphi$, and denote by $\mathcal{L}(\varphi)$ the set of all weighted words that are satisfied by φ. φ_1 and φ_2 are equivalent if they are satisfied by the same weighted words, in which case we write $\varphi_1 \equiv \varphi_2$.

Given the stochastic semantics of a WTA A, and semantics of WMTL$_{\leq}$ formula φ, we can define $Pr[A \models \varphi]$ to be the probability that a random run of A satisfies φ. This probability is well-defined because $\mathcal{L}(A) \cap \mathcal{L}(\varphi)$ is a countable union and intersection of measurable sets and thus it is measurable itself.

3 From Formulas to Monitors

In this section we present a novel procedure for translating WMTL$_{\leq}$ formulas into equivalent MWTA monitors, providing an essential and efficient component of our tool-chain. However, to enable monitor-based, statistical model checking it is essential that the generated MWTA is deterministic. Unfortunately, this might not always be possible as there are WMTL$_{\leq}$ formulas for which no equivalent deterministic MWTA exist[3]. As a remedy, we describe how basic syntactic transformations prior to translation allow us to obtain deterministic over- and under-approximating MWTAs for any given formula φ. In Section 5, we shall see that these approximations are tight and often exact.

3.1 Closures and Extended Formulas

In this section, we assume that φ is a WMTL$_{\leq}$ formula over propositions P and (observable) clocks C and has been transformed into negative normal form (NNF), i.e. an equivalent formula in which negations are applied to the atomic propositions only. We use $\mathrm{Sub}(\varphi)$ to denote all the sub-formulas of φ.

In order to further expand φ into a disjunctive normal form, we introduce for each $\phi_1 \mathsf{U}^c_{\leq d} \phi_2 \in \mathrm{Sub}(\varphi)$ and each $\phi_1 \mathsf{R}^c_{\leq d} \phi_2 \in \mathrm{Sub}(\varphi)$, one local clock x and two clock bounds $x \leq d$ and $x > d$ to express some timing information related to $\phi_1 \mathsf{U}^c_{\leq d} \phi_2$ and $\phi_1 \mathsf{R}^c_{\leq d} \phi_2$. Also, we introduce auxiliary formulas $\phi_1 \mathsf{U}^c_{\leq d-x} \phi_2$ and $\phi_1 \mathsf{R}^c_{\leq d-x} \phi_2$ to express some requirements that should be satisfied in the future when we try to guarantee $\phi_1 \mathsf{U}^c_{\leq d} \phi_2 \in \mathrm{Sub}(\varphi)$ or $\phi_1 \mathsf{R}^c_{\leq d} \phi_2 \in \mathrm{Sub}(\varphi)$ is true in the current state.

We define $X_\varphi = \{x_{\phi_1 \mathsf{U}^c_{\leq d} \phi_2} | \phi_1 \mathsf{U}^c_{\leq d} \phi_2 \in \mathrm{Sub}(\varphi)\} \cup \{x_{\phi_1 \mathsf{R}^c_{\leq d} \phi_2} | \phi_1 \mathsf{R}^c_{\leq d} \phi_2 \in \mathrm{Sub}(\varphi)\}$ to be the set of all local clocks for φ, where $x_{\phi_1 \mathsf{U}^c_{\leq d} \phi_2}$ is the clock assigned to $\phi_1 \mathsf{U}^c_{\leq d} \phi_2$ and $x_{\phi_1 \mathsf{R}^c_{\leq d} \phi_2}$ is the local clock assigned to $\phi_1 \mathsf{R}^c_{\leq d} \phi_2$. We call $x_{\phi_1 \mathsf{U}^c_{\leq d} \phi_2}$ a local clock of $\mathsf{U}_{\leq}$-type, and $x_{\phi_1 \mathsf{R}^c_{\leq d} \phi_2}$ a local clock of $\mathsf{R}_{\leq}$-type. The mapping m from local clocks X_φ to observable clocks C is defined by $m(x_{\phi_1 \mathsf{U}^c_{\leq d} \phi_2}) = c$ and $m(x_{\phi_1 \mathsf{R}^c_{\leq d} \phi_2}) = c$. The closure of φ, write as $\mathrm{CL}(\varphi)$, is now defined by the following rules:

1. $\mathit{true} \in \mathrm{CL}(\varphi)$, $\mathrm{Sub}(\varphi) \subseteq \mathrm{CL}(\varphi)$
2. If $\phi_1 \mathsf{U}^c_{\leq d} \phi_2 \in \mathrm{Sub}(\varphi)$ and x is the local clock assigned to $\phi_1 \mathsf{U}^c_{\leq d} \phi_2$, then $x \leq d$, $x > d$, $\phi_1 \mathsf{U}^c_{\leq d-x} \phi_2 \in \mathrm{CL}(\varphi)$
3. If $\phi_1 \mathsf{R}^c_{\leq d} \phi_2 \in \mathrm{Sub}(\varphi)$ and x is the local clock assigned to $\phi_1 \mathsf{R}^c_{\leq d} \phi_2$, then $x \leq d$, $x > d$, $\phi_1 \mathsf{R}^c_{\leq d-x} \phi_2 \in \mathrm{CL}(\varphi)$
4. If $\Phi_1, \Phi_2 \in \mathrm{CL}(\varphi)$, then $\Phi_1 \wedge \Phi_2$, $\Phi_1 \vee \Phi_2 \in \mathrm{CL}(\varphi)$

Obviously, $\mathrm{CL}(\varphi)$ has only finitely many different non-equivalent formulas.

For a local clock x, we use $\mathit{rst}(x)$ to represent that x will be reset at current step and $\mathit{unch}(x)$ to represent that x will not be reset at current step. The set of extended formulas for φ, write as $\mathrm{Ext}(\varphi)$, is now defined by the following rules:

[3] For instance, $\Diamond^\tau_{\leq 1}(p \wedge \Box^\tau_{\leq 1}(\neg r) \wedge \Diamond^\tau_{\leq 1}(q))$ is an example of a formula not equivalent to any deterministic MWTA.

1. If $\Phi \in \mathrm{CL}(\varphi)$, then Φ, $\mathrm{O}\Phi \in \mathrm{Ext}(\varphi)$
2. If $x \in X_\varphi$ is a local clock of $\mathsf{U}_{\leq}$-type, then $\mathrm{unch}(x) \in \mathrm{Ext}(\varphi)$
3. If $x \in X_\varphi$ is a local clock of $\mathsf{R}_{\leq}$-type, then $\mathrm{rst}(x) \in \mathrm{Ext}(\varphi)$
4. If $\Phi_1, \Phi_2 \in \mathrm{Ext}(\varphi)$, then $\Phi_1 \wedge \Phi_2$, $\Phi_1 \vee \Phi_2 \in \mathrm{Ext}(\varphi)$

Extended formulas can be interpreted using extended weighted words. An *extended weighted word* $\omega = (a_0, v_0, \nu_0)(a_1, v_1, \nu_1)(a_2, v_2, \nu_2)\ldots$ is a sequence where $w = (a_0, v_0)(a_1, v_1)(a_2, v_2)\ldots$ is a weighted word over 2^P and C, and for every $i \in \mathbb{N}$, ν_i is a clock valuation over X_φ such that for all $x \in X_\varphi$, either $\nu_{i+1}(x) = v_{i+1}(m(x)) - v_i(m(x))$ or $\nu_{i+1}(x) = \nu_i(x) + v_{i+1}(m(x)) - v_i(m(x))$.

The semantics for extended formulas is naturally induced by the semantics of $\mathrm{WMTL}_{\leq}$ formulas:

Definition 4. *Let $\omega = (a_0, v_0, \nu_0)(a_1, v_1, \nu_1)(a_2, v_2, \nu_2)\ldots$ be an extended weighted word and $\Phi \in Ext(\varphi)$. The satisfaction relation $\omega^i \models_e \Phi$ is inductively defined as follows:*

1. $\omega^i \models_e x \sim d$ *iff* $\nu_i(x) \sim d$
2. $\omega^i \models_e rst(x)$ *iff* $\nu_{i+1}(x) = v_{i+1}(m(x)) - v_i(m(x))$
3. $\omega^i \models_e unch(x)$ *iff* $\nu_{i+1}(x) = \nu_i(x) + v_{i+1}(m(x)) - v_i(m(x))$
4. $\omega^i \models_e \phi$ *iff* $w^i \models \phi$, *if* $\phi \in Sub(\varphi)$
5. $\omega^i \models_e \varphi_1 \mathsf{U}^c_{\leq d-x} \varphi_2$ *iff there exists* j *such that* $j \geq i$, $w^j \models \varphi_2$, $v_j(c) - v_i(c) \leq d - \nu_i(x)$, *and* $w^k \models \varphi_1$ *for all* k *with* $i \leq k < j$
6. $\omega^i \models_e \varphi_1 \mathsf{R}^c_{\leq d-x} \varphi_2$ *iff for all* $j \geq i$ *such that* $v_j(c) - v_i(c) \leq d - \nu_i(x)$, *either* $w^j \models \varphi_2$ *or there exists* k *with* $i \leq k < j$ *and* $w^k \models \varphi_1$
7. $\omega^i \models_e \Phi_1 \wedge \Phi_2$ *iff* $\omega^i \models_e \Phi_1$ *and* $\omega^i \models_e \Phi_2$
8. $\omega^i \models_e \Phi_1 \vee \Phi_2$ *iff* $\omega^i \models_e \Phi_1$ *or* $\omega^i \models_e \Phi_2$
9. $\omega^i \models_e O\Phi$ *iff* $\omega^{i+1} \models_e \Phi$

ω^i *is a* model *of* Φ *if* $\omega^i \models_e \Phi$ *and two extended WMTL$_{\leq}$-formulas are said* equivalent *if they have exactly the same models.*

3.2 Constructing Non-deterministic Monitors

As in the construction of Büchi automata from LTL formulas, we will break a formula into a disjunction of several conjunctions [9]. Each of the disjuncts corresponds to a transition of a resulting observer automaton and specifies the requirements to be satisfied in the current and in the next states. In the rest of this section, we use $rst(\{x_1, x_2, \ldots, x_n\})$ and $unch(\{y_1, y_2, \ldots, y_n\})$ to denote the formula of $rst(x_1) \wedge rst(x_2) \wedge \ldots \wedge rst(x_n)$ and the formula of $unch(y_1) \wedge unch(y_2) \wedge \ldots \wedge unch(y_n)$ respectively. A *basic conjunction* is an extended formula of the form:

$$\alpha \wedge g \wedge rst(X) \wedge unch(Y) \wedge O(\Psi),$$

where α is a conjunction of literals (a literal is a proposition or its negation), g is a conjunction of clock bounds, X is a set of local clocks of $\mathsf{R}_{\leq}$-type, Y is a set of local clocks of $\mathsf{U}_{\leq}$-type, and Ψ is a formula in $\mathrm{CL}(\varphi)$. $\alpha \wedge g \wedge rst(X) \wedge unch(Y)$

specifies the requirements to be satisfied in the current state and Ψ specifies the requirements in the next state. The next Lemma 1 and main Theorem 1 provides the construction of a monitor from a formula.

Lemma 1. *Each formula in CL(φ) can be transformed into a disjunction of several basic conjunctions by using the following rules and Boolean equivalences.*

1. $f\,\mathsf{U}^c_{\leq d}\,g = g \vee (f \wedge O((x \leq d) \wedge (f\,\mathsf{U}^c_{\leq d-x} g)))$, *where x is the clock assigned to* $f\,\mathsf{U}^c_{\leq d}\,g$
2. $f\,\mathsf{U}^c_{\leq d-x}\,g = g \vee (f \wedge unch(x) \wedge O((x \leq d) \wedge (f\,\mathsf{U}^c_{\leq d-x} g)))$
3. $f\,\mathsf{R}^c_{\leq d}\,g = g \wedge (f \vee (rst(x) \wedge O(((x \leq d) \wedge (f\,\mathsf{R}^c_{\leq d-x} g)) \vee (x > d))))$, *where x is the clock assigned to* $f\,\mathsf{R}^c_{\leq d}\,g$
4. $f\,\mathsf{R}^c_{\leq d-x}\,g = g \wedge (f \vee O((x \leq d) \wedge (f\,\mathsf{R}^c_{\leq d-x} g)) \vee (x > d)))$
5. $(Of) \wedge (Og) = O(f \wedge g)$
6. $(Of) \vee (Og) = O(f \vee g)$

Theorem 1. *Let φ be a WMTL$_{\leq}$-formula over the propositions P and the clocks C and is in NNF. Let the MWTA $A_\varphi = (L, \ell_0, \ell_a, C_M, E, m)$ over the clocks C and the actions $\mathcal{A} = 2^P$ be defined as follows:*

- $L = \{\{\phi\} \mid \phi \in CL(\varphi)\}$ *is a finite set of locations, and* $\ell_0 = \{\varphi\}$ *is the initial location;*
- $\ell_a = \{true\}$ *is the accepting location;*
- $C_M = X_\varphi$ *is the set of all local clocks for φ;*
- $(\{f_1\}, a, g, \lambda, \{f_2\}) \in E$ *iff* $\alpha \wedge g \wedge rst(X) \wedge unch(Y) \wedge O(f_2)$ *is a basic conjunction of f_1 and that a satisfies α, and for each $x \in X_\varphi$ of* $\mathsf{U}_{\leq}$*-type, $x \in \lambda$ iff $x \notin Y$, and for each $x \in X_\varphi$ of* $\mathsf{R}_{\leq}$*-type, $x \in \lambda$ iff $x \in X$;*
- m *is defined by* $m(x_{\phi_1 \mathsf{U}^c_{\leq d} \phi_2}) = c$ *and* $m(x_{\phi_1 \mathsf{R}^c_{\leq d} \phi_2}) = c$.

Then $\mathcal{L}(\varphi) = \mathcal{L}(A_\varphi)$.

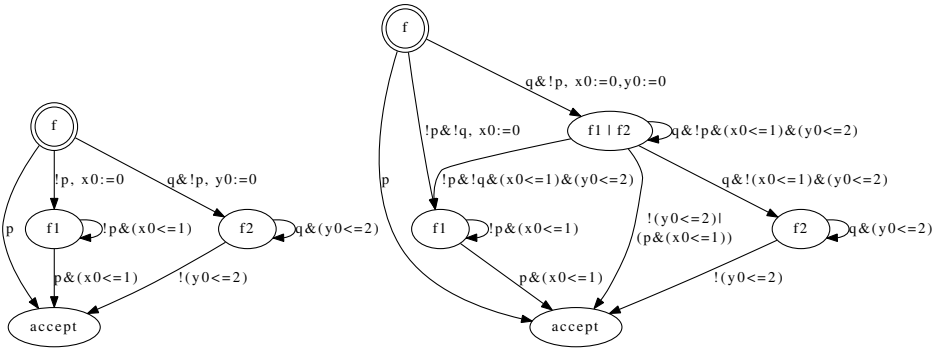
Example 1. Fig. 2a is a MWTA obtained with our approach for $f = (\Diamond^x_{\leq 1} p) \vee (\Box^y_{\leq 2} q) = (true\,\mathsf{U}^x_{\leq 1} p) \vee (false\,\mathsf{R}^y_{\leq 2} q)$.

3.3 Constructing Deterministic Monitors

The construction of section 3.2 might produce non-deterministic automata. In fact, as stated earlier, there exist WMTL$_{\leq}$ formulas for which no equivalent deterministic MWTA exists. To get deterministic MWTA for WMTL$_{\leq}$-formulas, we further translate formulas in disjunctive form into the following *deterministic* form by repeated use of the logical equivalence $p \Leftrightarrow (p \wedge q) \vee (p \wedge \neg q)$.

$$F = \bigvee_{i=1}^{n} \left(\alpha_i \wedge g_i \wedge \bigvee_{k=1}^{m_i} \left(rst(X_{ik}) \wedge unch(Y_{ik}) \wedge O(\Psi_{ik})\right)\right)$$

where for all $i \in \{1, \ldots, n\}$: m_i is a positive integer, $X_{ik} \subseteq X_\varphi$ is a set of local clocks of $\mathsf{R}_{\leq}$-type and $Y_{ik} \subseteq X_\varphi$ is a set of local clocks of $\mathsf{U}_{\leq}$-type, and for all $i \neq j$: $\alpha_i \wedge g_i \wedge \alpha_j \wedge g_j$ is *false*.



(a) Non-deterministic monitor (b) Deterministic under-approximation monitor

Fig. 2. Monitoring WTA for $f \equiv (\Diamond^{x}_{\leq 1} p) \vee (\Box^{y}_{\leq 2} q)$, with $f1 \equiv (x_0 \leq 1) \wedge (\mathit{true}\ \mathsf{U}^{x}_{\leq 1-x_0} p)$ and $f2 \equiv ((y_0 \leq 2) \wedge (\mathit{false}\ \mathsf{R}^{y}_{\leq 2-y_0} q)) \vee (y_0 > 2)$

Using the facts that O distributes over $\vee$, and $rst(X)$ and $unch(X)$ are monotonic in X, the following formulas are obviously strengthened (F^u) respectively weakened (F^o) versions of F:

$$F^u = \bigvee_{i=1}^{n} \left(\alpha_i \wedge g_i \wedge rst(\bigcup_{k=1}^{m_i} X_{ik}) \wedge unch(\bigcup_{k=1}^{m_i} Y_{ik}) \wedge O(\bigvee_{k=1}^{m_i} \Psi_{ik})\right)$$

$$F^o = \bigvee_{i=1}^{n} \left(\alpha_i \wedge g_i \wedge rst(\bigcap_{k=1}^{m_i} X_{ik}) \wedge unch(\bigcap_{k=1}^{m_i} Y_{ik}) \wedge O(\bigvee_{k=1}^{m_i} \Psi_{ik})\right)$$

Interestingly, by simply applying the construction of Theorem 1 to F^u (F^o) we immediately obtain a deterministic under-approximating (over-approximating) MWTA A^u_φ (A^o_φ) for φ. Moreover, if during the construction of A^u_φ we see that F^u is always semantically equivalent to F, then A^u_φ is an exact determinization of φ, i.e. $\mathcal{L}(A^u_\varphi) = \mathcal{L}(\varphi)$ (the same is true for overapproximation).

Example 2. (continued) Fig.2b is the under-approximation deterministic MWTA for $f = (\Diamond^{x}_{\leq 1} p) \vee (\Box^{y}_{\leq 2} q)$.

4 The Tool Chain

Figure 3 provides an architectural view of our tool chain. The tool chain takes as input a WMTL$_\leq$ formula φ, a WTA model M, as well as statistical parameters ϵ, α for controlling precision and confidence level. As a result a confidence interval for the probability $Pr[M \models \varphi]$ with the desired precision and confidence level is returned.

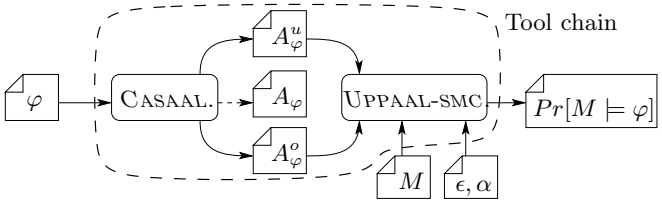


Fig. 3. Tool chain architecture

Casaal. The tool chain includes the new tool component Casaal. for generating monitors. The tool is implemented in C++ and is build on top of the Spot[4] open-source library for LTL to Büchi automata translation. We also use Buddy[5] BDD package to handle operations over Boolean formulas. Given a WMTL$_\leq$ formula φ, Casaal. may construct an exact monitoring WTA A_φ, as well as two – possibly approximating – monitoring WTAs, A^u_φ and A^o_φ. The tool also reports if one of these approximations is exact (i.e. recognizes exactly the language of φ). Table 1 demonstrates some experimental results for Casaal.. The formulas were also used in [13] and for comparison we list their results as well.

Table 1. Experimental results for WMTL$_\leq$ formulas

formula	automaton	states	trans	time(s)
$p\mathsf{U}^\tau_{\leq 1}(q\mathsf{U}^\tau_{\leq 1}(r\mathsf{U}^\tau_{\leq 1}s))$	nondet	5	14	0.02
	under	9	58	0.02
	over	9	56	0.04
	Geilen	14	30	
$(p \to \diamond^\tau_{\leq 5} q)\mathsf{U}^\tau_{\leq 100}\Box^\tau_{\leq 5}\neg p$	nondet	7	19	0.01
	under	9	32	0.01
	over	9	32	0.01
	Geilen	21	64	
$(((p\mathsf{U}^\tau_{\leq 4} q)\mathsf{U}^\tau_{\leq 3} r)\mathsf{U}^\tau_{\leq 2} s)\mathsf{U}^\tau_{\leq 1} t)$	nondet	17	121	0.02
	under	17	121	0.03
	over	17	121	0.03
	Geilen	60	271	

Uppaal-smc. [10,11] is a tool that allows to estimate and test $Pr[M \models \phi]$, i.e. the probability that a random run of a given WTA model M satisfies ϕ, where ϕ is a WMTL$_\leq$ formula restricted to the form $\diamond^c_{\leq d}\psi$ and ψ is a state predicate. Estimation is performed by generating a number of random simulations of M, where each simulation stops when either it reaches a state when ψ is satisfied, or $c \leq d$ is violated.

Combining Casaal. and Uppaal-smc. Let us describe how we use Uppaal-smc. together with the Casaal. tool to estimate the probability that a random run of a WTA model M satisfies a general WMTL$_\leq$ property ϕ, i.e. $Pr[M \models \phi]$.

Let us first assume, that one of two deterministic approximations for φ returned by Casaal. is exact. This means, that we have MWTA $A^{det}_\varphi = (L, \ell_0, \ell_a,$

[4] http://spot.lip6.fr/wiki/
[5] http://sourceforge.net/projects/buddy/develop

$C_M, E, m)$ such that $\mathcal{L}(A^{det}_{\varphi}) = \mathcal{L}(\varphi)$. First, we turn A^{det}_{φ} into input-enabled automaton by introducing a *rejecting* location l_r and adding complementary transitions to l_r from all other locations. Then we augment MWTA A^{det}_{φ} with a clock $c^{\dagger}$ that will grow with rate 1 in rejecting location ℓ_r, and with rate 0 in all other locations. Additionally, for every clock $c \in C_M$ we duplicate all rates and transition weights from the corresponding clock $m(c)$ to make sure, that the clocks of A^{det}_{φ} grow with the same rate as the corresponding clocks of the automaton M being monitored. Forming a parallel composition of M and A^{det}_{φ}, we may now use UPPAAL-SMC. to estimate the probability $p = Pr[M||A^{det}_{\varphi} \models \Diamond^{c^{\dagger}}_{\leq 1}(\ell_a)]$. This can be done because of the following theorem:

Theorem 2. *If M produces cost-divergent runs only, then each simulation of $M||A^{det}_{\varphi}$ will end up in accepting or rejecting location of A^{det}_{φ} after finite number of steps.*

If none of the two MWTAs A^{o}_{φ} and A^{u}_{φ} are exact determinization of A_{φ} (i.e. $\mathcal{L}(A^{u}_{\varphi}) \subsetneq \mathcal{L}(\varphi) \subsetneq \mathcal{L}(A^{o}_{\varphi})$), then we use both of them to compute upper (using A^{o}_{φ}) and lower (using A^{u}_{φ}) bounds for $Pr[M \models \varphi]$. Indeed, if n_1 (n_2, correspondingly) out of m random simulations of $M||A^{u}_{\varphi}$ ($M||A^{o}_{\varphi}$, correspondingly) ended in accepting location l^{u}_{a} (l^{o}_{a}, correspondingly), then with significance level of α we can accept a hypothesis H_1 (H_2, correspondingly) that $Pr[M \models \varphi] \geq n_1/m - \varepsilon$ ($Pr[M \models \varphi] \leq n_2/m + \varepsilon$). By combining hypothesis H_1 and H_2 we can obtain a confidence interval $[n_1/m - \varepsilon, n_2/m + \varepsilon]$ for $Pr[M \models \varphi]$ with significance level of $1 - (1 - \alpha)^2 = 2\alpha - \alpha^2$.

5 Case Studies

We performed several case studies to demonstrate the applicability of our tool chain. In the first case study we analyze the performance of CASAAL. on a set of randomly generated WMTL$_{\leq}$ formulas. In the second case study we use a model of a robot moving on a two-dimensional grid, this model was first analyzed in [5] using the manually constructed monitoring timed automaton.

5.1 Automatically Generated Formulas

In the first case study we analyze the performance of CASAAL. on a set of randomly generated WMTL$_{\leq}$ formulas. We generated 1000 formulas with 2, 3 and 4 actions, and created deterministic over and approximations for these formulas. Each of the formulas have 15 connectives (release, until, conjunction or disjunction) and four clocks.

For the formulas where only one or none of the approximations was exact (i.e. $\mathcal{L}(A^{u}_{\varphi}) \neq \mathcal{L}(A_{\varphi})$ or $\mathcal{L}(A^{o}_{\varphi}) \neq \mathcal{L}(A_{\varphi})$), we measured the "stochastic difference" between approximations by generating a number of random weighted words and estimating the probability that the over approximation accepts a random word, when the under approximation does not.

Table 2 reports the amount of formulas for which the under or over approximation was exact and the amount of formulas where none of them was exact. It

Table 2. Results for the random generated formula test

	# exact				Avg. time (s)		Avg. size		Stochastic difference	
Actions	under	over	none	one	under	over	under	over	no exact	one exact
2	831	542	169	289	0.24	1.01	6.35	6.35	0.27	0.15
3	706	370	294	336	1.42	2.75	12.29	12.29	0.05	0.03
4	586	233	414	353	8.66	13.05	22.97	22.97	0.01	0.02

also contains the average time spent for generating the monitors and the average number of locations, and the stochastic difference.

5.2 Robot Control

We consider the case of a robot moving on a two-dimensional grid that was explored in e.g. [5]. Each field of the grid is either `normal`, on `fire`, cold as `ice` or it is a wall which that cannot be passed. Also, there is a `goal` field that the robot must reach. The robot is moving in a random fashion i.e. it stays in a field for some time, and then randomly moves to one of the neighboring fields (if it is not a wall). Fig. 5 shows a robot controller implementing this along with the grid we use.

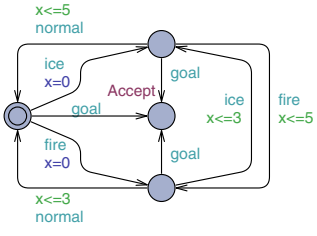


Fig. 4. Observer automaton used in [5]

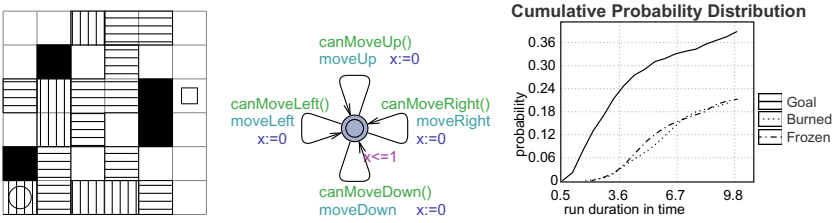
We are interested in the probability that the robot reaches its goal location without staying on consecutive fire fields for more than one time units and on consecutive ice fields for more than two time units.

In [5] the authors solved this problem by manually constructing a monitoring automaton to operate in parallel with the model of the robot. The automaton they used is depicted in Figure 4. Using WMTL$_{\leq}$ we can express the same requirement more easily as $\varphi \equiv (\varphi_1 \wedge \varphi_2)\mathsf{U}^{\tau}_{\leq 10}\texttt{goal}$, where:

$$\varphi_1 \equiv \texttt{ice} \implies \Diamond^{\tau}_{\leq 2}(\texttt{fire} \vee \texttt{normal} \vee \texttt{goal})$$
$$\varphi_2 \equiv \texttt{fire} \implies \Diamond^{\tau}_{\leq 1}(\texttt{ice} \vee \texttt{normal} \vee \texttt{goal})$$

Casaal. produces an MWTA (6 locations, 55 edges) that is an exact under-approximation for φ. Based on this MWTA, our tool chain estimates the probability that the random behavior of the robot satisfies φ to lie in the interval $[0.373, 0.383]$ with a confidence of 95%. Fig. 5c shows how we can visualize and compare the different distributions using the plot composer of Uppaal-smc..

Energy. We extend the model by limiting the energy of the robot that will stop moving when it runs out of energy. Furthermore, it can regain energy while staying on fire fields and use additional energy while staying on ice fields. Let c be the clock accumulating the amount of consumed energy. Now, we can express the property $\varphi \equiv (\varphi_1 \wedge \varphi_2 \wedge \neg\texttt{noEnergy})\mathsf{U}^{c}_{\leq 10}\texttt{goal}$ that the robot should not use



(a) (b) (c)

Fig. 5. (a) A 6×6 grid. The black fields are walls, the fields with vertical lines are on fire and the fields with horizontal lines contain ice. The circle indicates the robot's starting position and the square the goal. (b) WTA implementing the random movement of the robot. (c) Cumulative distribution of the robot reaching the goal, staying too long in the fire or too long on the ice.

more than 5 units of energy while obeying the requirements from before. The tool chain estimates the probability that the robot satisfies this requirement to lie in $[0.142; 0.152]$ with a confidence of 95%.

6 Related and Future Work

To our knowledge, we are the first to propose and implement an algorithm for translation of $\text{WMTL}_{\leq}$ formulas into monitoring automata. However, if we level down to $\text{MITL}_{\leq}$, there are several translation procedures described in the literature that are dealing with this logic. First, Rajeev Alur in [3] presents a procedure that is mostly theoretical and is not intended to be practically implemented. Second, Oded Maler et al. [16] proposed a procedure to translate MITL into temporal testers (not the classic timed automata), their procedure also has not been implemented. Nir Piterman et al. [17] proposed an approach how to translate MTL to deterministic timed automata under *finite variability* assumption (this assumption is not valid for the WTA stochastic semantics that we use). Finally, Marc Geilen[12] has implemented a procedure to translate $\text{MITL}_{\leq}$ to timed automata, but his approach works in the continuous semantics.

For future work we aim at extending our monitor- and approximate determinization constructions to $\text{WMTL}_{[a,b]}$ with (non-singleton) cost interval-bounds on the U modality in order to allow for SMC for this more expressive logic. Here a challenge will be how to bound the length of the random runs to be generated.

References

1. Agha, G., Meseguer, J., Sen, K.: Pmaude: Rewrite-based specification language for probabilistic object systems. Electronic Notes in Theoretical Computer Science 153(2), 213–239 (2006)
2. Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T.A., Ho, P., Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: The algorithmic analysis of hybrid systems. Theoretical Computer Science 138(1), 3–34 (1995)

3. Alur, R., Feder, T., Henzinger, T.A.: The benefits of relaxing punctuality. J. ACM 43, 116–146 (1996)
4. Alur, R., La Torre, S., Pappas, G.J.: Optimal Paths in Weighted Timed Automata. In: Di Benedetto, M.D., Sangiovanni-Vincentelli, A.L. (eds.) HSCC 2001. LNCS, vol. 2034, pp. 49–62. Springer, Heidelberg (2001)
5. Barbot, B., Chen, T., Han, T., Katoen, J.-P., Mereacre, A.: Efficient CTMC Model Checking of Linear Real-Time Objectives. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 128–142. Springer, Heidelberg (2011)
6. Behrmann, G., Fehnker, A., Hune, T., Larsen, K.G., Pettersson, P., Romijn, J., Vaandrager, F.W.: Minimum-Cost Reachability for Priced Timed Automata. In: Di Benedetto, M.D., Sangiovanni-Vincentelli, A.L. (eds.) HSCC 2001. LNCS, vol. 2034, pp. 147–161. Springer, Heidelberg (2001)
7. Bouyer, P., Larsen, K.G., Markey, N.: Model checking one-clock priced timed automata. Logical Methods in Computer Science 4(2) (2008)
8. Cassez, F., Larsen, K.G.: The Impressive Power of Stopwatches. In: Palamidessi, C. (ed.) CONCUR 2000. LNCS, vol. 1877, pp. 138–152. Springer, Heidelberg (2000)
9. Couvreur, J.-M.: On-the-Fly Verification of Linear Temporal Logic. In: Wing, J.M., Woodcock, J. (eds.) FM 1999. LNCS, vol. 1708, pp. 253–271. Springer, Heidelberg (1999)
10. David, A., Larsen, K.G., Legay, A., Mikučionis, M., Poulsen, D.B., van Vliet, J., Wang, Z.: Statistical Model Checking for Networks of Priced Timed Automata. In: Fahrenberg, U., Tripakis, S. (eds.) FORMATS 2011. LNCS, vol. 6919, pp. 80–96. Springer, Heidelberg (2011)
11. David, A., Larsen, K.G., Legay, A., Mikučionis, M., Wang, Z.: Time for Statistical Model Checking of Real-Time Systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 349–355. Springer, Heidelberg (2011)
12. Geilen, M.: An Improved On-the-Fly Tableau Construction for a Real-Time Temporal Logic. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 394–406. Springer, Heidelberg (2003)
13. Geilen, M., Dams, D.R.: An On-the-fly Tableau Construction for a Real-Time Temporal Logic. In: Joseph, M. (ed.) FTRTFT 2000. LNCS, vol. 1926, pp. 276–290. Springer, Heidelberg (2000)
14. Clarke Jr., E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press (1999)
15. Koymans, R.: Specifying real-time properties with metric temporal logic. Real-Time Syst. 2, 255–299 (1990)
16. Maler, O., Ničković, D., Pnueli, A.: From MITL to Timed Automata. In: Asarin, E., Bouyer, P. (eds.) FORMATS 2006. LNCS, vol. 4202, pp. 274–289. Springer, Heidelberg (2006)
17. Ničković, D., Piterman, N.: From MTL to Deterministic Timed Automata. In: Chatterjee, K., Henzinger, T.A. (eds.) FORMATS 2010. LNCS, vol. 6246, pp. 152–167. Springer, Heidelberg (2010)
18. Sen, K., Viswanathan, M., Agha, G.: On Statistical Model Checking of Stochastic Systems. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 266–280. Springer, Heidelberg (2005)
19. Younes, H.L.S.: Verification and Planning for Stochastic Processes with Asynchronous Events. PhD thesis, Carnegie Mellon University (2005)
20. Zuliani, P., Platzer, A., Clarke, E.M.: Bayesian statistical model checking with application to simulink/stateflow verification. In: HSCC 2010, pp. 243–252. ACM, New York (2010)

Duality between Merging Operators and Social Contraction Operators

José Luis Chacón and Ramón Pino Pérez

Departamento de Matemáticas
Facultad de Ciencias
Universidad de Los Andes
Mérida, Venezuela
{jlchacon,pino}@ula.ve

Abstract. In the AGM (Alchourrón-Gärdenfors-Makinson) framework there exists a duality between revision operators and contraction operators. This duality is given by the Levi identity and the Harper identity. The former allows to define a revision operator starting from a contraction operator. The latter allows to define a contraction operator starting from a revision operator. In this work we show that this duality can be extended to a duality between merging operators and social contraction operators through some identities in the style of the Levi and Harper identities.

1 Introduction

In belief change there are many operators aiming to model different situations in which the beliefs of one (or some) agent(s) evolve over time. Among these operators one can quote revision [1, 13, 14, 18], contraction [2–4], update [16, 17], abduction [27], extrapolation [11, 12], etc. The most studied are contraction and revision. In logic based representation of beliefs, a contraction occurs when we remove a sentence α from a closed theory K in order to obtain a closed theory K' in which α does not appear. A revision occurs when we wish to incorporate a sentence α in a closed theory K in order to obtain a new consistent closed theory K' containing α. The most natural procedure to perform contraction might be to suppress the piece of information and then take the logical closure. In symbols it is $Cn(K \setminus \{\alpha\})$. Unfortunately, this doesn't always work because many times $\alpha \in Cn(K \setminus \{\alpha\})$. Analogously, the most natural procedure to perform revision might be to add the piece of information and then take the logical closure (in symbols $Cn(K \cup \{\alpha\})$). Unfortunately, this doesn't work because many times $Cn(K \cup \{\alpha\})$ is inconsistent. This explains roughly why modeling contraction and revision is not a trivial task. The situation is very different with the expansion which can be modeled by a simple procedure: add the piece of information and then take the logical closure. Usually the symbol $+$ is used to denote expansion (which is actually unique), so expansion is defined by the following equation: $K + \alpha = Cn(K \cup \{\alpha\})$. Of course, expansion can lead to inconsistencies.

N. Bjørner and A. Voronkov (Eds.): LPAR-18, LNCS 7180, pp. 183–196, 2012.

At the end of 70's Levi [22] proposed a procedure to perform revision based on contraction: first contract K by the negation of α, then add α and finally take logical closure. In symbols, that means that we can define a revision operator $*$ in terms of a contraction operator $\dot{-}$ by the following identity:

$$K * \alpha = (K \dot{-} \neg\alpha) + \alpha \tag{1}$$

This identity is known as Levi's identity.

Almost simultaneously, Harper [15] proposed a procedure to perform contraction based on revision: first revise K by the negation of α and then intersect this theory with K. In symbols, we can define a contraction operator $\dot{-}$ in terms of a revision operator $*$ by the following identity:

$$K \dot{-} \alpha = (K * \neg\alpha) \cap K \tag{2}$$

This identity is known as Harper's identity.

Some years later contraction operators and revision operators were characterized [1, 13]. Moreover, it was proved that the Harper identity and the Levi identity define a very strong duality between contraction and revision operators in the sense that operators defined by Equation (1) are true revision operators and operators defined by Equation (2) are true contraction operators.

Other change operators aiming to model the process of merging several sources of information in a unique consistent piece of information have been introduced [5, 6, 20, 21, 23–26, 28, 29]. Understanding this kind of operators is very useful in multi-agents systems, distributed information, decision theory etc. Some representation theorems have been established and some families of this operators have been effectively built [20]. It has been proved that these merging operators, more precisely merging operators with integrity constraints, are extensions of revision operators. On the other hand, operators extending contraction operators to many sources of information have been defined by Booth [7, 8]. These operators are called social contraction operators. They aim to model negotiation processes in which agents have to produce a consensual belief base in which a given information is not present and this consensual belief is close to the original belief base of each agent.

Actually, Booth conjectured that there is a duality between social contraction operators and merging operators with integrity constraints. The goal of this work is to establish the strong duality between these classes of operators.

The rest of this work is organized as follows: Section 2 contains the basic definitions. Section 3 is devoted to the definition and characterization of merging operators with integrity constraints. Section 4 is devoted to the definition and characterization of social contraction operators. In Section 5, we establish the strong duality between the social contraction operators and merging operators with integrity constraints. We finish this work with a section containing some concluding remarks.

2 Preliminaries

We consider a propositional language $\mathcal{L}$ defined from a finite set of propositional variables $\mathcal{P}$ and the standard connectives, including $\top$ and $\bot$.

An interpretation ω is a total function from $\mathcal{P}$ to $\{0,1\}$. The set of all interpretations is denoted by $\mathcal{W}$. An interpretation ω is a model of a formula $\phi \in \mathcal{L}$ if and only if it makes it true in the usual truth functional way. $[\![\varphi]\!]$ denotes the set of models of the formula φ, i.e., $[\![\varphi]\!] = \{\omega \in \mathcal{W} \mid \omega \models \varphi\}$.

A *base* K is a finite set of propositional formulae. Sometimes, in order to simplify notation, we will identify the base K with the formula φ which is the conjunction of the formulae of K[1]. We denote by $\mathcal{K}$ the set of bases.

A *profile* Φ is a non-empty finite multi-set (bag) of bases $\Phi = \{\varphi_1, \ldots, \varphi_n\}$ (hence different agents are allowed to exhibit identical bases), and represents a group of n agents. In that case n is the size of the profile. Profiles sizes can vary. We denote by $\mathcal{E}$ the set of profiles.

We denote by $\bigwedge \Phi$ the conjunction of bases of $\Phi = \{\varphi_1, \ldots, \varphi_n\}$, i.e., $\bigwedge \Phi = \varphi_1 \wedge \ldots \wedge \varphi_n$. A profile Φ is said to be consistent if and only if $\bigwedge \Phi$ is consistent. The multi-set union is noted $\sqcup$. By abuse of notation we will write $\varphi \sqcup \Phi$ instead of $\{\varphi\} \sqcup \Phi$. We denote by Φ^n the profile in which Φ appears n times, more precisely $\Phi^n = \underbrace{\Phi \sqcup \ldots \sqcup \Phi}_{n}$. Two profiles are equivalent, denoted $\Phi_1 \equiv \Phi_2$, iff there is a bijective function f from Φ_1 onto Φ_2 such that for any $\varphi \in \Phi_1$, $f(\varphi) \equiv \varphi$.

A base (formula) φ is complete if it has only one model. A profile Φ is complete if all the bases of Φ are complete formulae.

If $\leq$ denotes a pre-order on $\mathcal{W}$ (i.e., a reflexive and transitive relation), then $<$ denotes the associated strict order defined by $\forall \omega, \omega' \in \mathcal{W}$, $\omega < \omega'$ if and only if $\omega \leq \omega'$ and $\omega' \not\leq \omega$. A pre-order is *total* if $\forall \omega, \omega' \in \mathcal{W}$, $\omega \leq \omega'$ or $\omega' \leq \omega$. A pre-order that is not total is called *partial*. Let $\leq$ be a pre-order on A, and $B \subseteq A$, then $\min(B, \leq) = \{b \in B \mid \nexists a \in B \ a < b\}$.

If A is a set, we denote $|A|$ the cardinality of A. The symbol $\subseteq$ will denote set containment and $\subset$ strict set containment, i.e., $A \subset B$ if and only if $A \subseteq B$ and $A \neq B$.

Let $\Delta : \mathcal{E} \times \mathcal{L} \mapsto \mathcal{K}$ be a function. We will use the notation $\Delta_\mu(\Phi)$ instead of $\Delta(\Phi, \mu)$.

There are some set theoretical notions (semantical) which are dual, in the finite case, to the previous ones. The set of non empty subsets of $\mathcal{W}$ will be denoted $\mathcal{B}$. If S is in $\mathcal{B}$, the set $\mathcal{W} \setminus S$ is denoted by $\overline{S}$. We suppose we have a finite set of sources of information $\mathsf{Sources} = \{1, \ldots, n\}$ with $n \geq 1$. The number n of sources of information can vary. The information of a source i will be a set $U_i \in \mathcal{B}$. An s-profile (of information relative to $\mathsf{Sources}$) is an element $\overrightarrow{U} \in \mathcal{B}^n$. We use $\overrightarrow{U}, \overrightarrow{U}^1, \ldots$ to denote s-profiles. The idea is that in an

[1] This identification will be done when the approach is not sensitive to syntactical representation. When the approach is sensitive to syntactical representation, it will be important to distinguish between K and the conjunction of its formulae (see e.g. [19]). The operators in this work are all syntax independent.

s-profile $\overrightarrow{U} = (U_1, \ldots, U_n)$, U_i is the information of the source i. An s-profile is consistent if $\bigcap \overrightarrow{U} = \bigcap_i U_i \neq \emptyset$, otherwise is inconsistent. Given two s-profiles $\overrightarrow{U}^1$ and $\overrightarrow{U}^2$, we write $\overrightarrow{U}^1 \subseteq \overrightarrow{U}^2$ when $U_i^1 \subseteq U_i^2$ for each source $i \in \mathsf{Sources}$. If f is a function with codomain $\mathcal{B}^n$, we write $f^i(x)$ to denote the $i-$th element of $f(x)$. If $\overrightarrow{U} \in \mathcal{B}^n$ and σ is a permutation[2] we define $\sigma(\overrightarrow{U}) = (U_{\sigma(1)}, \ldots, U_{\sigma(n)})$. If $\overrightarrow{U} \in \mathcal{B}^n$ and $\overrightarrow{V} \in \mathcal{B}^m$, the vector $(U_1, \ldots, U_n, V_1, \ldots, V_m) \in \mathcal{B}^{n+m}$ will be denoted $\overrightarrow{U} \sqcup \overrightarrow{V}$. If $\overrightarrow{U}, \overrightarrow{V} \in \mathcal{B}^n$ we define $\overrightarrow{U} \cup \overrightarrow{V} = (U_1 \cup V_1, \ldots, U_n \cup V_n)$ and $\overrightarrow{U} \cap \overrightarrow{V} = (U_1 \cap V_1, \ldots, U_n \cap V_n)$. If $S \in \mathcal{B}$ and $\overrightarrow{U} \in \mathcal{B}^n$ we write $\overrightarrow{U} \cup S$ to denote the vector $(U_1 \cup S, \ldots, U_n \cup S)$ and $\overrightarrow{U} \cap S$ to denote the vector $(U_1 \cap S, \ldots, U_n \cap S)$. We put $|\overrightarrow{V}| = n$ when $\overrightarrow{V} \in \mathcal{B}^n$. Finally we define $\mathcal{H} = \bigcup_{n \geq 1} \mathcal{B}^n$.

3 Merging Operators

Definition 1. *$\Delta : \mathcal{E} \times \mathcal{L} \mapsto \mathcal{K}$ is said to be an integrity constraints merging operator (IC-merging operator for short) iff the following properties hold:*

(IC0) $\Delta_\mu(\Phi) \vdash \mu$
(IC1) *If μ is consistent, then $\Delta_\mu(\Phi)$ is consistent*
(IC2) *If $\bigwedge \Phi$ is consistent with μ, then $\Delta_\mu(\Phi) \leftrightarrow \bigwedge \Phi \wedge \mu$*
(IC3) *If $\Phi_1 \leftrightarrow \Phi_2$ and $\mu_1 \leftrightarrow \mu_2$, then $\Delta_{\mu_1}(\Phi_1) \leftrightarrow \Delta_{\mu_2}(\Phi_2)$*
(IC4) *If $\varphi_1 \vdash \mu$ and $\varphi_2 \vdash \mu$, then $\Delta_\mu(\{\varphi_1, \varphi_2\}) \wedge \varphi_1$ is consistent if and only if $\Delta_\mu(\{\varphi_1, \varphi_2\}) \wedge \varphi_2$ is consistent*
(IC5) $\Delta_\mu(\Phi_1) \wedge \Delta_\mu(\Phi_2) \vdash \Delta_\mu(\Phi_1 \sqcup \Phi_2)$
(IC6) *If $\Delta_\mu(\Phi_1) \wedge \Delta_\mu(\Phi_2)$ is consistent, then $\Delta_\mu(\Phi_1 \sqcup \Phi_2) \vdash \Delta_\mu(\Phi_1) \wedge \Delta_\mu(\Phi_2)$*
(IC7) $\Delta_{\mu_1}(\Phi) \wedge \mu_2 \vdash \Delta_{\mu_1 \wedge \mu_2}(\Phi)$
(IC8) *If $\Delta_{\mu_1}(\Phi) \wedge \mu_2$ is consistent, then $\Delta_{\mu_1 \wedge \mu_2}(\Phi) \vdash \Delta_{\mu_1}(\Phi)$*

Some of these properties had been proposed by Revesz [29] in order to define his *model fitting* operators.

Intuitively $\Delta_\mu(\Phi)$ is a belief base *close* to profile Φ satisfying the integrity constraint μ. This idea is what the postulates try to capture. The meaning of the postulates is the following: (IC0) assures that the result of the merging satisfies the integrity constraints. (IC1) states that if the integrity constraints are consistent, then the result of the merging will be consistent. (IC2) states that if possible, the result of the merging is simply the conjunction of the belief bases with the integrity constraints. (IC3) is the principle of irrelevance of syntax, i.e. if two belief sets are equivalent and two integrity constraints bases are logically equivalent then the belief bases resulting of the two mergings will be logically equivalent. (IC4) is the fairness postulate, the point is that when we merge two belief bases, merging operators must not give preference to one of them. (IC5) expresses the following idea: if two groups Φ_1 and Φ_2 agree on some alternatives then these alternatives will be chosen if we join the two groups. (IC5) and (IC6) together state that if one could find two subgroups which agree on at least one

[2] The set of all permutations of n will be denoted S_n.

alternative, then the result of the global merging will be exactly those alternatives the two groups agree on. (IC7) and (IC8) are a direct generalization of the (R5-R6) postulates for revision (version Katsuno-Mendelzon or K*7-K*8 version AGM). They state some conditions about integrity constraints conjunctions. Actually, they ensure that the notion of *closeness* is well-behaved. For instance, if an alternative A is chosen among a set of alternatives, then if the set of alternatives is narrowed but the alternative A remains in this set, the alternative A will be still chosen. This quite natural property is found in different rational choice theories (Social Choice, Decision, etc).

In order to establish a useful representation theorem for this kind of operators we need the following definition:

Definition 2. *A* syncretic assignment *is a function mapping each profile Φ to a total pre-order $\leq_\Phi$ over interpretations such that for any profiles Φ, Φ_1, Φ_2 and for any belief bases φ, φ' the following conditions hold:*

1. *If $\omega \models \bigwedge \Phi$ and $\omega' \models \bigwedge \Phi$, then $\omega \simeq_\Phi \omega'$*
2. *If $\omega \models \bigwedge \Phi$ and $\omega' \not\models \bigwedge \Phi$, then $\omega <_\Phi \omega'$*
3. *If $\Phi_1 \leftrightarrow \Phi_2$, then $\leq_{\Phi_1} = \leq_{\Phi_2}$*
4. *$\forall \omega \models \varphi \ \exists \omega' \models \varphi' \ \omega' \leq_{\varphi \sqcup \varphi'} \omega$*
5. *If $\omega \leq_{\Phi_1} \omega'$ and $\omega \leq_{\Phi_2} \omega'$, then $\omega \leq_{\Phi_1 \sqcup \Phi_2} \omega'$*
6. *If $\omega <_{\Phi_1} \omega'$ and $\omega \leq_{\Phi_2} \omega'$, then $\omega <_{\Phi_1 \sqcup \Phi_2} \omega'$*

Theorem 1 ([20]). *An operator Δ is an IC merging operator if and only if there exists a syncretic assignment that maps each profile Φ to a total pre-order $\leq_\Phi$ such that*

$$[\![\Delta_\mu(\Phi)]\!] = \min([\![\mu]\!], \leq_\Phi)$$

When this equation holds we will say that the assignment represents the operator.

This theorem has been generalized to the framework of infinite Propositional Logic (see [10]).

4 Social Contraction Operators

In this section we will consider some operators aiming to model the process of removing one piece of information of a group of agents. Because the pieces of information are represented in (finite) propositional logic, we can opt for a syntactical or semantical presentation. Although they are equivalent, we have chosen giving a semantical presentation of operators and rationality postulates for at least three reasons: the meaning of postulates is easier to grasp, the set theoretical aspects of the presentation can be transposable to other domains as decision theory and game theory and this presentation seems to us easier to understand.

Formally, we consider functions of the following shape $f : \mathcal{B} \times \mathcal{H} \to \mathcal{H}$ with the property $|f(S, \overrightarrow{U})| = |\overrightarrow{U}|$. This kind of functions will be called social contraction functions, SC-functions for short. Given an s-profile $\overrightarrow{U}$ and a special

information S in $\mathcal{B}$ considered as the information to contract, $f(S, \overrightarrow{U})$ has to represent a modification of $\overrightarrow{U}$ such that the belief in each source and $\overline{S}$ are mutually consistent. Notice that the natural idea to modify each source is adding to it some models in $\overline{S}$. The postulates of social contraction operators impose control in performing this task.

When S is fixed we can associate to a SC-function f, a function $f_S : \mathcal{H} \to \mathcal{H}$, by letting $f_S(\overrightarrow{U}) = f(S, \overrightarrow{U})$. If $f_S(\overrightarrow{U}) = (T_1, \ldots, T_n)$ we define $f_S^i(\overrightarrow{U}) = T_i$ for $i = 1, \ldots, n$.

An interesting function associated to a SC-function is the following one: $\widetilde{f}_S(\overrightarrow{U}) := \bigcap f_S(\overrightarrow{U}) \cap \overline{S}$, where $\bigcap f_S(\overrightarrow{U}) = \bigcap_{i=1}^n f_S^i(\overrightarrow{U})$.

Next we define the rationality postulates characterizing the social contraction operators. After the definition we will comment about the meaning of these postulates.

Definition 3. *Let $f : \mathcal{B} \times \mathcal{H} \to \mathcal{H}$ be a SC-function. We say that f is a social contraction operator iff the following properties hold:*

(SC$_\mathrm{m}$) *If $\omega \in f_S^i(\overrightarrow{U}) \backslash U_i$, then $\omega \in \bigcap f_S(\overrightarrow{U})$.*
(SC0) *$\sigma(f_S(\overrightarrow{U})) = f_S(\sigma(\overrightarrow{U}))$, for $\overrightarrow{U} \in \mathcal{B}^n$ and σ a permutation.*
(SC1) *$\overrightarrow{U} \subseteq f_S(\overrightarrow{U}) \subseteq \overrightarrow{U} \cup \overline{S}$.*
(SC2) *$\overline{S} \neq \emptyset \Rightarrow \widetilde{f}_S(\overrightarrow{U}) \neq \emptyset$.*
(SC3) *$\bigcap \overrightarrow{U} \cap \overline{S} \neq \emptyset \Rightarrow f_S(\overrightarrow{U}) = \overrightarrow{U}$.*
(SC4) *If $U \subseteq \overline{S}$ and $V \subseteq \overline{S}$, then $f_S^1(U, V) \cap V \neq \emptyset \Rightarrow f_S^2(U, V) \cap U \neq \emptyset$.*
(SC5) *$\bigcap f_S(\overrightarrow{U}) \cap \bigcap f_S(\overrightarrow{V}) \cap \overline{S} \subseteq \bigcap f_S(\overrightarrow{U} \sqcup \overrightarrow{V}) \cap \overline{S}$.*
(SC6) *If $f_S(\overrightarrow{U}) \cap f_S(\overrightarrow{V}) \cap \overline{S} \neq \emptyset$ then $\widetilde{f}_S(\overrightarrow{U} \sqcup \overrightarrow{V}) \subseteq \widetilde{f}_S(\overrightarrow{U}) \cap \widetilde{f}_S(\overrightarrow{V})$.*
(SC7) *$\bigcap f_S(\overrightarrow{U}) \cap (\overline{S \cup T}) \subseteq \bigcap f_{S \cup T}(\overrightarrow{U}) \cap (\overline{S \cup T})$.*
(SC8) *If $\bigcap f_S(\overrightarrow{U}) \cap (\overline{S \cup T}) \neq \emptyset$, then*

$$\bigcap f_{S \cup T}(\overrightarrow{U}) \cap (\overline{S \cup T}) \subseteq \bigcap f_S(\overrightarrow{U}) \cap (\overline{S \cup T})$$

SC$_\mathrm{m}$ is a basic postulate guaranteeing that all sources weaken the information in the same way. It is a sort of monotony or regularity. If a model is added to the source i, it has to be added to all the sources. In some sense this postulate corresponds to a sort of fairness in the process: all agents have to be treated equally. SC0 is the postulate of anonymity: the order of the sources is irrelevant. SC1 ensures that the contraction process consists in adding models of $\overline{S}$. Actually, if the size of the s-profile $\overrightarrow{U}$ is 1, *i.e.* there is a unique source of information, this postulate coincides with the semantical version of Recovery postulate[3] for contraction operators (postulate $K \dot{-} 5$, see [1, 13, 14]). SC2 guarantees that the output of the contraction will be "consistent" whenever S is not a "tautology" (here "consistent" means nonempty set and "tautology" means the whole set $\mathcal{W}$). SC3 says that there is nothing to do when each source does not "entail" S (here the entailment is understood as the inclusion of sets). SC4 concerns fairness.

[3] The Recovery postulate for contraction operators reads:
if $\alpha \in K$ then $K \subseteq (K \dot{-} \alpha) + \alpha$.

When there are two sources to contract both entailing the negation of the information to contract, if a model of one source is added to the other source, then the inverse situation occurs. SC5 says that the common models added to two s-profiles have to be added to the union (concatenation) of these s-profiles. SC6, together with SC5, says that if there are really common models added to two profiles these models are exactly the models added to the union (concatenation) of these s-profiles. SC7 establishes that if the models added to perform the contraction by S are in $\overline{T}$, then they are added to perform the contraction by $S \cup T$. SC8 is just the other inclusion in the case where there are effectively models added to perform the contraction by S which are in $\overline{T}$.

5 Duality

Notice that if $S \subset \mathcal{W}$ there is a formula φ such that $[\![\varphi]\!] = S$. Actually, the set of such formulas is a class of equivalence under the relation $\varphi \equiv \psi$ iff $[\![\varphi]\!] = [\![\psi]\!]$. We denote by S_α a formula such that $[\![S_\alpha]\!] = S$. For any formula φ, the set $[\![\varphi]\!]$ will be denoted by φ_m. It is clear that $(S_\alpha)_m = S$ and $(\varphi_m)_\alpha \equiv \varphi$. The following relations are very easy to check: $S_\alpha \wedge T_\alpha \equiv (S \cap T)_\alpha$, $\psi_m \cap \phi_m = (\psi \wedge \phi)_m$, $S_\alpha \vee T_\alpha \equiv (S \cup T)_\alpha$ and $\psi_m \cup \phi_m = (\psi \vee \phi)_m$.

For an s-profile $\overrightarrow{U} \in \mathcal{B}^n$, the multiset $\{(U_i)_\alpha : U_i \in \overrightarrow{U}\}$ will be denoted $(\overrightarrow{U})_\alpha$. Conversely, for a profile $\Phi = \{\varphi_1, \ldots, \varphi_n\}$, we define $\overrightarrow{\Phi}_m$ the set of s-profiles $\overrightarrow{U}$ in $\mathcal{B}^n$ such that there exists a permutation σ verifying $\sigma(\overrightarrow{U}) = ((\varphi_1)_m, \ldots, (\varphi_n)_m)$.

Now we define the identities of Harper and Levi generalized:

$$f_S(\overrightarrow{U}) = \overrightarrow{U} \cup (\Delta_{\overline{S}_\alpha}(\overrightarrow{U})_\alpha)_m \tag{3}$$

$$\Delta_\mu(\Phi) = \left(\bigcap f_{(\neg\mu)_m}(\overrightarrow{\Phi}_m) \cap \mu_m\right)_\alpha \tag{4}$$

These identities will be the duality announced. The following theorem establishes the first duality result:

Theorem 2 (From merging to social contraction). *Let Δ be a merging operator with integrity constraints. Let f be defined by Equation 3 (the identity of Harper generalized). Then f is a social contraction operator.*

Proof. Suppose $f_S(\overrightarrow{U}) = \overrightarrow{U} \cup (\Delta_{\overline{S}_\alpha}(\overrightarrow{U})_\alpha)_m$ with Δ a merging operator with integrity constraints. We have to verify that all the postulates of Definition 3 hold.

(SC$_m$) This is clearly verified because we are adding the same models $((\Delta_{\overline{S}_\alpha}(\overrightarrow{U})_\alpha)_m)$ to each source.

(SC0) This condition is also clearly verified because all the sources are weakened by $(\Delta_{\overline{S}_\alpha}(\overrightarrow{U})_\alpha)_m$ without regard to her position in the s-profile $\overrightarrow{U}$.

(SC1) Notice that, by (IC0) we have $(\Delta_{\overline{S}_\alpha}(\overrightarrow{U})_\alpha)_m \subset ((\overline{S})_\alpha)_m = \overline{S}$. Then $U_i \subset U_i \cup (\Delta_{\overline{S}_\alpha}(\overrightarrow{U})_\alpha)_m \subset U_i \cup \overline{S}$.

(SC2) The postulate (IC1) says that the consistency of μ guarantees the consistency of $\Delta_\mu(\Phi)$. If $\overline{S} \neq \varnothing$, then $(\overline{S})_\alpha$ is consistent; thus, $\Delta_{\overline{S}_\alpha}(\overrightarrow{U})_\alpha$ is consistent. That is $(\Delta_{\overline{S}_\alpha}(\overrightarrow{U})_\alpha)_m \neq \varnothing$. Therefore the condition SC2 is verified because $\widetilde{f}_S(\overrightarrow{U}) = \bigcap f_S(\overrightarrow{U}) \cap \overline{S} \supset (\Delta_{\overline{S}_\alpha}(\overrightarrow{U})_\alpha)_m$.

(SC3) Notice that postulate (IC3) says that if Φ and μ are mutually consistent then $\Delta_\mu(\Phi) \equiv \bigwedge \Phi \wedge \mu$. Suppose that $\bigcap_i \overrightarrow{U} \cap \overline{S} \neq \varnothing$, that is, $\bigwedge(\overrightarrow{U}_\alpha) \wedge (\overline{S})_\alpha$ is consistent. From this, by (IC3), we have $(\Delta_{\overline{S}_\alpha}(\overrightarrow{U})_\alpha) = \bigwedge(\overrightarrow{U}_\alpha) \wedge (\overline{S})_\alpha$. Thus, $(\Delta_{\overline{S}_\alpha}(\overrightarrow{U})_\alpha)_m = \bigcap(\overrightarrow{U}_\alpha)_m \cap ((\overline{S})_\alpha)_m = \bigcap \overrightarrow{U} \cap \overline{S} \subset U_i$ for $i = 1, \ldots, n$, therefore $U_i \cup (\Delta_{\overline{S}_\alpha}(\overrightarrow{U})_\alpha)_m = U_i$, that is $f_S(\overrightarrow{U}) = \overrightarrow{U}$ and therefore condition SC3 holds.

(SC4) This condition follows from (IC4). Suppose that $U \subset \overline{S}$, $V \subset \overline{S}$ and $f_S^1(U,V) \cap V \neq \emptyset$. Thus, $(U)_\alpha \vdash (\overline{S})_\alpha$, $V_\alpha \vdash \overline{S}_\alpha$. By definition

$$f_S(U,V) = (U \cup (\Delta_{\overline{S}_\alpha}(U_\alpha \sqcup V_\alpha)_m, V \cup (\Delta_{\overline{S}_\alpha}(U_\alpha \sqcup V)_\alpha)_m)$$

By hypothesis, $(U \cup (\Delta_{\overline{S}_\alpha}(U_\alpha \sqcup V_\alpha)_m) \cap V \neq \emptyset$. If $U \cap V \neq \emptyset$, by (CS3), we have $f_S(U,V) = (U,V)$ and therefore $f_S^2(U,V) \cap U \neq \emptyset$. In the case $U \cap V = \emptyset$ we have $(\Delta_{\overline{S}_\alpha}(U_\alpha \sqcup V_\alpha))_m \cap V \neq \emptyset$. Thus, $\Delta_{\overline{S}_\alpha}(U_\alpha \sqcup V_\alpha) \wedge V_\alpha$ is consistent; this together with $U_\alpha \vdash S_\alpha$ $V_\alpha \vdash S_\alpha$ and Postulate (IC4) entail $\Delta_{\overline{S}_\alpha}(U_\alpha \sqcup V_\alpha) \wedge U_\alpha$ is consistent. Therefore $(\Delta_{\overline{S}_\alpha}(U_\alpha \sqcup V_\alpha))_m \cap U \neq \emptyset$; thus, by definition of $f_S(\overrightarrow{U})$ we obtain $f_S^2(U,V) \cap U \neq \emptyset$.

(SC5) Suppose $w \in \bigcap f_S(\overrightarrow{U}) \cap \bigcap f_S(\overrightarrow{V}) \cap \overline{S}$. Then $w \in ((\Delta_{(\overline{S})_\alpha}(\overrightarrow{U})_\alpha)_m \cup (\bigcap \overrightarrow{U})) \cap ((\Delta_{(\overline{S})_\alpha}(\overrightarrow{V})_\alpha)_m) \cup (\bigcap \overrightarrow{V})) \cap \overline{S}$. If $w \in \bigcap \overrightarrow{U}$, since $w \in \overline{S}$, we have, by (IC2), $(\Delta_{\overline{S}_\alpha}(\overrightarrow{U}))_m = \bigcap \overrightarrow{U} \cap \overline{S}$. From this, it follows $w \in (\Delta_{\overline{S}_\alpha}(\overrightarrow{U}))_m$; if $w \in \bigcap \overrightarrow{V}$, then $w \in (\Delta_{\overline{S}_\alpha}(\overrightarrow{V})_\alpha)_m$. In any case $w \in (\Delta_{\overline{S}_\alpha}(\overrightarrow{U})_\alpha)_m \cap (\Delta_{\overline{S}_\alpha}(\overrightarrow{V})_\alpha)_m$. By (IC5) ($\Delta_\mu(\Phi_1) \wedge \Delta_\mu(\Phi_2) \vdash \Delta_\mu(\Phi_1 \sqcup \Phi_2)$), we obtain $w \in (\Delta_{\overline{S}_\alpha}(\overrightarrow{U} \sqcup \overrightarrow{V})_\alpha)_m$. In particular, $w \in \bigcap f_S(\overrightarrow{U} \sqcup \overrightarrow{V}) \cap \overline{S}$. Therefore $\bigcap f_S(\overrightarrow{U}) \cap \bigcap f_S(\overrightarrow{V}) \cap \overline{S} \subset \bigcap f_S(\overrightarrow{U} \sqcup \overrightarrow{V}) \cap \overline{S}$.

(SC6) Take $w \in \bigcap f_S(\overrightarrow{U} \sqcup \overrightarrow{V}) \cap \overline{S}$, then

$$w \in ((\Delta_{\overline{S}_\alpha}(\overrightarrow{U} \sqcup \overrightarrow{V})_\alpha)_m \cup \big(\bigcap \overrightarrow{U} \cap \bigcap \overrightarrow{V}\big) \cap \overline{S}$$

as before $w \in (\Delta_{\overline{S}_\alpha}(\overrightarrow{U} \sqcup \overrightarrow{V})_\alpha)_m$. By hypothesis

$$\bigcap f_S(\overrightarrow{U}) \cap \bigcap f_S(\overrightarrow{V}) \cap \overline{S} \neq \varnothing$$

Thus, by definition, $(\Delta_{\overline{S}_\alpha}(\overrightarrow{U})_\alpha)_m \cap (\Delta_{\overline{S}_\alpha}(\overrightarrow{V})_\alpha)_m \neq \varnothing$, therefore $(\Delta_{\overline{S}_\alpha}(\overrightarrow{U})_\alpha) \wedge (\Delta_{\overline{S}_\alpha}(\overrightarrow{V})_\alpha)$ is consistent. By (IC6) we have

$$\Delta_{\overline{S}_\alpha}(\overrightarrow{U} \sqcup \overrightarrow{V})_\alpha \vdash (\Delta_{\overline{S}_\alpha}(\overrightarrow{U})_\alpha) \wedge (\Delta_{\overline{S}_\alpha}(\overrightarrow{V})_\alpha)$$

Thus $w \in (\Delta_{\overline{S}_\alpha}(\overrightarrow{U})_\alpha)_m \cap (\Delta_{\overline{S}_\alpha}(\overrightarrow{V})_\alpha)_m$. Therefore

$$w \in (\Delta_{\overline{S}_\alpha}(\overrightarrow{U})_\alpha)_m \cap (\Delta_{\overline{S}_\alpha}(\overrightarrow{V})_\alpha)_m$$

and from this we obtain

$$w \in \bigcap f_S(\overrightarrow{U}) \cap \bigcap f_S(\overrightarrow{V}) \cap \overline{S}$$

that is, SC6 is verified.

(SC7) We want to see that $\bigcap f_S(\overrightarrow{U}) \cap (\overline{S \cup T}) \subset f_{S\cup T}(\overrightarrow{U}) \cap (\overline{S \cup T})$. Take $w \in \bigcap f_S(\overrightarrow{U}) \cap (\overline{S \cup T})$. Then, $w \in ((\Delta_{(\overline{S})_\alpha}(\overrightarrow{U})_\alpha)_m) \cap \overline{T}$; by (IC7), we have

$$w \in (\Delta_{\overline{S}_\alpha \wedge \overline{T}_\alpha}(\overrightarrow{U})_\alpha)_m = (\Delta_{(\overline{S} \cap \overline{T})_\alpha}(\overrightarrow{U})_\alpha)_m$$

But notice that, by definition of f, we have

$$f_{S\cup T}(\overrightarrow{U}) = \overrightarrow{U} \cup (\Delta_{(\overline{S\cup T})_\alpha}(\overrightarrow{U}_\alpha))_m \text{ and } (\Delta_{(\overline{S\cup T})_\alpha}(\overrightarrow{U}_\alpha))_m = (\Delta_{\overline{S}_\alpha \wedge \overline{T}_\alpha}(\overrightarrow{U}_\alpha))_m$$

Thus, in particular, we have

$$w \in f_{S\cup T}(\overrightarrow{U}) \cap (\overline{S \cup T})$$

(SC8) Suppose $\bigcap f_S(\overrightarrow{U}) \cap (\overline{S \cup T}) \neq \varnothing$, that is $(\Delta_{(\overline{S})_\alpha}(\overrightarrow{U})_\alpha) \wedge (\overline{T})_\alpha$ is consistent. By (IC8), we have $\Delta_{(\overline{S})_\alpha \wedge (\overline{T})_\alpha}(\overrightarrow{U})_\alpha) \vdash \Delta_{(\overline{S})_\alpha}(\overrightarrow{U})_\alpha \wedge (\overline{T})_\alpha$. Then, if $w \in f_{S\cup T}(\overrightarrow{U}) \cap (\overline{S \cup T})$, by definition, $w \in (\Delta_{(\overline{S})_\alpha \wedge (\overline{T})_\alpha}(\overrightarrow{U})_\alpha))_m$. Thus, $w \in (\Delta_{\overline{S}_\alpha}(\overrightarrow{U})_\alpha)_m \cap \overline{T}$, therefore $w \in \bigcap f_S(\overrightarrow{U}) \cap (\overline{S \cup T})$.

This finishes the proof that the function f is a social contraction operator. ∎

Theorem 3 (From social contraction to merging). *Let f be a social contraction operator. Let Δ be the operator defined by Equation 4 (the generalized Levi identity). Then Δ is a merging operator with integrity constraints.*

Proof. Let Δ be defined by Equation 4, that is

$$\Delta_\mu(\Phi) = \left(\bigcap f_{(\neg\mu)_m}(\overrightarrow{\Phi}_m) \cap \mu_m\right)_\alpha$$

Recall that we associate to Φ a vector $\overrightarrow{\Phi}$ where all his elements appear. Then, to this vector of formulas we associate an s-profile $\overrightarrow{\Phi}_m$. By (SC0) we have $\sigma(f_S(\overrightarrow{U})) = f_S(\sigma(\overrightarrow{U}))$, for $\overrightarrow{U} \in \mathcal{B}^n$ and $\sigma \in S_n$. Thus $\bigcap f_S(\overrightarrow{U}) = \bigcap f_S(\sigma(\overrightarrow{U}))$, because intersection is commutative. Then $\bigcap f_{(\neg\mu)_m}(\overrightarrow{\Phi}_m) = \bigcap f_{(\neg\mu)_m}(\overrightarrow{U})$ for $\overrightarrow{U} \in \overrightarrow{\Phi}_m$. This means that Δ is well defined. Now we proceed to verify that Δ satisfies the postulates of Definition 1.

(IC0) This postulate is verified because $(\Delta_\mu(\Phi))_m \subseteq \mu_m$ and therefore $\Delta_\mu(\Phi) \vdash \mu$

(IC1) Suppose that μ is consistent, that is, $\mu_m \neq \varnothing$. By (SC2) we have $\widetilde{f}_{(\neg\mu)_m}(\overrightarrow{\Phi}_m) \neq \varnothing$ and by definition $\bigcap f_{(\neg\mu)_m}(\overrightarrow{\Phi}_m) \cap \mu_m \neq \varnothing$.That is, $\Delta_\mu(\Phi)$ is consistent.

(IC2) Suppose that Φ is consistent with μ. That is, $\bigwedge \Phi \wedge \mu$ is consistent. Then $\bigcap \overrightarrow{\Phi}_m \cap \mu_m \neq \varnothing$ and, by (SC3) we have $f_{(\neg\mu)_m}(\overrightarrow{\Phi}_m) = \overrightarrow{\Phi}_m$. Then, by definition,

$$\Delta_\mu(\Phi) \equiv \left(\bigcap \overrightarrow{\Phi}_m \cap \mu_m\right)_\alpha \equiv \bigwedge \Phi \wedge \mu$$

(IC3) Suppose that $\Phi \equiv \Psi$ and $\mu \leftrightarrow \nu$, that is, there exists a bijection g from Φ into Ψ such that $g(\varphi) \equiv \varphi$ for all $\varphi \in \Phi_1$. Thus, $\bigcap f_{(\neg\mu)_m}(\overrightarrow{\Phi}_m) = \bigcap f_{(\neg\mu)_m}(\overrightarrow{\Psi}_m)$, because $(g(\varphi))_m = (\varphi)_m$. Moreover $\mu_m = \nu_m$. Then

$$\bigcap f_{(\neg\mu)_m}(\overrightarrow{\Phi}_m) \cap \mu_m = \bigcap f_{(\neg\mu)_m}(\overrightarrow{\Psi}_m) \cap \nu_m$$

Thus,

$$\Delta_\mu(\Phi) \leftrightarrow \Delta_\nu(\Psi)$$

(IC4) Suppose $\varphi \vdash \mu$, $\varphi' \vdash \mu$ and $\Delta_\mu(\varphi \sqcup \varphi') \wedge \varphi' \nvdash \bot$. Then, $\varphi_m \subseteq \mu_m$, $\varphi'_m \subseteq \mu_m$ and $f^1_{(\neg\mu)_m}(\varphi_m, \varphi'_m) \cap f^2_{(\neg\mu)_m}(\varphi_m, \varphi'_m) \cap \varphi'_m \neq \varnothing$. Then $f^1_{(\neg\mu)_m}(\varphi_m, \varphi'_m) \cap \varphi'_m \neq \varnothing$. By (SC4), we have $f^2_{(\neg\mu)_m}(\varphi_m, \varphi'_m) \cap \varphi_m \neq \varnothing$ and, by (SC1)

$$\varphi_m \subseteq f^1_{(\neg\mu)_m}(\varphi_m, \varphi'_m)$$

Thus, $f^1_{(\neg\mu)_m}(\varphi_m, \varphi'_m) \cap f^2_{(\neg\mu)_m}(\varphi_m, \varphi'_m) \cap \varphi_m \neq \varnothing$, therefore

$$\Delta_\mu(\varphi \sqcup \varphi') \wedge \varphi \nvdash \bot$$

(IC5) Suppose $w \in (\Delta_\mu(\Phi) \wedge \Delta_\mu(\Psi))_m = (\Delta_\mu(\Phi))_m \cap (\Delta_\mu(\Psi))_m$. Then $w \in \bigcap f_{(\neg\mu)_m}(\overrightarrow{\Phi}_m) \cap \bigcap f_{(\neg\mu)_m}(\overrightarrow{\Psi}_m) \cap \mu_m$. By (SC5), we have

$$\bigcap f_{(\neg\mu)_m}(\overrightarrow{\Phi}_m) \cap \bigcap f_{(\neg\mu)_m}(\overrightarrow{\Psi}_m) \cap \mu_m \subseteq \bigcap f_{(\neg\mu)_m}(\overrightarrow{\Phi}_m \sqcup \overrightarrow{\Psi}_m) \cap \mu_m$$

Then $w \in \bigcap f_{(\neg\mu)_m}(\overrightarrow{\Phi}_m \sqcup \overrightarrow{\Psi}_m) \cap \mu_m = (\Delta_\mu(\Phi \sqcup \Psi))_m$. Thus,

$$\Delta_\mu(\Phi) \wedge \Delta_\mu(\Psi) \vdash \Delta_\mu(\Phi \sqcup \Psi)$$

(IC6) Suppose $\Delta_\mu(\Phi) \wedge \Delta_\mu(\Psi)$ is consistent, that is $\bigcap f_{(\neg\mu)_m}(\overrightarrow{\Phi}_m) \cap \bigcap f_{(\neg\mu)_m}(\overrightarrow{\Psi}_m) \cap \mu_m \neq \varnothing$. In this case the condition (SC6) says

$$\bigcap f_{(\neg\mu)_m}(\overrightarrow{\Phi}_m \sqcup \overrightarrow{\Psi}_m) \cap \mu_m \subset \bigcap f_{(\neg\mu)_m}(\overrightarrow{\Phi}_m) \cap \bigcap f_{(\neg\mu)_m}(\overrightarrow{\Psi}_m) \cap \mu_m$$

Thus,

$$\Delta_\mu(\Phi \sqcup \Psi) \vdash \Delta_\mu(\Phi) \wedge \Delta_\mu(\Psi)$$

(IC7) Suppose $w \in (\Delta_\mu(\Phi) \wedge \nu)_m$. By the definition of Δ, $w \in \bigcap f_{(\neg\mu)_m}(\overrightarrow{\Phi}_m) \cap \mu_m \cap \nu_m$. By (SC7), we have

$$\bigcap f_{(\neg\mu)_m}(\overrightarrow{\Phi}) \cap (\mu_m \cap \nu_m) \subset \bigcap f_{(\neg\mu\wedge\nu)}(\overrightarrow{\Phi}) \cap (\mu_m \cap \nu_m)$$

Thus, $w \in \bigcap f_{(\neg\mu\wedge\nu)}(\overrightarrow{\Phi}) \cap (\mu \wedge \nu)_m = (\Delta_{\mu\wedge\nu}(\Phi))_m$. Therefore,

$$\Delta_\mu(\Phi) \wedge \nu \vdash \Delta_{\mu\wedge\nu}(\Phi)$$

(IC8) Suppose that $\Delta_\mu(\Phi) \wedge \nu$ is consistent, that is

$$\left(\bigcap f_{(\neg\mu)_m}(\overrightarrow{\Phi}) \cap \mu_m\right) \cap \nu_m \neq \varnothing$$

Condition (SC8) says in this case

$$\bigcap f_{(\neg\mu\wedge\nu)_m}(\overrightarrow{\Phi}) \cap \mu_m \cap \nu_m \subset \bigcap f_{(\neg\mu)_m}(\overrightarrow{\Phi}) \cap (\mu_m \cap \nu_m)$$

Then

$$\Delta_{\mu\wedge\nu}(\Phi) \vdash \Delta_\mu(\Phi) \wedge \nu$$

This completes the proof. ∎

Let's denote by $\mathcal{SC}$ the functional mapping a merging operator Δ into a social contraction operator through the generalized Harper identity (Equation (3)). Let's denote by $\mathcal{O}$ the functional mapping a social contraction operator f into a merging operator through the generalized Levi identity (Equation (4)). Actually, these functionals are one the inverse of the other. More precisely we have the following result:

Theorem 4 (Equivalence between merging and social contraction). *Let $\mathcal{SC}$ and $\mathcal{O}$ be the functionals previously defined. Then for any merging operator Δ and any social contraction operator f we have*

$$\mathcal{O}(\mathcal{SC}(\Delta)) = \Delta \quad \text{and} \quad \mathcal{SC}(\mathcal{O}(f)) = f$$

Proof. Define $S = (\neg\mu)_m$ and $\overrightarrow{U} = \overrightarrow{\Phi}_m$. If Δ is a merging operator, by Equation (3)

$$f_{(\neg\mu)_m}(\overrightarrow{\Phi}_m) = \overrightarrow{\Phi}_m \cup (\Delta_\mu(\overrightarrow{\Phi}))_m$$

Then $\bigcap f_{(\neg\mu)_m}(\overrightarrow{\Phi}_m) \cap \mu_m = \bigcap(\overrightarrow{\Phi}_m \cup (\Delta_\mu(\overrightarrow{\Phi}))_m) \cap \mu_m$. From this, it follows

$$\bigcap f_{(\neg\mu)_m}(\overrightarrow{\Phi}_m) \cap \mu_m = \big(\bigcap \overrightarrow{\Phi}_m \cap \mu_m\big) \cup (\Delta_\mu(\Phi))_m = (\Delta_\mu(\Phi))_m$$

The last equality is obtained in the following way: either $\bigcap \overrightarrow{\Phi}_m \cap \mu_m = \varnothing$ and the equality is obvious; or $\bigcap \overrightarrow{\Phi}_m \cap \mu_m \neq \varnothing$ and, by (IC2), this is $(\Delta_\mu(\Phi))_m$. From this the equality is straightforward. This proves that $\mathcal{O}(\mathcal{SC}(\Delta)) = \Delta$.

Now let f be a social contraction operator. Let Δ be the merging operator obtained by Equation (4). Consider $S = (\neg\mu)_m$ and $\overrightarrow{U} = \overrightarrow{\Phi}_m$. Then

$$\Delta_{\overline{S}_\alpha}(\overrightarrow{U}_\alpha) \leftrightarrow \Big(\bigcap f_S(\overrightarrow{U}) \cap \overline{S}\Big)_\alpha$$

If $g(\overrightarrow{U}) = \overrightarrow{U} \cup (\Delta_{\overline{S}_\alpha}(\overrightarrow{U}_\alpha))_m$, then $g(\overrightarrow{U}) = \overrightarrow{U} \cup (\bigcap f_S(\overrightarrow{U}) \cap \overline{S})$. Suppose that $w \in g^i(\overrightarrow{U})\backslash U_i$. Then, $w \in (\Delta_{\overline{S}_\alpha}(\overrightarrow{U}_\alpha))_m$. Therefore $w \in \bigcap f_S(\overrightarrow{U}) \cap \overline{S}$. From this, it follows

$$g(\overrightarrow{U}) \subset \overrightarrow{U} \cup \big(\bigcap f_S(\overrightarrow{U}) \cap \overline{S}\big)$$

But $\overrightarrow{U} \cup (\bigcap f_S(\overrightarrow{U}) \cap \overline{S}) \subset \overrightarrow{U} \cup f_S(\overrightarrow{U}) = f_S(\overrightarrow{U})$. Thus

$$g(\overrightarrow{U}) \subset f_S(\overrightarrow{U})$$

If $w \in f_S^i(\overrightarrow{U})\backslash U_i$, by ($\mathrm{SC_m}$) and (SC1) we have $w \in \bigcap f_S(\overrightarrow{U}) \cap \overline{S}$. Thus, if $w \in f_S^i(\overrightarrow{U})$, then $w \in U_i \cup (\bigcap f_S(\overrightarrow{U}) \cap \overline{S}) = g^i(\overrightarrow{U})$. Therefore

$$f_S(\overrightarrow{U}) \subset g(\overrightarrow{U})$$

That is

$$f_S(\overrightarrow{U}) = g(\overrightarrow{U})$$

This proves that $\mathcal{SC}(\mathcal{O}(f)) = f$. ∎

As a corollary of the previous theorem and of the representation theorem for merging operators (Theorem 1) we obtain the following representation theorem for social contraction operators:

Theorem 5 (Representation for social contraction). *f is a social contraction operator iff there exists a syncretic assignment mapping each s-profile $\overrightarrow{U}$ into a total preorder $\leq_{\overrightarrow{U}}$ such that*

$$f_S(\overrightarrow{U}) = \overrightarrow{U} \cup \min(\overline{S}, \leq_{\overrightarrow{U}}).$$

6 Concluding Remarks

We have established the duality between merging operators with integrity constraints and the social contraction operators through generalized Harper and

Levi identities. This duality extends the well known duality between revision operators and contraction operators. Actually, in order to prove this duality we have made use of the equivalence between formulas and sets of models in the finite propositional framework.

The presentation of social contraction operators in a semantical (set theoretical) setting, opens the possibility of application of these operators in other domains, e.g. Decision theory or Social choice theory.

The representation theorem for social contraction operators, Theorem 5, is very useful because all techniques for building syncretic assignments, can be directly imported, in order to build social contraction operators.

The duality between between merging operators with integrity constraints and the social contraction operators gives some other results without effort. For instance, the social contraction operators are a true generalization of AGM contraction operators. This is because the merging operators are a true generalization of AGM revision operators (see [20]).

Some work in perspective is to explore the relationships between the social belief remove operators introduced in [9], in particular the notion of equilibrium therein, and a similar notion (to be found) for merging operators.

References

1. Alchourrón, C.E., Gärdenfors, P., Makinson, D.: On the logic of theory change: Partial meet contraction and revision functions. Journal of Symbolic Logic 50, 510–530 (1985)
2. Alchourrón, C.E., Makinson, D.: The logic of theory change: Contraction functions and their associated revision functions. Theoria 48, 14–37 (1982)
3. Alchourrón, C.E., Makinson, D.: On the logic of theory change: Safe contraction. Studia Logica 44, 405–422 (1985)
4. Alchourrón, C.E., Makinson, D.: Maps between some different kinds of contraction function: the finite case. Studia Logica 45, 187–198 (1986)
5. Baral, C., Kraus, S., Minker, J.: Combining multiple knowledge bases. IEEE Transactions on Knowledge and Data Engineering 3(2), 208–220 (1991)
6. Baral, C., Kraus, S., Minker, J., Subrahmanian, V.S.: Combining knowledge bases consisting of first-order theories. Computational Intelligence 8(1), 45–71 (1992)
7. Booth, R.: Social contraction and belief negotiation. In: Proceedings of the Eighth Conference on Principles of Knowledge Representation and Reasoning (KR 2002), pp. 374–384 (2002)
8. Booth, R.: Social contraction and belief negotiation. Information Fusion 7(1), 19–34 (2006)
9. Booth, R., Meyer, T.: Equilibria in social belief removal. Synthese 177(supplement-1), 97–123 (2010)
10. Chacón, J.L., Pino Pérez, R.: Merging operators: Beyond the finite case. Information Fusion 7(1), 41–60 (2006)
11. Dupin de Saint-Cyr, F., Lang, J.: Belief extrapolation (or how to reason about observations and unpredicted change). Artif. Intell. 175(2), 760–790 (2011)
12. Dupin de Saint-Cyr, F., Lang, J.: Belief extrapolation (or how to reason about observations and unpredicted change). In: Proceedings of the Eighth Conference on Principles of Knowledge Representation and Reasoning (KR 2002), pp. 497–508 (2002)

13. Gärdenfors, P.: Knowledge in flux. MIT Press (1988)
14. Gärdenfors, P. (ed.): Belief Revision. Cambridge University Press (1992)
15. Harper, W.L.: Rational conceptual change. In: PSA 1976 East Lansing, vol. 12, pp. 462–494. Philosophy of Science Association, Mich. (1977)
16. Herzig, A., Rifi, O.: Update operations: a review. In: Proceedings of the Thirteenth European Conference on Artificial Intelligence (ECAI 1998), pp. 13–17 (1998)
17. Katsuno, H., Mendelzon, A.O.: On the difference between updating a knowledge base and revising it. In: Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning (KR 1991), pp. 387–394 (1991)
18. Katsuno, H., Mendelzon, A.O.: Propositional knowledge base revision and minimal change. Artificial Intelligence 52, 263–294 (1991)
19. Konieczny, S., Lang, J., Marquis, P.: DA^2 merging operators. Artificial Intelligence 157(1-2), 49–79 (2004)
20. Konieczny, S., Pino Pérez, R.: Merging information under constraints: a logical framework. Journal of Logic and Computation 12(5), 773–808 (2002)
21. Konieczny, S., Pino Pérez, R.: Logic based merging. Journal of Philosophical Logic 40, 239–270 (2011)
22. Levi, I.: Subjunctives, dispositions and chances. Synthese 34, 423–455 (1977)
23. Liberatore, P., Schaerf, M.: Arbitration (or how to merge knowledge bases). IEEE Transactions on Knowledge and Data Engineering 10(1), 76–90 (1998)
24. Lin, J.: Integration of weighted knowledge bases. Artificial Intelligence 83(2), 363–378 (1996)
25. Lin, J., Mendelzon, A.O.: Merging databases under constraints. International Journal of Cooperative Information System 7(1), 55–76 (1998)
26. Lin, J., Mendelzon, A.O.: Knowledge base merging by majority. In: Dynamic Worlds: From the Frame Problem to Knowledge Management. Kluwer (1999)
27. Lobo, J., Uzcátegui, C.: Abductive change operators. Fundamenta Informaticae 27(4), 385–411 (1996)
28. Revesz, P.Z.: On the semantics of theory change: arbitration between old and new information. In: Proceedings of the 12th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Databases, pp. 71–92 (1993)
29. Revesz, P.Z.: On the semantics of arbitration. International Journal of Algebra and Computation 7(2), 133–160 (1997)

Automatic Generation of Invariants for Circular Derivations in SUP(LA)*

Arnaud Fietzke, Evgeny Kruglov, and Christoph Weidenbach

Max-Planck-Institut für Informatik, Saarbrücken, Germany
Saarland University – Computer Science, Saarbrücken, Germany
{fietzke,ekruglov,weidenbach}@mpi-inf.mpg.de

Abstract. The hierarchic combination of linear arithmetic and first-order logic with free function symbols, FOL(LA), results in a strictly more expressive logic than its two parts. The SUP(LA) calculus can be turned into a decision procedure for interesting fragments of FOL(LA). For example, reachability problems for timed automata can be decided by SUP(LA) using an appropriate translation into FOL(LA). In this paper, we extend the SUP(LA) calculus with an additional inference rule, automatically generating inductive invariants from partial SUP(LA) derivations. The rule enables decidability of more expressive fragments, including reachability for timed automata with unbounded integer variables. We have implemented the rule in the SPASS(LA) theorem prover with promising results, showing that it can considerably speed up proof search and enable termination of saturation for practically relevant problems.

1 Introduction

One important aspect for successful development of automated reasoning calculi for logical languages is the potential of the calculus to act as a decision procedure for known decidable classes and to be an instrument for detecting new decidable fragments. This is because a sound and complete calculus for some logical language that can at the same time be used as a decision procedure has a high potential to be successfully applied in practice. The superposition calculus has been very successful in this respect for first-order logic, e.g., [3,10,17]. This is further illustrated by the fact that the leading first-order ATPs (E, SPASS, Vampire) are all superposition-based.

In this paper we continue this line of work for the FOL(LA) language, the hierarchic combination of first-order logic with linear arithmetic. The hierarchic superposition calculus SUP(LA) [1] is a sound calculus for FOL(LA) and together with a sufficient completeness assumption, also complete. Completeness cannot be achieved in general, because the FOL(LA) language can express second-order properties. For example, starting with LA over the reals, the naturals can be expressed in FOL(LA) [18] and it is known that the addition of a

* This work has been partly supported by the German Transregional Collaborative Research Center SFB/TR 14 AVACS.

N. Bjørner and A. Voronkov (Eds.): LPAR-18, LNCS 7180, pp. 197–211, 2012.

single monadic predicate to the LA language already causes undecidability [15], in general.

Nevertheless, the SUP(LA) calculus is a decision procedure for the FOL(LA) ground case [19] and for the FOL(LA) fragment resulting from the translation of timed automata [13]. In this paper we extend the latter result to the fragment corresponding to the translation of timed automata extended with unbounded integer variables. Termination of the SUP(LA) calculus on this fragment is made possible by a new simplification technique based on the automatic generation of inductive invariants. The invariant generation rule combines ideas from acceleration for automata [16,5] with the automatic detection of infinite loops [20] in SUP(LA) derivations.

The following example illustrates the basic idea: assume we have used the clause $x = 1 \parallel \rightarrow P(x)$ in a derivation of $x = 2 \parallel \rightarrow P(x)$ (clauses are in purified form: arithmetic literals to the left of $\parallel$, first-order literals to the right; $x=1 \parallel \rightarrow P(x)$ means $\forall x(x=1 \rightarrow P(x))$). Depending on how it was derived, the same sequence of inferences may be applied to the second clause, yielding a third clause with right-hand side $P(x)$. For instance, the second clause may have been obtained by resolving the first one with $x'=x+1 \parallel P(x) \rightarrow P(x')$. Then we could also derive $x=3 \parallel \rightarrow P(x)$, $x=4 \parallel \rightarrow P(x)$ and so on. The idea of the invariant generation rule is to detect such loops during proof search, in the form of clauses with the same free (i.e., non-arithmetic) part (up to variable renaming), and to determine the transformation relating their arithmetic constraints. If it is possible to express the transitive closure of this transformation as a conjunction of arithmetic literals, then a corresponding invariant clause is derived. In the above example, such a clause would be $k \geq 1, x = k \parallel \rightarrow P(x)$, where k is an integer variable.

This paper is organized as follows: Section 2 gives some preliminary definitions relating to superposition modulo linear arithmetic. Section 3 defines the constraint induction rule in its general form, and presents a class of linear arithmetic constraints for which it can be effectively implemented. In Section 4, we define timed automata extended with unbounded integer variables, and we show that SUP(LA) together with the constraint induction rule provides a decision procedure for the corresponding reachability problem. Section 5 deals with our implementation of the rule and shows some promising experimental results. We end with a summary of the results and an outlook in Section 6. Detailed definitions and proofs can be found in a technical report [12].

2 Preliminaries

We will use the notions and notations for hierarchic superposition modulo linear arithmetic SUP(LA) [4,1]. In SUP(LA), clauses appear in purified form $\Lambda \parallel \Gamma \rightarrow \Delta$ where Λ is a sequence of linear arithmetic literals over real and integer variables, called the *clause constraint*, and Γ, Δ are sequences of free first-order atoms, called the *free part*, sharing universally quantified variables with Λ. Semantically, a clause $\Lambda \parallel \Gamma \rightarrow \Delta$ is interpreted as the universal

closure of the implication $(\bigwedge \Lambda \wedge \bigwedge \Gamma) \rightarrow \bigvee \Delta$. A constrained empty clause $\Lambda \parallel \square$ represents a contradiction if Λ is satisfiable.

We use lowercase Latin characters x, y, z to denote variables. Vectors of variables are denoted by boldface characters ($\mathbf{x}$). We use the notation $\Lambda[\mathbf{x}]$ to mean that $\mathbf{x}$ are the variables occurring in Λ. When $\mathbf{x}$ is clear from the context, we also denote by $\Lambda[\mathbf{y}]$ the result of substituting all occurrences of variables from $\mathbf{x}$ in Λ by the corresponding variables from $\mathbf{y}$. Substitutions are denoted by lowercase Greek letters (σ, τ). A substitution is called *simple*, if it maps every variable of arithmetic sort to an arithmetic term.

The overall superposition calculus is based on a reduction ordering that is total on ground atoms. In particular all ground terms of the arithmetic sort containing only arithmetic symbols are assumed to be strictly smaller than any ground term containing a free function symbol. For example, this can be achieved by an LPO (lexicographic path ordering) where the arithmetic symbols are smaller in the precedence than any free symbol. This ordering on the ground atoms is then lifted via the usual twofold multiset extension to clauses. A ground clause C is *redundant* in some clause set N, if it follows from smaller clauses in N. Redundancy is lifted by instantiation to clauses with variables.

To keep the presentation simple, we use superposition left (ordered resolution) the only inference rule, and subsumption as the only reduction rule. We will not need factoring for the types of clause sets considered in this paper.

A clause $C_1 = \Lambda_1 \parallel \Gamma_1 \rightarrow \Delta_1$ *subsumes* a clause $C_2 = \Lambda_2 \parallel \Gamma_2 \rightarrow \Delta_2$ if there is a substitution σ such that $\Gamma_1\sigma \subseteq \Gamma_2$, $\Delta_1\sigma \subseteq \Delta_2$ and $\forall \mathbf{x}\, \exists \mathbf{y}\, (\Lambda_2 \rightarrow \Lambda_1\sigma)$ holds in the theory of linear arithmetic, where $\mathbf{x}$ are the variables occurring in Λ_2 and $\mathbf{y}$ the variables occurring in $\Lambda_1\sigma$ but not in Λ_2. Note that in theorem proving derivations, forward subsumption (i.e. removing a newly derived clause which is subsumed by an old clause) does not need to be strict to maintain completeness.

The ordered resolution rule is

$$\frac{\Lambda_1 \parallel \Gamma_1, A \rightarrow \Delta_1 \qquad \Lambda_2 \parallel \Gamma_2 \rightarrow \Delta_2, B}{\Lambda_3 \parallel (\Gamma_1, \Gamma_2 \rightarrow \Delta_1, \Delta_2)\sigma}$$

such that σ is the most general simple unifier of A and B; A is strictly maximal in $\Gamma_1, A \rightarrow \Delta_1$; B is strictly maximal in $\Gamma_2 \rightarrow \Delta_2, B$.

The calculus SUP(LA) is complete for clause sets that enjoy *sufficient completeness*, meaning that every ground non-arithmetic term is equal to some arithmetic ground term. A sufficient condition for a clause set to be sufficiently complete is the absence of function symbols ranging into the arithmetic sorts (real or integer).

3 Constraint Induction

Given a relation $R \subseteq \mathbb{R}^{2n}$, the composition $R \circ R$ is the relation such that $(R \circ R)(x_1, \ldots, x_n, x'_1, \ldots, x'_n)$ holds if and only if there exist $y_1, \ldots, y_n$ such that $R(x_1, \ldots, x_n, y_1, \ldots, y_n)$ and $R(y_1, \ldots, y_n, x'_1, \ldots, x'_n)$. If we define $R^1 =$

R and $R^k = R^{k-1} \circ R$, then the *transitive closure* of R is the relation R^+ such that $R^+(x_1, \ldots, x_n, x'_1, \ldots, x'_n)$ if and only if there exists $k \geq 1$ such that $R^k(x_1, \ldots, x_n, x'_1, \ldots, x'_n)$.

If a clause in a derivation has an ancestor (i.e., a clause to which it is related by a sequence of rule applications) with the same free part (modulo variable renaming), then the clause can be used to derive a third clause with the same free part, and so on. This yields a potentially infinite sequence of inferences, where clauses differing only in the arithmetic constraint are being derived.

The idea of the invariant generation rule is to find the transformation relating the constraints along the sequence and to compute its transitive closure. To find the transformation, the sequence of inferences is applied to a parameterized version of the initial clause, as shown in Figure 1. If the closure can itself be expressed as a constraint, then we can derive a corresponding *inductive invariant clause* which can be used to subsume all its instances, thereby avoiding repeated applications of the same sequence of inference rules.

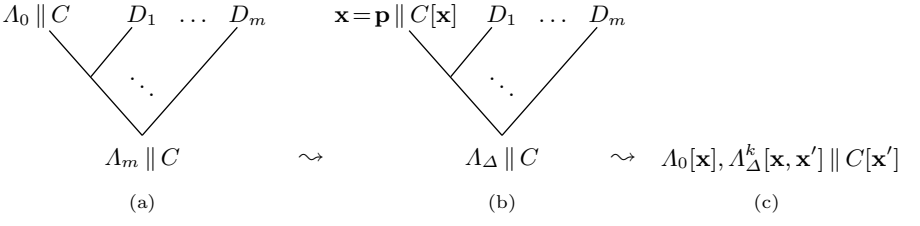


Fig. 1. After a loop has been detected during proof search (a), the corresponding inferences are replayed on a parameterized clause (b) and the inductive invariant clause is derived (c).

The parameterized clause is of the form $x_1 = p_1, \ldots, x_n = p_n \parallel C[x_1, \ldots, x_n]$, where p_i are fresh parameters (i.e., arithmetic constants) not appearing anywhere in the clause set, one for each arithmetic variable in the clause $\Lambda_0 \parallel C$. After the inferences leading from $\Lambda_0 \parallel C$ to $\Lambda_m \parallel C$ have been performed on the parameterized clause, a clause of the form $\Lambda_\Delta \parallel C$ is obtained. This replaying of inferences is always possible, because the SUP(LA) calculus does not take the clause constraints into account when deciding which inferences to perform (the constraints are only considered when testing for subsumption, or when checking satisfiability of an empty clause's constraint). Also note that the parameters p_i are introduced only for the purpose of replaying the derivation, and do never appear in the actual clause set, thus they play no semantic role. The constraint Λ_Δ will contain variables from the free part, as well as parameters p_i, which stand for the constraint variables of the original parameterized clause[1].

[1] Possibly after simplification and variable elimination to get rid of variables not occurring in C.

Example 1. Consider the inference

$$\frac{x{=}1 \parallel \ \rightarrow P(x) \qquad x'{=}x{+}1 \parallel P(x) \rightarrow P(x')}{x{=}2 \parallel \ \rightarrow P(x)}$$

from the introduction. We would now perform the inference

$$\frac{x{=}p \parallel \ \rightarrow P(x) \qquad x'{=}x{+}1 \parallel P(x) \rightarrow P(x')}{x{=}p{+}1 \parallel \ \rightarrow P(x)}$$

to get $x = p + 1$ as Λ_Δ.

If we replace the parameters by their corresponding variables, and replace the remaining variables by their primed versions, we obtain $\Lambda_\Delta[x_1, \ldots, x_n, x'_1, \ldots, x'_n]$, which describes a relation[2] $R_\Delta \subseteq \mathbb{R}^{2n}$. We write Λ^k_Δ for the constraint representing R^k_Δ, if it exists. This constraint will in general contain k as an additional integer variable (we chose k to be distinct from all x_i, x'_i). Note that $(\Lambda_0[\mathbf{x}] \wedge \Lambda^k_\Delta[\mathbf{x}, \mathbf{x}', k])\{k \mapsto 1\}$ is equivalent to $\Lambda_m[\mathbf{x}]$.

Definition 2 (Constraint Induction). *Let N be a clause set containing two clauses $\Lambda_0 \parallel C, \Lambda_m \parallel C$ with identical free part (up to variable renaming) such that $\Lambda_m \parallel C$ was derived from $\Lambda_0 \parallel C$ using clauses $D_1, \ldots, D_m$ in N. The constraint induction rule is the inference rule*

$$\frac{\Lambda_0 \parallel C \quad D_1 \ \ldots \ D_m \quad \Lambda_m \parallel C}{\Lambda_0[\mathbf{x}], \Lambda^k_\Delta[\mathbf{x}, \mathbf{x}', k] \parallel C[\mathbf{x}']}$$

where Λ_Δ is the constraint obtained by replaying the derivation as described above.

Proposition 3 (Soundness of Constraint Induction). *Let N be a clause set, and assume $\Lambda_0[\mathbf{x}], \Lambda^k_\Delta[\mathbf{x}, \mathbf{x}', k]$ was derived from $\Lambda_0 \parallel C, D_1, \ldots, D_m, \Lambda_m \parallel C \in N$ by constraint induction. Then $N \models \Lambda_0[\mathbf{x}], \Lambda^k_\Delta[\mathbf{x}, \mathbf{x}', k]$.*

Proof:

$$\begin{aligned}
& C[\mathbf{p}], D_1, \ldots, D_m \models \Lambda_\Delta[\mathbf{p}, \mathbf{x}'] \rightarrow C[x'] && (1)\\
\Longrightarrow \quad & D_1, \ldots, D_m \models (C[\mathbf{p}] \wedge \Lambda_\Delta[\mathbf{p}, \mathbf{x}']) \rightarrow C[\mathbf{x}'] && (2)\\
\Longrightarrow \quad & D_1, \ldots, D_m \models (C[\mathbf{x}] \wedge \Lambda_\Delta[\mathbf{x}, \mathbf{x}']) \rightarrow C[\mathbf{x}'] && (3)\\
\Longrightarrow \quad & D_1, \ldots, D_m \models \big(C[\mathbf{x}] \wedge \Lambda^k_\Delta[\mathbf{x}, \mathbf{x}', k]\big) \rightarrow C[\mathbf{x}'] && (4)\\
\Longrightarrow \quad & N \models \big(\Lambda[\mathbf{x}] \wedge \Lambda^k_\Delta[\mathbf{x}, \mathbf{x}', k]\big) \rightarrow C[\mathbf{x}'] && (5)
\end{aligned}$$

(1) holds by soundness of SUP(LA) and the fact that $\mathbf{x} = \mathbf{p} \parallel C[\mathbf{x}]$ is equivalent to $C[\mathbf{p}]$; (2) follows because $C[\mathbf{p}]$ is ground; (3) follows because the $\mathbf{p}$ do not

[2] Some parameters and variables may not occur in Λ_Δ, we may then just consider them to be unconstrained, i.e., they can take any value in $\mathbb{R}$.

occur outside of $C[\mathbf{p}]$ and $\Lambda_\Delta[\mathbf{p}, \mathbf{x}']$; (4) follows by induction on k; (5) follows because $D_1, \ldots, D_m \in N$ and $N \models \Lambda[\mathbf{x}] \rightarrow C[\mathbf{x}]$.

Of course, the constraint induction rule is only applicable if Λ^k_Δ exists and can be effectively computed. We will now look at a class of linear arithmetic constraints for which this is always the case. Given two relations $R_1 \subseteq \mathbb{R}^{2n}$ and $R_2 \subseteq \mathbb{R}^{2m}$, the *product of* R_1, R_2 is the relation $R \subseteq \mathbb{R}^{2(m+n)}$ such that $R(\mathbf{x}, \mathbf{y}, \mathbf{x}', \mathbf{y}')$ if and only if $R_1(\mathbf{x}, \mathbf{x}')$ and $R_2(\mathbf{y}, \mathbf{y}')$, where $\mathbf{x} = x_1, \ldots, x_n$ and $\mathbf{y} = y_1, \ldots, y_m$. If R is the product of R_1, R_2, then $R^k(\mathbf{x}, \mathbf{y}, \mathbf{x}', \mathbf{y}')$ if and only if $R_1^k(\mathbf{x}, \mathbf{x}')$ and $R_2^k(\mathbf{y}, \mathbf{y}')$. Hence we can compute the transitive closure of a product relation if we can compute the transitive closure for each component relation.

Proposition 4. *Let* $R(x_1, \ldots, x_n, x'_1, \ldots, x'_n) \subseteq \mathbb{R}^{2n}$ *be defined by*

$$\bigwedge_{i \in I} x_i + \alpha_{ij} x_j + a_i \ \# \ x'_i \ \wedge \bigwedge_{\alpha_{ij} \neq 0} x'_j = 0$$

for $I \subseteq \{1, \ldots, n\}$, $\alpha_{ij} \in \mathbb{R}$, $\alpha_{ii} = 0$ *for all* $1 \leq i \leq n$, $a_i \in \mathbb{R} \cup \{-\infty, \infty\}$ *and* $\# \in \{<, \leq, \geq, >\}$. *Then* $R^k(x_1, \ldots, x_n, x'_1, \ldots, x'_n)$ *holds if and only if*

$$\bigwedge_{i \in I} x_i + \alpha_{i,j} x_j + k a_i \ \# \ x'_i \ \wedge \bigwedge_{\alpha_{ij} \neq 0} x'_j = 0$$

Proof: By induction on k.

Proposition 5. *Let* $R(x_1, \ldots, x_n, x'_1, \ldots, x'_n) \subseteq \mathbb{R}^{2n}$ *be defined by*

$$\bigwedge_{l=1}^{m} \sum_{j \in J} \beta_{lj} x_j \leq d_l \ \wedge \bigwedge_{j \in J} x'_j = \delta_j x_j + c_j$$

for $J \subseteq \{1, \ldots, n\}$, $m \geq 1$, $\delta_j \in \{0, 1\}$ *and* $c_j, \beta_{lj}, d_l \in \mathbb{R}$. *Then* $R^k(x_1, \ldots, x_n, x'_1, \ldots, x'_n)$ *holds for* $k \geq 2$ *if and only if*

$$\bigwedge_{l=1}^{m} \left(\sum_{j \in J} \beta_{lj} x_j \leq d_l \ \wedge \sum_{j \in J} \beta_{lj} \left(\delta_j \left(x_j + (k-2) c_j \right) + c_j \right) \leq d_l \right)$$
$$\wedge \bigwedge_{j \in J} x'_j = \delta_j \left(x_j + (k-1) c_j \right) + c_j$$

Proof: A straightforward proof using matrix operations can be found in [5].

In the following, we will apply the induction rule to constraints that describe products of the kinds of relations described in Propositions 4 and 5. It turns out that this is sufficient to turn SUP(LA) with constraint induction into a decision procedure for timed automata extended with unbounded integer variables, as

long as they satisfy certain flatness properties (Section 4) and also speed up proof search, shorten proofs and enable termination of saturation for other kinds of problems (Section 5).

If we don't insist on being able to express the transitive closure as a single conjunction, then it becomes possible to compute the transitive closure of more involved types of constraints [8,21,14,6]. For instance, if the closure can be expressed in Presburger arithmetic, we can derive several clauses that together constitute the inductive invariant (by expressing the closure in disjunctive normal form and introducing one clause per disjunct). For the time being, we restrict ourselves to constraints of the above form, as this already yields nice results. We plan to investigate extensions of the rule in future work.

4 Finite Saturation of Extended Timed Automata

For a set of variables X, the sets $\mathsf{CC}(X)$, $\mathsf{IG}(X)$ and $\mathsf{IA}(X)$ of *clock constraints* and *integer guards*, respectively, are defined as

$$\mathsf{CC}(X) : \mathrm{cc} ::= x \circ c \mid x - y \circ c \mid \mathrm{cc} \wedge \mathrm{cc}$$
$$\mathsf{IG}(X) : \mathrm{ig} ::= a_1 x_1 + \cdots + a_n x_n \leq a \mid \mathrm{ig} \wedge \mathrm{ig}$$

where $x \in X$, $c \in \mathbb{N}$, $\circ \in \{<, \leq, =, \geq, >\}$, and $a_i, a \in \mathbb{Z}$. The set $\mathsf{IA}(X)$ of *integer assignments* consists of all substitutions mapping each $x \in X$ to a term of the form a or $x + a$, for $a \in \mathbb{Z}$.

Definition 6 (Extended Timed Automaton). *An* extended timed automaton *is a tuple*

$$\mathcal{T} = (L, l^0, X, \mathrm{ig}_0, \{\mathrm{inv}_l\}_{l \in L}, E)$$

where L is a finite set of locations *with* initial location $l^0 \in L$*, X is a finite set of variables partitioned into subsets X_C, X_D of real-valued* clock variables *and integer-valued variables, respectively;* $\mathrm{ig}_0 \in \mathsf{IG}(X_D)$ *describes the initial values of the integer variables;* $\mathrm{inv}_l \in \mathsf{CC}(X_C)$ *is the* invariant *of location l;* $E \subseteq L \times \mathsf{CC}(X_C) \times \mathsf{IG}(X_D) \times \mathsf{IA}(X_D) \times 2^{X_C} \times L$ *is a finite set of* edges*. An edge $(l, \mathrm{cc}, \mathrm{ig}, \mathrm{ia}, Z, l')$ represents a transition from location l to location l'. The constraints* cc *and* ig *determine when the edge is enabled, and the set Z contains the clocks to be reset to zero when taking the edge, together with the assignment* ia*. If $X = X_C$, $\mathcal{T}$ is a classical timed automaton [2,13].*

States of an extended timed automaton are tuples (l, ν) consisting of a location $l \in L$ and a valuation $\nu \in \mathbb{R}^X$ for all clocks and integer variables. The initial states are of the form (l^0, ν_0) where ν_0 assigns zero to all clocks and the values of integer variables satisfy ig_0. The automaton can stay in a location as long as the clock values satisfy the location's invariant. When the valuation of a state satisfies the guards cc and ig of an outgoing edge, the corresponding transition can be taken, resetting the clocks in Z and applying the assignment ia.

Let $\mathcal{T} = (L, l^0, X, \mathrm{ig}_0, \{\mathrm{inv}_l\}_{l \in L}, E)$ be an extended timed automaton. The encoding of reachability for extended timed automata is analogous to that for classical timed automata [13], except that clauses encoding discrete transitions now also include integer guards and assignments. We use a reachability predicate Reach, and constant symbols $l \in L$ for every location[3]. The vector $\mathbf{x}$ contains the clock variables variables X_C, $\mathbf{z}$ contains the integer variables X_D. The clause

$$\mathbf{x}{=}0,\ \ \mathrm{ig}_0(\mathbf{z}) \parallel \ \rightarrow \mathrm{Reach}(\mathbf{x}, \mathbf{z}, l^0).$$

encodes reachability of the initial states. For every location $l \in L$,

$$t{\geq}0,\ \mathbf{x}'{=}\mathbf{x}{+}t,\ \ \mathrm{inv}_l[\mathbf{x}'] \parallel \ \mathrm{Reach}(\mathbf{x}, \mathbf{z}, l) \rightarrow \mathrm{Reach}(\mathbf{x}', \mathbf{z}, l).$$

encodes time-reachability for location l. For a variable x and set of variables Z, we define the substitution ρ_Z to be $\rho_Z(x) = 0$ if $x \in Z$, and $\rho_Z(x) = x$ otherwise, and we extend it to vectors of variables pointwise. For every edge $e = (l, \mathrm{cc}, \mathrm{ig}, \mathrm{ia}, Z, l')$ in E, the clause

$$\mathrm{cc}[\mathbf{x}],\ \mathbf{x}'{=}\rho_Z(\mathbf{x}),\ \mathrm{ig}(\mathbf{z}),\ \mathbf{z}'{=}\mathrm{ia}(\mathbf{z}),\ \mathrm{inv}_{l'}[\mathbf{x}'] \parallel \ \mathrm{Reach}(\mathbf{x}, \mathbf{z}, l) \rightarrow \mathrm{Reach}(\mathbf{x}', \mathbf{z}', l').$$

represents the discrete transition from l to l' via e. A *reachability conjecture* is a clause of the form $\Lambda \parallel \ \mathrm{Reach}(\mathbf{x}, \mathbf{z}, l) \rightarrow$.

The states described by the reachability conjecture are reachable if and only if the empty clause can be derived from the clause set. In [13], we show how to ensure that the positive literals of such clauses are always strictly maximal in the clause. This guarantees that starting from the encoding of an extended timed automaton and one (or more) reachability conjecture, only negative unit clauses can be derived (that's why we don't need factoring). The inferences correspond to a backward traversal of the automaton's state space, starting from the states represented by the reachability conjecture. This restriction to backward traversal ensures termination of saturation for the encoding of classical timed automata (without integer variables). In the case of extended timed automata, this alone is no longer sufficient, since the assignments to the integer variables cannot be assumed to be monotonic. Thus assignments to integer variables that occur on a cycle may lead to non-termination of saturation, because such a cycle will induce a loop during proof search. This loop however can be handled by the constraint induction rule if the clock constraints and clock resets on such a cycle satisfy certain properties.

Definition 7 (Acceleratable cycle). *Let* $(L, l^0, X, \mathrm{ig}_0, \{\mathrm{inv}_l\}_{l \in L}, E)$ *be an extended timed automaton. A sequence* $(e_0, \ldots, e_{n-1})$ *of edges* $e_i = (l_i, \mathrm{cc}_i, \mathrm{ig}_i, \mathrm{ia}_i, Z_i, l_i')$ $\in E$ *is called a* cycle *if* $l_i' = l_{i+1 \bmod n}$ *for all* $0 \leq i < n$. *It is called a* simple cycle*, if additionally* $l_i \neq l_j$ *for all* $i \neq j$. *Following [16], a simple cycle is called* acceleratable*, if all invariants and guards on the cycle contain at most a single clock variable, which is the same for all invariants and guards on the*

[3] For readability, we omit the additional terms ensuring maximality of right-hand sides [13].

cycle, and this clock, say x_m, is reset on all incoming edges to l_0. The clock x_m is called the clock of the cycle*, and l_0 is called the* reset location*. By* acceleratable cycle*, we mean an acceleratable simple cycle. By an* integer cycle*, we mean a cycle where at least one edge contains an assignment to integer variables.*

In [16] it is shown that for any acceleratable simple cycle, there exists an interval $[a, b]$ of clock values, such that $[a, b]$ contains exactly all the possible execution times of the cycle, independently of any path prefix. It follows that any $k \geq 1$ consecutive executions of the cycle take time in $[ka, kb]$.

Let us see what happens during saturation when a cycle $(e_0, \ldots, e_{n-1})$ is reached. We denote by C_t^i the time-reachability clause for location l_i, and by C_d^i the discrete-step clause corresponding to edge e_i. Let C_0 be a reachability conjecture referring to location l_0. The clause C_0 can be resolved with C_d^{n-1} to yield a clause C_1 referring to location l_{n-1}, which in turn can be resolved with C_t^{n-1}. After $2n$ resolution steps, we obtain a clause C_{2n} which again refers to location l_0, as shown in Figure 2. Since clauses C_0 and C_{2n} have the same free part $\mathrm{R}(\mathbf{x}, \mathbf{z}, l_0) \rightarrow$, the induction rule may be applied, under the condition that replaying the derivation (as explained in Section 3) yields a constraint Λ_Δ of the required form.

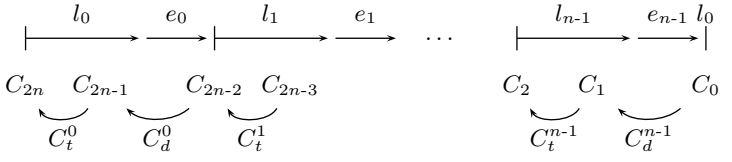


Fig. 2. Backward traversal of a cycle $(e_0, \ldots, e_{n-1})$

The parameterized version of C_0 has the form $\mathbf{x} = \mathbf{q}, \mathbf{z} = \mathbf{p} \parallel \mathrm{Reach}(\mathbf{x}, \mathbf{z}, l_0) \rightarrow$ where $\mathbf{p}, \mathbf{q}$ are vectors of fresh parameters, one for each x_i and z_i, respectively. This clause is successively resolved with the clauses $C_d^{n-1}, C_t^{n-1}, \ldots, C_d^0$, yielding $\Lambda_\Delta[\mathbf{p}, \mathbf{q}, \mathbf{x}, \mathbf{z}] \parallel \mathrm{Reach}(\mathbf{x}, \mathbf{z}, l_0) \rightarrow$. Since there are no atomic constraints containing both variables from X_C and from X_D, the constraint Λ_Δ is of the form $\Lambda_x[\mathbf{p}, \mathbf{x}], \Lambda_z[\mathbf{q}, \mathbf{z}]$ and hence represents a product of two independent relations. After renaming we obtain two constraints $\Lambda_x[\mathbf{x}, \mathbf{x}']$ (referring only to clock variables) and $\Lambda_z[\mathbf{z}, \mathbf{z}']$ (referring only to integer variables) which can be shown to be of the forms required by Proposition 4 and Proposition 5, respectively, by induction over the derivation.

Theorem 8. *Let $\mathcal{T}$ be an extended timed automaton such that any integer cycle is acceleratable, and any location belongs to at most one integer cycle. Let N be a clause set containing the encoding of $\mathcal{T}$ and a reachability conjecture. Then N can be finitely saturated by SUP(LA) with constraint induction.*

Proof: Consider a fair derivation $N = N_0, N_1, N_2, \ldots$ from N where $N_{i+1} = N_i \cup \{C_i\}$ and C_i is the non-redundant result of an inference from clauses from N_i,

and no clause in N_i subsumes C_i. Assume for contradiction that the derivation is infinite. Since there are only finitely many locations, there must be infinitely many clauses in the derivation referring to the same location, say l, (those are clauses of the form $\Lambda \parallel \mathrm{Reach}(\mathbf{x}, \mathbf{z}, l) \rightarrow$) and hence l must lie on a cycle. If no path from l back to itself involves any integer operations, then l can only repeat finitely often ([13], Theorem 4.6). Hence l must lie on an integer cycle, which by assumption is unique and acceleratable, and at least one of its locations is a reset location, say l_r. Furthermore, l_r must also repeat infinitely often, hence there is an infinite sequence $C_{i_1}, C_{i_2}, \ldots$ of clauses referring to l_r. Since the derivation is fair, we eventually apply the constraint induction rule to two successive such clauses, say C_{i_j} and $C_{i_{j+1}}$. Assume the rule is a applied at step j of the derivation i.e., the resulting invariant clause is C_j. Writing $\Lambda_{i_j}, \Lambda_{i_{j+1}}$ for the constraints of clause $C_{i_j}, C_{i_{j+1}}$, respectively, the invariant clause has the form $\Lambda_{i_j}[\mathbf{x}], \Lambda^k_\Delta[\mathbf{x}, \mathbf{x}', k] \parallel \mathrm{Reach}(\mathbf{x}', l_r) \rightarrow$. This clause cannot eliminated by forward subsumption, for otherwise there would have to be a clause $\Lambda' \parallel \mathrm{Reach}(\mathbf{x}, l_r) \rightarrow$ in N_j such that

$$\forall \mathbf{x}, \mathbf{x}', k. \left(\Lambda_{i_j}[\mathbf{x}], \Lambda^k_\Delta[\mathbf{x}, \mathbf{x}', k] \rightarrow \exists \mathbf{y}. \Lambda'[\mathbf{x}', \mathbf{y}]\right)$$

would have to hold, where $\mathbf{y}$ are the variables of Λ' different from $\mathbf{x}, \mathbf{x}', k$. But then the last premise of the constraint induction rule would also be subsumed, because $\Lambda_{i_{j+1}}$ is equivalent to $\left(\Lambda_{i_j}[\mathbf{x}], \Lambda^k_\Delta[\mathbf{x}, \mathbf{x}', k]\right)\{k \mapsto 1\}$, and so the rule could not have been applied in the first place. It follows that the invariant clause is contained in N_{j+1} and all subsequent clause sets, since backward subsumption has to be strict. The invariant clause can be resolved with the clauses corresponding to the edges in the cycle, yielding clauses of the form $\Lambda[\mathbf{x}, \mathbf{x}', k] \parallel \mathrm{Reach}(\mathbf{x}', l) \rightarrow$ for every location l on the cycle. Any further traversal of the cycle then yields clauses of the form $\Lambda[\mathbf{x}, \mathbf{x}', k+1] \parallel \mathrm{Reach}(\mathbf{x}', l) \rightarrow$, which are subsumed, as

$$\forall \mathbf{x}, \mathbf{x}', k. \left(\Lambda[\mathbf{x}, \mathbf{x}', k+1] \rightarrow \exists k'. \Lambda[\mathbf{x}, \mathbf{x}', k']\right)$$

holds. Finally, all clauses $C_{i_{j+m}}$, $m > 0$, are instances of C_j (via instantiation of k), and hence eliminated by forward subsumption, so the sequence $C_{i_1}, C_{i_2}, \ldots$ cannot be infinite, a contradiction.

Since the encoding of extended timed automata does not introduce any function symbols ranging into the arithmetic sorts, it is sufficiently complete, and SUP(LA) is therefore refutationally complete for such encodings. Together with Theorem 8, this implies that SUP(LA) is a decision procedure for the reachability problem in extended timed automata.

5 Implementation and Results

We have implemented the constraint induction rule in our SPASS(LA) theorem prover [1]. SPASS(LA) currently uses Z3 [9] as a back end for constraint solving, both for satisfiability and implication checking. Although Z3 supports

mixed real/integer constraints, it turned out that when checking implication between two constraints both containing integer variables (as they arise in our approach), Z3 almost always returned "unknown". Since the implication check is needed for subsumption and hence is ultimately the key to termination, we decided to implement our own implication test for mixed constraints. The test consists of a preprocessing step, which tries to eliminate all conjuncts containing integer variables from the right-hand side of the implication, followed by a call to Z3 with the resulting implication problem. The preprocessing works as follows: suppose we are trying to prove the implication $\forall \mathbf{x}.\Lambda_2 \Rightarrow \exists \mathbf{y}.\Lambda_1$, where Λ_1, Λ_2 are constraints, $\mathbf{x}$ are the variables of Λ_2 and $\mathbf{y}$ are the variables of Λ_1 not occurring in Λ_2. Suppose there are atomic constraints $\phi_1 \in \Lambda_1$, $\phi_2 \in \Lambda_2$ such that $\phi_1 = x - \sum_{i=1}^{n} \alpha_i k_i \ \# \ c$ and $\phi_2 = x - \sum_{J \subseteq \{1,\dots,n\}} \sum_{j \in J} \alpha_j k'_j \ \# \ c + d$, where $\#$ is one of $<, \leq, =, \geq$ or $>$, x is a real (or integer) variable, k_i, k'_j are integer variables and $c, d \in \mathbb{R}$. If $d = \sum_{L \subseteq \{1,\dots,n\}} \sum_{l \in L} m_l \alpha_{i_l}$ (where m_l are integer constants ≥ 1) such that L contains at least the indices missing from J, i.e., $(\{1, \dots, n\} \setminus J) \subseteq L$, then ϕ_2 implies $\exists (k'_j)_{j \in J}.\phi_1$: assign m_l to k'_l, and either k_j or $k_j + m_j$ to the other k'_j. In this case, we can remove ϕ_1 from Λ_1. In the implementation, we currently only consider the case where $L = \{i\}$ for some $i \in \{1, \dots, n\}$, and either $J = \{1, \dots, n\}$ or $J = \{1, \dots, n\} \setminus L$, which is enough to handle all implication problems arising in our examples. Nevertheless, we are investigating the use of other solvers that implement complete quantifier elimination for mixed constraints.

Example 9 (Extended timed automaton). Consider the extended timed automaton in Figure 3, where x_1, x_2 are clocks and z_1, z_2 are integer variables. We want to check whether location L_2 is reachable with a valuation such that $z_1 \geq z_2$ and $x_2 < 12$. Since x_2 is never reset to zero, its value represents the total time elapsed since first entering L_1. As the cycle at L_1 must be traversed four times before z_1 has overtaken z_2, and each cycle traversal takes at least three time units, such a state is not reachable.

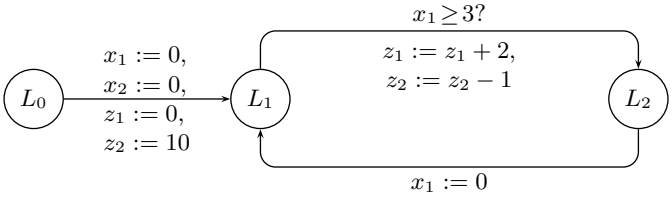


Fig. 3. An extended timed automaton

This problem can be encoded by the following clause set, where the last clause is the negated conjecture:[4]

[4] For simplicity, we use $L_i(\dots)$ instead of $\mathrm{Reach}(\dots, L_i)$, and we also omit L_0.

$$
\begin{array}{rcl}
x_1 = 0,\ x_2 = 0,\ z_1 = 0,\ z_2 = 0 & \| & \rightarrow L_1(x_1, x_2, z_1, z_2) \\
t \geq 0,\ x_1' = x_1 + t,\ x_2' = x_2 + t & \| & L_1(x_1, x_2, z_1, z_2) \rightarrow L_1(x_1', x_2', z_1, z_2) \\
z_1' = z_1' + 2,\ z_2' = z_2' - 1 & \| & L_1(x_1, x_2, z_1, z_2) \rightarrow L_2(x_1, x_2, z_1', z_2') \\
t \geq 0,\ x_1' = x_1 + t,\ x_2' = x_2 + t & \| & L_2(x_1, x_2, z_1, z_2) \rightarrow L_2(x_1', x_2', z_1, z_2) \\
x_1' = 0 & \| & L_2(x_1, x_2, z_1, z_2) \rightarrow L_1(x_1', x_2, z_1, z_2) \\
z_1 \geq z_2,\ x_2 < 12 & \| & L_2(x_1, x_2, z_1, z_2) \rightarrow
\end{array}
$$

The clause set is satisfiable, and without the constraint induction rule, SPASS(LA) does not terminate. With constraint induction activated, the invariant clause

$$k \geq 1,\ x_1 = 0,\ x_2 \geq 3k,\ z_1 = 2k,\ z_2 = 10 - k \ \| \ \rightarrow L_1(x_1, x_2, z_1, z_2)$$

is derived as soon as the cycle has been traversed once, and is used to subsume all other L_1-clauses. SPASS(LA) terminates with the answer "completion found"[5] after deriving 23 clauses.

The next example shows that the induction rule is also useful for speeding up proof search and finding shorter proofs in the case of unsatisfiable clause sets.

Example 10 (Water tank controller). Figure 4 depicts a water tank controller [1] monitoring the water level x in a water tank, into which water is flowing with a constant rate c_{in}. Whenever the water level is greater than 200, the controller opens a valve through which water leaves the tank at a constant rate of c_{out}.

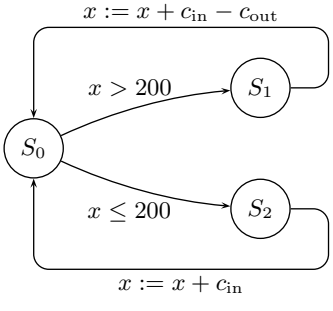


Fig. 4. Water tank controller

We may want to prove that, starting from an empty tank, the water level can reach $200 + c_{\text{in}}$. This problem can be encoded by the following clause set:

$$
\begin{array}{rcl}
x > 200 & \| & S_0(x) \rightarrow S_1(x) \\
x \leq 200 & \| & S_0(x) \rightarrow S_2(x) \\
x' = x + c_{\text{in}} - c_{\text{out}} & \| & S_1(x) \rightarrow S_0(x') \\
x' = x + c_{\text{in}} & \| & S_2(x) \rightarrow S_0(x') \\
x = 0 & \| & \rightarrow S_0(x) \\
x \geq 201 & \| & S_0(x) \rightarrow
\end{array}
$$

[5] A completion is a satisfiable saturation of the initial clause set.

For $c_{\text{in}} = 1$ and $c_{\text{out}} = 2$[6], SPASS(LA) without constraint induction needs to derive 1212 clauses before finding a proof of length 211. The proof consists of repeated traversals of the $S_0 \to S_1 \to S_0$ cycle with increasing values of x, until $x = 201$ is reached.

With constraint induction activated, as soon as the clause $x = 1 \| \to S_0(x)$ has been derived from the initial clause $x = 0 \| \to S_0(x)$ (using the second and fourth clause) SPASS(LA) detects the cycle and derives the invariant clause

$$1 \leq k \leq 201,\ x = k \ \| \ \to S_0(x).$$

which is resolved with the negated conjecture, yielding the empty clause. The proof has length 9 and SPASS(LA) finds it after deriving 13 clauses in total.

If we replace the last clause with $x > 201 \ \| \ S_0(x) \to$, the clause set becomes satisfiable. Without constraint induction, SPASS(LA) now derives 1214 clauses before answering "completion found", whereas with constraint induction, only 23 clauses need to be derived (among them the above invariant clause).

Table 1 shows the results from the above examples, together with the total time spent on the problem.

Table 1. Summary of experimental results

		SUP(LA)		SUP(LA)+ind	
Problem		clauses derived	time	clauses derived	time
Extended TA	sat	–	–	23	0.25s
Water tank	unsat	1212	33s	13	0.15s
Water tank	sat	1214	33s	23	0.18s

6 Conclusion

We have presented the constraint induction rule that automatically generates inductive invariants during proof search in the context of superposition modulo linear arithmetic. The rule applies to loops in which repeated applications of the same sequence of inferences yield clauses which differ only in their arithmetic constraints (their free parts being identical up to renaming of universally quantified variables). The derived invariant summarizes these clauses by representing the transitive closure of the transformation relating the clauses in the loop. The loop can thus be avoided, by using the invariant clause to subsumes its instances, provided that the invariant clause is smaller in the clause ordering (which is required to maintain completeness of the calculus). In order to find a well-founded ordering for which this is the case, one has to ensure that the constraint induction rule is only applied a finite number of times.

[6] In principle, c_{in} and c_{out} don't need to be instantiated, since the invariant computation does not care about the values of constants, but our implementation does not yet handle constant symbols in constraints.

As evidenced by our implementation, the constraint induction rule can considerably speed up proof search, enabling termination of saturation in cases where it would otherwise diverge, and allowing shorter proofs to be found. Since the induction rule applies to clauses with the same free part and invariants thus only talk about the arithmetic constraints, their computation does not require proof generalization and schematization techniques that are necessary to compute invariants for the full first-order setting [20]. Nevertheless, the induction rule significantly increases the power of the SUP(LA) calculus, making it possible to turn it into a decision procedure for reachability in timed automata extended with unbounded integer variables. The decidability of the reachability problem for extended timed automata is not a new result in itself, as it can be obtained from results on counter automata [8,7]. However, we are able to obtain the result using a general-purpose approach like superposition (which applies to full first-order logic), extended with an induction rule that is also applicable outside the specific automata setting.

Preliminary testing of our implementation shows that the rule enables termination of saturation and the finding of short proofs for practically interesting problems. We are currently evaluating the use of the rule for problems from program and protocol verification (particularly in the setting of first-order probabilistic timed automata [11]) and ontology reasoning. Finally, we are working on extending the rule to handle wider classes of constraints.

References

1. Althaus, E., Kruglov, E., Weidenbach, C.: Superposition Modulo Linear Arithmetic SUP(LA). In: Ghilardi, S., Sebastiani, R. (eds.) FroCoS 2009. LNCS, vol. 5749, pp. 84–99. Springer, Heidelberg (2009)
2. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. 126(2), 183–235 (1994)
3. Bachmair, L., Ganzinger, H., Waldmann, U.: Superposition with Simplification as a Decision Procedure for the Monadic Class with Equality. In: Mundici, D., Gottlob, G., Leitsch, A. (eds.) KGC 1993. LNCS, vol. 713, pp. 83–96. Springer, Heidelberg (1993)
4. Bachmair, L., Ganzinger, H., Waldmann, U.: Refutational theorem proving for hierarchic first-order theories. Applicable Algebra in Engineering, Communication and Computing, AAECC 5(3/4), 193–212 (1994)
5. Boigelot, B., Wolper, P.: Symbolic Verification with Periodic Sets. In: Dill, D.L. (ed.) CAV 1994. LNCS, vol. 818, pp. 55–67. Springer, Heidelberg (1994)
6. Bozga, M., Iosif, R., Konečný, F.: Fast Acceleration of Ultimately Periodic Relations. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 227–242. Springer, Heidelberg (2010)
7. Bozga, M., Iosif, R., Lakhnech, Y.: Flat parametric counter automata. Fundam. Inform. 91(2), 275–303 (2009)
8. Comon, H., Jurski, Y.: Multiple Counters Automata, Safety Analysis and Presburger Arithmetic. In: Vardi, M.Y. (ed.) CAV 1998. LNCS, vol. 1427, pp. 268–279. Springer, Heidelberg (1998)

9. 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)
10. Fermüller, C.G., Leitsch, A., Hustadt, U., Tamet, T.: Resolution decision procedures. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, vol. II, ch.25, pp. 1791–1849. Elsevier (2001)
11. Fietzke, A., Hermanns, H., Weidenbach, C.: Superposition-Based Analysis of First-Order Probabilistic Timed Automata. In: Fermüller, C.G., Voronkov, A. (eds.) LPAR-17. LNCS, vol. 6397, pp. 302–316. Springer, Heidelberg (2010)
12. Fietzke, A., Kruglov, E., Weidenbach, C.: Automatic generation of inductive invariants by SUP(LA). Technical Report MPI-I-2012-RG1-002, Max-Planck-Institut für Informatik (2012)
13. Fietzke, A., Weidenbach, C.: Superposition as a decision procedure for timed automata. In: MACIS, pp. 52–62 (2011)
14. Finkel, A., Leroux, J.: How to Compose Presburger-Accelerations: Applications to Broadcast Protocols. In: Agrawal, M., Seth, A.K. (eds.) FSTTCS 2002. LNCS, vol. 2556, pp. 145–156. Springer, Heidelberg (2002)
15. Halpern, J.Y.: Presburger arithmetic with unary predicates is Π_1^1 complete. Journal of Symbolic Logic 56(2), 637–642 (1991)
16. Hendriks, M., Larsen, K.G.: Exact acceleration of real-time model checking. Electr. Notes Theor. Comput. Sci. 65(6) (2002)
17. Jacquemard, F., Rusinowitch, M., Vigneron, L.: Tree Automata with Equality Constraints Modulo Equational Theories. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 557–571. Springer, Heidelberg (2006)
18. Korovin, K., Voronkov, A.: Integrating Linear Arithmetic into Superposition Calculus. In: Duparc, J., Henzinger, T.A. (eds.) CSL 2007. LNCS, vol. 4646, pp. 223–237. Springer, Heidelberg (2007)
19. Kruglov, E., Weidenbach, C.: SUP(T) decides the first-order logic fragment over ground theories. In: MACIS, pp. 126–148 (2011)
20. Peltier, N.: A General Method for Using Schematizations in Automated Deduction. In: Goré, R.P., Leitsch, A., Nipkow, T. (eds.) IJCAR 2001. LNCS (LNAI), vol. 2083, pp. 578–592. Springer, Heidelberg (2001)
21. Wolper, P., Boigelot, B.: Verifying Systems with Infinite but Regular State. In: Vardi, M.Y. (ed.) CAV 1998. LNCS, vol. 1427, pp. 88–97. Springer, Heidelberg (1998)

Moral Reasoning under Uncertainty

The Anh Han[1,⋆], Ari Saptawijaya[1,2,⋆⋆], and Luís Moniz Pereira[1]

[1] Centro de Inteligência Artificial (CENTRIA)
Departamento de Informática, Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa, 2829-516 Caparica, Portugal
[2] Fakultas Ilmu Komputer, Universitas Indonesia, Kampus UI Depok 16424, Indonesia
{h.anh,ar.saptawijaya}@campus.fct.unl.pt, lmp@fct.unl.pt

Abstract. We present a Logic Programming framework for moral reasoning under uncertainty. It is enacted by a coherent combination of our two previously implemented systems, Evolution Prospection for decision making, and P-log for probabilistic inference. It allows computing available moral judgments via distinct kinds of prior and post preferences. In introducing various aspects of uncertainty into cases of classical trolley problem moral dilemmas, we show how they may appropriately influence moral judgments, allowing decision makers to opt for different choices, and for these to be externally appraised, even when subject to incomplete evidence, as in courts.

Keywords: Moral Reasoning, Uncertainty Reasoning, Evolution Prospection, Logic Programming, P-log.

1 Introduction

There has been growing interest in understanding morality from a scientific point of view, arising from diverse fields, e.g. primatology [6], cognitive sciences [12,16], neuroscience [26], and other interdisciplinary perspectives [14]. The study of morality has attracted the artificial intelligence community too. Research on modeling moral reasoning computationally have been reported assiduously since the AAAI 2005 Fall Symposium on Machine Ethics [2], and recently in book form [3,27]. We remit to these references and our own previous work [22,23] for detailed background motivation, techniques, and promises of this burgeoning field.

In prior work we exploit features of logic programming, e.g. default negation, abduction and preferences, to model moral reasoning, and employ prospective logic programming with evolution prospection [21,20]. Possible decisions in a moral dilemma are modeled as abducible hypotheses. Abductive solutions (cf. Def. 1 below) are then computationally generated which capture hypothetical decisions and their consequences. The solutions violating integrity constraints, e.g. those containing actions involving intentional killing, are ruled out. Finally, a posteriori preferences single out those generated hypothetical decisions that characterize preferred moral decisions, including the use of utility functions.

⋆ TAH acknowledges the support from FCT-Portugal, grant SFRH/BD/62373/2009.
⋆⋆ AS acknowledges the support from FCT-Portugal, grant SFRH/BD/72795/2010.

N. Bjørner and A. Voronkov (Eds.): LPAR-18, LNCS 7180, pp. 212–227, 2012.

This paper aims to show how evolution prospection of conceivable scenarios can be extended to handle moral judgments under uncertainty, by employing a combination of our XSB-Prolog system with P-log [4,11] for computing scenarios' probabilities and utilities. It extends our previous work [22,23] in now further enabling judgmental reasoning under uncertainty concerning the facts, the effects, and even the actual actions performed. For illustration, the newly introduced extensions effectively show in detail how to declaratively model and computationally deal with uncertainty in prototypical classic moral situations known generically as the trolley problem [7].

The theory's implemented system can thus prospectively consider moral judgments, under hypothetical and uncertain situations, to decide on the most likely appropriate one. The overall moral reasoning is accomplished via a priori constraints and a posteriori preferences on abductive solutions tagged with uncertainty and utility measures, features henceforth made available in prospective logic programming.

The paper is composed by first introducing the trolley problem and its moral dilemmas, in Section 2, followed by a formal description of scenario evolution prospection under uncertainty, in Section 3. Forthwith, illustrative trolley problem examples of decision making under uncertainty are detailed, in Section 4. In Section 5 we adopt instead the external observer's point of view when passing moral judgment on an agent's choices, even if its actions, circumstances, and available evidence are uncertain to a degree. There follows, in Section 6, a discussion of the examples' results in the light of well-known moral principles, and of how these are respected. Some conclusions are then drawn, in Section 7.

This work is neither a proposal for machine incorporated ethics nor an ethics for humans who use machines, as often addressed in the literature.

2 The Trolley Problem and the Principle of Double Effect

The trolley problem presents several moral dilemmas that inquire whether it is permissible to harm one or more individuals for the purpose of saving others. It has the following initial circumstance [12]: "*There is a trolley and its conductor has fainted. The trolley is headed toward five people walking on the track. The banks of the track are so steep that they will not be able to get off the track in time.*" Given this circumstance, there exist several cases of moral dilemmas [16]. The following three are considered here (see Figure 22.2 (1)-(3) of [23] for graphical illustration):

Bystander. Hank is standing next to a switch that can turn the trolley onto a side track, thereby preventing it from killing the five people. However, there is a man standing on the side track. Hank can throw the switch, killing him; or he can refrain from doing so, letting the five die. Is it morally permissible for Hank to throw the switch?

Footbridge. Ian is on the bridge over the trolley track, next to a heavy man, which he can shove onto the track in the path of the trolley to stop it, preventing the killing of five people. Ian can shove the man onto the track, resulting in death; or he can refrain from doing so, letting the five die. Is it morally permissible for Ian to shove the man?

Loop Track. Ned is standing next to a switch that can temporarily turn the trolley onto a side track, without stopping, only to join the main track again. There is a heavy man on the side track. If the trolley hits the man, he will slow down the trolley, giving time

for the five to escape. Ned can throw the switch, killing the man; or he can refrain from doing so, letting the five die. Is it morally permissible for Ned to throw the switch?

The trolley problem suite has been used in tests to assess moral judgments of subjects from demographically diverse populations [12,16]. Interestingly, although all three cases have the same goal, i.e. to save five albeit killing one, subjects come to different judgments on whether the action to reach the goal is permissible or impermissible, i.e. permissible for the Bystander case, but impermissible for the Footbridge and Loop Track cases. As reported by [16], the judgments appear to be widely shared among demographically diverse populations.

Although subjects have difficulty to uncover which moral rules they apply for reasoning in the above cases, their judgments appear to be consistent with the so-called the principle of double effect, enunciated as follows [12]:

Harming another individual is permissible if it is the foreseen consequence of an act that will lead to a greater good; in contrast, it is impermissible to harm someone else as an intended means to a greater good.

The key expression is *intended means*, i.e. performing a (harming) action intentionally to attain greater good. Humans must not deliberately be harmed as means to an end.

3 Evolution Prospection under Uncertainty with P-log

We describe how P-log can be integrated into an evolving prospective agent system. We start by briefly recalling the constructs of the Evolution Prospection (EP) system and P-log, to the extent we use them here.

3.1 Evolution Prospection

The implemented EP system has proven useful for decision making [20,24], under different application domains, including Elder Care and Ambient Intelligence in home environment [10,9,24]. The ease in expressing preferences in EP [19,20] enables to take into account agents' preferences with precision. The EP system is implemented on top of ABDUAL, a preliminary implementation of [1], using XSB Prolog [28].

Language. Let $\mathcal{L}$ be a first order language. A domain literal in $\mathcal{L}$ is a domain atom A or its default negation $not\ A$. The latter is used to express that the atom is false by default (closed world assumption). A domain rule in $\mathcal{L}$ is of the form: $A \leftarrow L_1, \ldots, L_t\ (t \geq 0)$, where A is a domain atom and $L_1, \ldots, L_t$ are domain literals. An integrity constraint (IC) in $\mathcal{L}$ is a rule with an empty head. A program P over $\mathcal{L}$ is a set of domain rules and integrity constraints, standing for all their ground instances.

In this paper, we only consider Normal Logic Programs (NLPs), i.e. the head of a rule is an atom or empty. We focus furthermore on abductive logic programs, i.e. NLPs allowing for abducibles – user-specified positive literals without rules, whose truth-value is not fixed. Abducibles instances or their default negations may appear in bodies of rules, like any other literal. They stand for hypotheses, each of which may independently be assumed true, in positive literal or default negation form, as the case may be, in order to produce an abductive solution to a query:

Definition 1 (Abductive Solution). *An abductive solution is a consistent collection of abducible instances or their negations that, when replaced by true everywhere in P, affords a model of P (for the specific semantics used on P), which satisfies the query and the ICs – a so-called abductive model.*

Active Goals. In each cycle of its evolution the agent has a set of active goals or desires. We introduce the $on\_observe/1$ predicate, which we consider as representing active goals or desires that, once triggered by the observations figuring in its rule bodies, cause the agent to attempt their satisfaction by launching all the queries standing for them, or using preferences to select them. The rule for an active goal AG is of the form: $on\_observe(AG) \leftarrow L_1, ..., L_t \ (t \geq 0)$, where L_1,...,L_t are domain literals. During evolution, an active goal may be triggered by some events, previous commitments or some history-related information.

When starting a cycle, the agent collects its active goals by finding all the $on\_observe(AG)$ that hold under the initial theory without performing any abduction, then finds abductive solutions for their conjunction.

Preferring Abducibles. A declared abducible A can be assumed only if it is a considered one, i.e. if it is expected in the given situation, and, moreover, there is no expectation to the contrary. The *A* in the body indicates it is a collected abducible by a search attempt for a query's abductive solution whenever this rule is used.

$$consider(A) \leftarrow A,\ expect(A),\ not\ expect\_not(A)$$

The rules about expectations are domain-specific knowledge contained in the theory of the program, and effectively constrain the abducible hypotheses available in a situation. To express preference criteria among abducibles, we envisage an extended language $\mathcal{L}^\star$. A preference atom in $\mathcal{L}^\star$ is of the form $a \lhd b$, where a and b are abducibles. It means that if b can be assumed (i.e. considered), then $a \lhd b$ forces a to be assumed too if it may be allowed for consideration. A preference rule in $\mathcal{L}^\star$ is of the form:

$$a \lhd b \leftarrow L_1, ..., L_t$$

where $L_1, ..., L_t \ (t \geq 0)$ are domain literals over $\mathcal{L}^\star$.

A priori preferences are used to produce the most interesting or relevant considered conjectures about possible future states. They are taken into account when generating possible scenarios (abductive solutions), which will subsequently be preferred amongst each other a posteriori, after having been generated, and specified consequences of interest taken into account.

A Posteriori Preferences. Having computed possible scenarios, represented by abductive solutions, more favorable scenarios can be preferred a posteriori. Typically, *a posteriori* preferences are performed by evaluating consequences of abducibles in abductive solutions. An *a posteriori* preference has the form:

$$A_i \ll A_j \leftarrow holds\_given(L_i, A_i),\ holds\_given(L_j, A_j)$$

where A_i, A_j are abductive solutions and L_i, L_j are domain literals. This means that A_i is preferred to A_j a posteriori if L_i and L_j are true as the side-effects of abductive solutions A_i and A_j, respectively, without any further abduction being permitted when just testing for the side-effects. Optionally, in the body of the preference rule there can be any Prolog predicate used to quantitatively compare the consequences of the two abductive solutions.

A posteriori preferences between two abductive solutions is enacted by comparing a pair of consequences of each abductive solution. However, more often than not, one abductive solution might have several relevant consequences that contribute to make it either more or less preferred than the other abductive solution it is being compared to. All those relevant consequences are needed to be taken into account for decision making. This is similar to the problem addressed in standard decision theory [8] which is to decide between two actions by evaluating each action's relevant consequences based on an utility function mapping consequences to utilities, typically assumed to be real valued, and a given decision rule. The only difference is that here we need to evaluate consequences of sets of actions which represent the abductive solutions. The decision rule uses the utility to choose among actions, or abductive solutions in this case. Technically, a decision rule maps one abductive solution to a real value based on the values of its consequences. Then, the one having greater utility is preferred to the one having less.

There have been many decision rules studied in the literature. The best-known one is *expected utility maximization*. In general, as for any type of decision rules, it has a set of relevant consequences and a real-valued utility function mapping those consequences to real numbers are given [8]. This decision rule also requires a probability measure that characterizes the decision maker's uncertainty with respect to the consequences of a hypothetical abductive solution. It orders the abductive solutions according to the expected utility[1] of their consequences given the probability measure. Thus, *a posteriori* preferences using *expected utility maximization* decision rule have the form:

$$A_i \ll A_j \leftarrow expected\_utility(A_i, U_i),\ expected\_utility(A_j, U_j), U_i > U_j$$

where A_i, A_j are abductive solutions. This means that A_i is preferred to A_j a posteriori if the expected utility of relevant consequences of A_i is greater than the expected utility of the ones of A_j.

3.2 P-log

The P-log system in its original form [4] uses ASP as a tool for computing all stable models of the logical part of P-log. Although ASP has proven a useful paradigm for solving a variety of combinatorial problems, its non-relevance property [5] makes the P-log system sometimes computationally redundant. A new implementation of P-log [11], which we deploy in this work, uses the XASP package of XSB Prolog [28] for

[1] Expected utility of a set of consequences C given a probability measure Pr mapping the consequences to probability values, i.e. $Pr : C \rightarrow [0, 1]$, and an utility function U mapping consequences to real-value utilities, i.e. $U : C \rightarrow \mathbb{R}$, is obtained by the formula: $E(C, Pr, U) = \sum_{X \in C} Pr(X)U(X)$.

interfacing with Smodels [18], an answer set solver. The power of ASP allows the representation of both classical and default negation, to produce 2-valued models. Moreover, using XSB as the underlying processing platform enables collecting the relevant abducibles for a query, obtained by need with top-down search. Furthermore, XSB permits to embed arbitrary Prolog code for recursive definitions. Consequently, it allows more expressive queries not supported in the original version, such as meta queries (probabilistic built-in predicates can be used as usual XSB predicates, thus allowing the full power of probabilistic reasoning in XSB) and queries in the form of any XSB predicate expression [11]. In addition, the tabling mechanism of XSB [25] significantly improves the performance of the system.

In general, a P-log program Π consists of a sorted signature, declarations, a regular part, a set of random selection rules, a probabilistic information part, and a set of observations and actions.

Sorted Signature and Declaration. The sorted signature Σ of Π contains a set of constant symbols and term-building function symbols, which are used to form terms in the usual way. Additionally, the signature contains a collection of special function symbols called attributes. Attribute terms are expressions of the form $a(\bar{t})$, where a is an attribute and $\bar{t}$ is a vector of terms of the sorts required by a. A literal is an atomic expression, p, or its explicit negation, neg_p.

The declaration part of a P-log program can be defined as a collection of sorts and sort declarations of attributes. A sort c can be defined by listing all the elements $c = \{x_1, ..., x_n\}$ or by specifying the range of values $c = \{L..U\}$ where L and U are the integer lower bound and upper bound of the sort c. Attribute a with domain $c_1 \times ... \times c_n$ and range c_0 is represented as follows:

$$a : c_1 \times ... \times c_n \texttt{ --> } c_0$$

If attribute a has no domain parameter, we simply write $a : c_0$. The range of attribute a is denoted by $range(a)$.

Regular Part. This part of a P-log program consists of a collection of XSB Prolog rules, facts and integrity constraints (IC) formed using literals of Σ. An IC is encoded as a XSB rule with the `false` literal in the head.

Random Selection Rule. This is a rule for attribute a having the form:

$$random(RandomName, a(\bar{t}), DynamicRange) \texttt{ :- } Body$$

This means that the attribute instance $a(\bar{t})$ is random if the conditions in $Body$ are satisfied. The $DynamicRange$ allows to restrict the default range for random attributes. The $RandomName$ is a syntactic mechanism used to link random attributes to the corresponding probabilities. A constant $full$ can be used in $DynamicRange$ to signal that the dynamic range is equal to $range(a)$.

Probabilistic Information. Information about probabilities of random attribute instances $a(\bar{t})$ taking a particular value y is given by probability atoms (or simply pa-atoms) which have the following form:

$$pa(RandomName, a(\bar{t}, y), d\_(A, B)) \texttt{ :- } Body$$

meaning that if the $Body$ were true, and the value of $a(\bar{t})$ were selected by a rule named $RandomName$, then $Body$ would cause $a(\bar{t}) = y$ with probability $\frac{A}{B}$. Note that the probability of an atom $a(\bar{t}, y)$ will be directly assigned if the corresponding $pa/3$ atom is the head of some *pa-rule* with a true body. To define probabilities of the remaining atoms we assume that, by default, all values of a given attribute which are not assigned a probability are equally likely.

Observations and Actions. These are, respectively, statements of the forms $obs(l)$ and $do(l)$, where l is a literal. Observations $obs(a(\bar{t}, y))$ are used to record the outcomes y of random events $a(\bar{t})$, i.e. random attributes and attributes dependent on them. Statement $do(a(\bar{t}, y))$ indicates $a(\bar{t}) = y$ is enforced as the result of a deliberate action.

In an EP program, P-log code is embedded by putting it between reserved keywords, `beginPlog` and `endPlog`. In P-log, probabilistic information can be obtained using the XSB Prolog built-in predicate $pr/2$ [11]. Its first argument is the query, the probability of which is needed to compute. The second argument captures the result. Thus, probabilistic information can be easily embedded by using $pr/2$ like a usual Prolog predicate, in any constructs of EP programs, including active goals, preferences, and integrity constraints. What is more, since P-log [11] allows to code Prolog probabilistic meta-predicates (Prolog predicates that depend on $pr/2$ predicates), we also can directly use probabilistic meta-information in EP programs. We will illustrate those features with several examples below.

4 Moral Reasoning under Uncertainty

We modify the trolley problems to introduce different aspects of uncertainty, and show how that can be modeled in our framework. Undoubtedly, real moral problems might contain several aspects of uncertainty, and decision makers need to take them into account when reasoning. In moral situations the uncertainty of the decision makers about different aspects such as the actual external environment, beliefs and behaviors of other agents involved in the situation, as well as the success in performing different actual or hypothesized actions, are inescapable. We show that the levels of uncertainty of several such combined aspects may affect the moral decision, reflecting that, with different levels of uncertainty with respect to the de facto environment and success of actions involved, the moral decision makers—such as juries—may consider different choices and verdicts. In the following, we introduce uncertainty into the above mentioned trolley problems. Uncertainty is modeled using probability.

4.1 Revised Bystander Case

The first aspect present in every trolley problem where we can introduce uncertainty is that of how probable the five people walking will die when the trolley is let head on to them without outside intervention, or there is intervention though unsuccessful. People can help each other get off the track. Maybe they would not have enough time in order for all to get out and survive. That is, the moral decision makers now need to account for how probable the five people, or only some of them, might die. It is

reasonable to assume that the probability of a person dying depends on whether he gets help from others; and, more elaborately, on how many people help him. The P-log program modeling this scenario is as follows:

```
beginPlog.
1. person = {1..5}.   bool = {t,f}.
2. die : person --> bool.       random(rd(P), die(P), full).
3. helped : person --> bool.    random(rh(P), helped(P), full).
4. pa(rh(P), helped(P,t), d_(3,5)) :- person(P).
5. pa(rd(P), die(P,t), d_(1,1))    :- helped(P,f).
   pa(rd(P), die(P,t), d_(4,10))   :- helped(P,t).
6. die_5(V):-pr(die(1,t)&die(2,t)&die(3,t)&die(4,t)&die(5,t),V).
endPlog.
```

Two sorts $person$ and $bool$ are declared in line 1. There are two random attributes, die and $helped$. Both of them map a person to a boolean value, saying if a person either dies or does not die, and, if a person either gets help or does not get any, respectively (lines 2-3). The pa-rule in line 4 says that a person might get help from someone with probability $3/5$. In line 5, it is said that a person who does not get any help will surely die (first rule) and the one who gets help dies with probability $4/10$ (second rule in line 5). This rule represents the degree of conviction of the decision maker about how probable a person can survive provided that he is helped. Undoubtedly, this degree affects the final decision to be made. The meta-probabilistic predicate $die\_5/1$ in line 6 is used to compute the probability of all five people dying. Note that in P-log, the joint probability of two events A and B is obtained by the query $pr(A\&B, V)$.

We can see this modeling is not elaborate enough. It is reasonable to assume that the more help a person gets, the more the chance he has to succeed in getting off the track on time. For the sake of clearness of representation, we use a simplified version.

Consider now the Bystander Case with this uncertainty aspect being taken into account, i.e. the uncertainty of five people dying when merely watching the trolley head for them. It can be coded as follows:

```
expect(watching).     trolley_straight <- watching.
end(die(5), Pr) <- trolley_straight, prolog(die_5(Pr)).
```

The abducible of throwing the switch and its consequence is modeled as:

```
expect(throwing_switch).     kill(1) <- throwing_switch.
end(save_men,ni_kill(N)) <- kill(N).
```

The *a posteriori* preferences, which model the double effect principle, are provided by:

```
Ai << Aj <- holds_given(end(die(N),Pr),Ai), U is N*Pr,
    holds_given(end(save_men,ni_kill(K)),Aj), U < K.
Ai << Aj <- holds_given(end(save_men,ni_kill(N)),Ai),
    holds_given(end(die(K),Pr), Aj), U is K*Pr, N < U.
```

There are two abductive solutions in this trolley case, either watching or throwing the switch. In the next stage, the *a posteriori* preferences are taken into account. It is easily

seen that the final decision directly depends on the probability of five people dying, namely, whether that probability is greater than $1/5$.

Let PrD denote the probability that a person dies when he gets help, coded in the second pa-rule (line 5) of the above P-log program. If $PrD = 0.4$ (as currently in the P-log code), the probability of five people dying is 0.107. Hence, the final choice is to merely watch. If PrD is changed to 0.6, the probability of five people dying is 0.254. Hence, the final best choice is to throw the switch. That is, in a real world situation where uncertainty is unavoidable, in order to appropriately provide a moral decision, the system needs to take into account the uncertainty level of relevant factors.

4.2 Revised Footbridge Case

Consider now the following revised version of the Footbridge Case.

Example 1 (Revised Footbridge Case). Ian is on the footbridge over the trolley track and a switch there. He is next to a man, which he can shove so that the man falls near the switch and can turn the trolley onto a parallel empty side track, thereby preventing it from killing the five people. However, the man can die because the bridge is high and he can also fall on the side track, thus very probably getting killed by the trolley due to not being able to get off the track, having been injured from the drop. Also, as a side effect, the fallen man's body might stop the trolley, though this not being Ian's actual intention. In addition, if he is not dead, he may take revenge on Ian.

Ian can shove the man from the bridge, possibly resulting in death or in being avenged; or he can refrain from doing so, possibly letting the five die. Is it morally permissible for Ian to shove the man? One may consider the analysis below either as Ian's own decision making deliberation before he acts, or else that of an outside observer's evaluation of Ian's actions after the fact; a jury's, say.

There are several aspects in this scenario where uncertainty might emerge. First, similarly to the *Revised Bystander* case, the five people may help each other to escape. Second, how probably does the shoved man fall near the switch? How probably does the fallen man die because the bridge is high? And if the man falls on the sidetrack, how probably can the trolley be stopped by his body? These can be programmed in P-log as:

```
beginPlog.
1. bool = {t,f}.   fallen_position = {on_track, near_switch}.
2. shove : fallen_position.  random(rs, shove, full).
   pa(rs, shove(near_switch), d_(7,10)).
3. shoved_die : bool.        random(rsd, shoved_die, full).
   pa(rsd, shoved_die(t), d_(1,1)) :- shove(on_track).
   pa(rsd, shoved_die(t), d_(5,10)) :- shove(near_switch).
4. body_stop_trolley : bool.   random(rbs, body_stop_trolley, full).
   pa(rbs, body_stop_trolley(t), d_(4,10)).
endPlog.
```

The sort $fallen\_position$ declared in line 1 represents possible positions the man can fall at: on the track (on_track) or near the switch ($near\_switch$). The random attribute $shove$ declared in line 2 has no domain parameter and gets a value of $fallen\_position$ sort. The fallen position of shoving is biased to $near\_switch$ with probability $7/10$ (pa-rule in line 2). The probability of its range complement, *on_track*, is implicitly taken

by P-log to be the probability complement of $3/10$. The random attribute $shoved\_die$ declared in line 3 encodes how probable the man dies after being shoved, depending on which position he fell at (two pa-rules in line 3). If he fell on the track, he would surely die (first pa-rule); otherwise, if he fell near the switch, he would die with probability 0.5 (second pa-rule). The random attribute $body\_stop\_trolley$ is declared in line 4 to encode the probability of a body successfully stopping the trolley. Based on this P-log modeling, the Revised Footbridge Case can be represented as:

```
1. abds([watching/0, shove_heavy_man/0]).
2. on_observe(decide).
   decide <- watching.   decide <- shove_heavy_man.
   <- watching, shove_heavy_man.
3. expect(watching).  trolley_straight <- watching.
   end(die(5),Pr) <- trolley_straight, prolog(die_5(Pr)).
4. expect(shove_heavy_man).
5. stop_trolley(on_track, Pr) <- shove_heavy_man,
       prolog(pr(body_stop_trolley(t)&shove(on_track), Pr)).
6. not_stop_trolley(on_track, Pr) <- shove_heavy_man,
       prolog(pr(body_stop_trolley(f)&shove(on_track), Pr1)),
       prolog(die_5(V)), prolog(Pr is Pr1*V).
7. redirect_trolley(near_switch, Pr) <- throwing_switch(Pr).
   throwing_switch(Pr) <- shove_heavy_man,
       prolog(pr(shoved_die(f)&shove(near_switch), Pr)).
8. not_redirect_trolley(near_switch, Pr) <- shove_heavy_man,
       prolog(pr(shoved_die(t)'|'shove(near_switch), Pr1)),
       prolog(die_5(V)), prolog(Pr is Pr1*V).
9. revenge(shove, Pr) <- shove_heavy_man,
       prolog(pr(shoved_die(f), PrShovedAlive)),
       prolog(Pr is 0.01*PrShovedAlive).
10.Ai '|<' Aj <- expected_utility(Ai, U1),
                 expected_utility(Aj,U2), U1 > U2.

beginProlog.     % beginning of just Prolog code
11.consequences([stop_trolley(on_track,_),not_stop_trolley(on_track,_),
  redirect_trolley(near_switch,_),not_redirect_trolley(near_switch,_),
  revenge(shove,_),end(die(_),_)]).
12.utility(stop_trolley(on_track,_),-1).
   utility(not_stop_trolley(on_track,_),-6).
   utility(redirect_trolley(near_switch,_),0).
   utility(not_redirect_trolley(near_switch,_),-5).
   utility(revenge(shove,_),-10).   utility(end(die(N),_),-N).
13.prc(C, P) :- arg(2,C,P).
endProlog.       % end of just Prolog code
```

There are two abducibles, $watching$ and $shove\_heavy\_man$, declared in line 1. Both are a priori expected (lines 3 and 4) and have no expectation to the contrary. Furthermore, only one can be chosen for the only active goal $decide$ of the program (IC in line 2). Thus, there are two possible abductive solutions: $[watching,\ not\ shove\_heavy\_man]$ and $[shove\_heavy\_man,\ not\ watching]$.

In the next stage, the *a posteriori* preference in line 10 is taken into account, in order to rule out the abductive solution with smaller expected utility. Let us look at the relevant consequences of each abductive solution. The list of relevant consequences of the program is declared in line 11.

The one comprising the action of merely watching has just one relevant consequence: five people dying, i.e. $end(die(5), \_)$ (line 3). The other, that of shoving the heavy man,

has these possible relevant consequences: the heavy man falls on the track and his body either stops the trolley (line 5) or does not stop it (line 6); the man falls near the switch, does not die and thus, can throw the switch to redirect the trolley (line 7). But if he too may die, he consequently cannot redirect the trolley (line 8); one other possible consequence needed to be taken into account is that if the man is not dead, he might take revenge on Ian afterwards (line 9).

The utility of the relevant consequences are given in line 12. Their occurrence probability distribution is captured in line 13, using reserved predicate $prc/2$, the first argument of which is a consequence being instantiated during the computation of the built-in predicate $expected\_utility/2$ and the second argument the corresponding probability value, encoded as second argument of each relevant consequence (line 3 and lines 5-9).

Now we can see how the final decision given by our system varies depending on the uncertainty levels of the decision maker with respect to the aspects considered above. Let us denote *PrNS, PrDNS*, and *PrRV* the probabilities of shoving the man to fall near the switch, of the shoved man dying given that he fell near the switch, and of Ian being avenged given that the shoved man is alive, respectively. In the current encoding, $PrNS = 7/10$, $PrDNS = 5/10$ (lines 2-3 of the P-log code) and $PrRV = 0.01$.

Table 1 shows the final decision made with respect to different levels of uncertainty aspects, encoded with the above variables. Columns $E(watch)$ and $E(shove)$ record the expected utilities of choices *watching* and *shoving*, respectively. The last column records the final decision – the one having greater utility, i.e. less people dying.

Table 1. Decisions made with different levels of Uncertainty

	PrNS	PrDNS	PrD	PrRV	E(watch)	E(shove)	Final
1	0.7	0.5	0.4	0.01	-0.8404	-0.7567	shove
2	0.7	0.5	0.2	0.01	-0.3888	-0.4334	watch
3	0.7	0.5	0.4	0.2	-0.8404	-1.4217	watch
4	0.9	0.1	0.4	0.2	-0.8404	-1.8045	watch
5	0.9	0.1	0.2	0.01	-0.3888	-0.1879	shove
6	0.9	0.5	0.2	0.01	-0.3888	-1.1624	watch
7	1.0	0	0	0.01	-0.1562	-0.1	shove
8	1.0	0	0	0.02	-0.1562	-0.2	watch
9	1.0	0	1.0	0.02	-5	-0.2	shove
10	1.0	0	1.0	0.2	-5	-2	shove
11	1.0	0	1.0	0.6	-5	-6	watch

The table gives rise to these (reasonable) interpretations: the stronger Ian believes five people can get off the track by helping each other (i.e. the smaller *PrD* is), the more the chance he decides to merely watch the trolley go (experiment 2 vs. 1; 8 vs. 9); the more Ian believes the shoved man dies (thus he cannot throw the switch), the greater the chance he decides to merely watch the trolley go (experiment 6 vs. 5); the more Ian believes that the shoved person, or his acquaintances, will take revenge on him, the more the chance he decides to merely watch the trolley go (experiment 3 vs. 1; 8 vs. 7; 11 vs. 10); even in the worst case of watching ($PrD = 1$) and in best chance of the

trolley being redirected (the shoved man surely falls near the switch, i.e. $PrNS = 1.0$, and does not die, i.e. $PrDNS = 0$), then, if Ian really believes that the shoved person will take revenge (e.g. $PrRV \geq 0.6$), he will just watch (experiment 11 vs. 9 and 10). The latter interpretation means the decision maker's benefit and safety precede other factors.

In short, although the table is not big enough to thoroughly cover all the cases, it manages to show that our approach to modeling morality under uncertainty succeeds in reasonably reflecting that a decision maker, or a jury pronouncing a verdict, comes up with differently weighed moral decisions, depending on the levels of uncertainty with respect to the different aspects and circumstances of the moral problem.

5 Moral Reasoning Concerning Uncertain Actions

Usually moral reasoning is performed upon conceptual knowledge of the actions. But it often happens that one has to pass a moral judgment on a situation without actually observing the situation, i.e. there is no full, certain information about the actions. In this case, it is important to be able to reason about the actions, under uncertainty, that might have occurred, and thence provide judgment adhering to moral rules within some prescribed uncertainty level. Courts, for example, are required to proffer rulings beyond reasonable doubt. There is a vast body of research on proof beyond reasonable doubt within the legal community, e.g. [17]. The following example is not intended to capture the full complexity found in a court. Consider this variant of the Footbridge case.

Example 2. Suppose a board of juries in a court is faced with the case where the action of Ian shoving the man onto the track was not observed. Instead, they are only presented with the fact that the man died on the side-track and Ian was seen on the bridge at the occasion. Is Ian guilty (beyond reasonable doubt), i.e. does he violate the double effect principle, of shoving the man onto the track intentionally?

To answer this question, one should be able to reason about the possible explanations of the observations, on the available evidence. The following code shows a model for this example. Given the active goal *judge* (line 2), two abducibles are available, i.e. *verdict(guilty_beyond_reasonable_doubt)* and *verdict(not_guilty)*. Depending on how probable each possible verdict, either *verdict(guilty_beyond_reasonable_doubt)* or *verdict(not_guilty)* is expected a priori (line 3 and 9). The sort *intentionality* in line 4 represents the possibilities of an action being performed intentionally (*int*) or non-intentionally (*not_int*). Random attributes *df_run* and *br_slip* in line 5 and 6 denote two kinds of evidence: Ian was definitely running on the bridge in a hurry (*df_run*) and the bridge was slippery at the time (*br_slip*), respectively. Each has prior probability of $4/10$. The probability with which shoving is performed intentionally is captured by the random attribute *shoved* (line 7), which is causally influenced by both evidence. Line 9 defines when the verdicts (*guilty* and *not_guilty*) are considered highly probable using the meta-probabilistic predicate *pr_iShv/1*, shown by line 8. It denotes the probability of intentional shoving, whose value is determined by the existence of evidence that Ian was running in a hurry past the man (signaled by predicate *evd_run/1*) and that the bridge was slippery (signaled by predicate *evd_slip/1*).

```
1. abds([verdict/1]).
2. on_observe(judge).
   judge <- verdict(guilty_beyond_reasonable_doubt).
   judge <- verdict(not_guilty).
3. expect(verdict(X)) <- prolog(highly_probable(X)).
beginPlog.
4. bool = {t, f}.   intentionality = {int, not_int}.
5. df_run : bool.   random(rdr,df_run,full).
   pa(rdr,df_run(t),d_(4, 10)).
6. br_slip : bool.  random(rsb,br_slip,full).
   pa(rsb,br_slip(t),d_(4, 10)).
7. shoved : intentionality.    random(rs, shoved, full).
   pa(rs,shoved(int),d_(97,100)) :- df_run(f),br_slip(f).
   pa(rs,shoved(int),d_(45,100)) :- df_run(f),br_slip(t).
   pa(rs,shoved(int),d_(55,100)) :- df_run(t),br_slip(f).
   pa(rs,shoved(int),d_(5,100))  :- df_run(t),br_slip(t).
:- dynamic evd_run/1, evd_slip/1.
8. pr_iShv(Pr) :- evd_run(X), evd_slip(Y), !,
     pr(shoved(int) '|' obs(df_run(X)) & obs(br_slip(Y)), Pr).
   pr_iShv(Pr) :- evd_run(X), !,
     pr(shoved(int) '|' obs(df_run(X)), Pr).
   pr_iShv(Pr) :- evd_slip(Y), !,
     pr(shoved(int) '|' obs(br_slip(Y)), Pr).
   pr_iShv(Pr) :- pr(shoved(int), Pr).
9. highly_probable(guilty_beyond_reasonable_doubt) :-
                                   pr_iShv(PrG), PrG > 0.95.
   highly_probable(not_guilty) :- pr_iShv(PrG), PrG < 0.6.
endPlog.
```

Using the above model, different judgments can be delivered by our system, subject to available evidence and attending truth value. We exemplify some cases in the sequel. If both evidence are available, where it is known that Ian was running in a hurry on the slippery bridge, then he may have bumped the man accidentally, shoving him unintentionally onto the track. This case is captured by the first *pr_iShv* rule (line 8): the probability of intentional shoving is 0.05. Thus, the atom *highly_probable(not_guilty)* holds (line 10). Hence, *verdict(not_guilty)* is the preferred final abductive solution (line 3). The same abductive solution is obtained if it is observed that the bridge was slippery, but whether Ian was running in a hurry was not observable. The probability of intentional shoving, captured by *pr_iShv*, is 0.29.

On the other hand, if the evidence shows that Ian was not running in a hurry and the bridge was also not slippery, then they do not support the explanation that the man was shoved unintentionally, e.g., by accidental bumping. The action of shoving is more likely to have been performed intentionally. Using the model, the probability of 0.97 is returned and, being greater than 0.95, *verdict(guilty_beyond_reasonable_doubt)* becomes the sole abductive solution. In another case, if it is only known the bridge was not slippery and no other evidence is available, then the probability of intentional shoving becomes 0.79, and, by lines 4 and 10, no abductive solution is preferred. This translates into the need for more evidence as the available one is not enough to issue judgment.

6 Discussion

We discuss other aspects of the trolley problems when uncertainty may occur. In the Loop Track Case, we can consider Ned's uncertainty about how probable the man

standing on the side track with his back turned can realize that the trolley is going toward him, and get out of the track on time. In this case, one question is whether the action of throwing the switch, which possibly kills the man on the side track, is considered as intentional killing like in the original version. We argue that it should, because Ned's intention is to use the man as a means to stop the trolley, even if he is not sure his intention is achievable. If he threw the switch, he must hope that there would be some chance for the trolley to be slowed down by the man's body. Otherwise he would never do it.

Related to the question of permissibility of actions, let us come back to the Revised Footbridge Case. In the original version, the action of shoving a heavy man from the bridge to stop the trolley in order to save five people is impermissible according two both the double and triple effect principles [13,23] since the action is performed in order to bring about an evil: Ian hopes the shoved man's body will stop the trolley. The situation is clearer if we suppose that the shoved man has some chance of surviving and leaving the track on time, so that the trolley still goes forward onto the five people. In this situation, Ian would hope the shoved man dead. Thus, we can see clearly that Ian's intention is to kill the man to stop the trolley. In the revised version, although Ian shoved the man as an intended means to stop the trolley, his intention is not to kill the man to stop the trolley, but rather on the hope the man survives to throw the switch to stop the trolley. Thus, the action of shoving the man in this version is permissible by the third effect principle, despite still being impermissible by the double effect one. In the examples analysis we rely on the double effect principle, and the triple effect one too.

7 Conclusions

This work is neither a proposal for machine incorporated ethics nor an ethics for humans who use machines, as often addressed in the literature. In contradistinction, it purports to be a proof of principle that our understanding of ethical behavior, even under uncertainty, can in part be computationally modeled and implemented. To be sure, it can (1) be a starting point for imbuing machines with ethics but, beyond that, (2) provide a testing ground for our understanding and experimentation with ethical theories for decision making and moral judgment, and (3) afford us an initial tool to empower the generation of ethical problems for the teaching and explanation of ethical judgment [15].

References

1. Alferes, J.J., Pereira, L.M., Swift, T.: Abduction in well-founded semantics and generalized stable models via tabled dual programs. Theory and Practice of Logic Programming 4(4), 383–428 (2004)
2. Anderson, M., Anderson, S.L.: The status of machine ethics: a report from the AAAI Symposium. Minds and Machines 17, 1–10 (2007)
3. Anderson, M., Anderson, S.L.: Machine Ethics. Cambridge U. P. (2011)
4. Baral, C., Gelfond, M., Rushton, N.: Probabilistic reasoning with answer sets. Theory and Practice of Logic Programming 9(1), 57–144 (2009)

5. Castro, L., Swift, T., Warren, D.S.: XASP: Answer set programming with XSB and Smodels (2007), http://xsb.sourceforge.net/shadow_site/manual2/node129.html
6. de Waal, F.: Primates and Philosophers, How Morality Evolved. Princeton U. P. (2006)
7. Foot, P.: The problem of abortion and the doctrine of double effect. Oxford Review 5, 5–15 (1967)
8. Halpern, J.Y.: Reasoning about Uncertainty. MIT Press (2005)
9. Han, T.A., Pereira, L.M.: Collective intention recognition and elder care. In: AAAI 2010 Fall Symposium on Proactive Assistant Agents (PAA 2010). AAAI (2010)
10. Han, T.A., Pereira, L.M.: Proactive intention recognition for home ambient intelligence. In: IE Workshop on AI Techniques for Ambient Intelligence, Ambient Intelligence and Smart Environments, vol. 8, pp. 91–100. IOS Press (2010)
11. Han, T.A., Kencana Ramli, C.D.P., Damásio, C.V.: An Implementation of Extended P-Log Using XASP. In: Garcia de la Banda, M., Pontelli, E. (eds.) ICLP 2008. LNCS, vol. 5366, pp. 739–743. Springer, Heidelberg (2008)
12. Hauser, M.D.: Moral Minds, How Nature Designed Our Universal Sense of Right and Wrong. Little Brown (2007)
13. Kamm, F.M.: Intricate Ethics: Rights, Responsibilities, and Permissible Harm. Oxford U. P. (2006)
14. Katz, L.D. (ed.): Evolutionary Origins of Morality, Cross-Disciplinary Perspectives. Imprint Academic (2002)
15. Lopes, G., Pereira, L.M.: Prospective Storytelling Agents. In: Carro, M., Peña, R. (eds.) PADL 2010. LNCS, vol. 5937, pp. 294–296. Springer, Heidelberg (2010), http://centria.di.fct.unl.pt/~lmp/publications/slides/padl10/quick_moral_robot.avi
16. Mikhail, J.: Universal moral grammar: Theory, evidence, and the future. Trends in Cognitive Sciences 11(4), 143–152 (2007)
17. Newman, J.O.: Quantifying the standard of proof beyond a reasonable doubt: a comment on three comments. Law, Probability and Risk 5(3-4), 267–269 (2006)
18. Baral, C., Gelfond, M., Rushton, N.: Probabilistic Reasoning With Answer Sets. In: Lifschitz, V., Niemelä, I. (eds.) LPNMR 2004. LNCS (LNAI), vol. 2923, pp. 21–33. Springer, Heidelberg (2003)
19. Pereira, L.M., Dell'Acqua, P., Pinto, A.M., Lopes, G.: Inspecting and preferring abductive models. In: Handbook on Reasoning-based Intelligent Systems. World Scientific Publishers (2011) (forthcoming), http://centria.fct.unl.pt/~lmp/publications/online-papers/rbis.pdf
20. Pereira, L.M., Han, T.A.: Evolution prospection in decision making. Intelligent Decision Technologies 3(3), 157–171 (2009)
21. Pereira, L.M., Lopes, G.: Prospective Logic Agents. In: Neves, J., Santos, M.F., Machado, J.M. (eds.) EPIA 2007. LNCS (LNAI), vol. 4874, pp. 73–86. Springer, Heidelberg (2007)
22. Pereira, L.M., Saptawijaya, A.: Moral decision making with ACORDA. In: Short Paper LPAR 2007 (2007)
23. Pereira, L.M., Saptawijaya, A.: Modelling morality with prospective logic. In: Machine Ethics, pp. 398–421. Cambridge U. P. (2011)

24. Pereira, L.P., Han, T.A.: Intention recognition with evolution prospection and causal bayesian networks. In: Madureira, A., et al. (eds.) Computational Intelligence for Engineering Systems: Emergent Applications, vol. 46, pp. 1–33. Springer, Heidelberg (2011)
25. Swift, T.: Tabling for non-monotonic programming. Annals of Mathematics and Artificial Intelligence 25(3-4), 210–240 (1999)
26. Tancredi, L.: Hardwired Behavior, What Neuroscience Reveals about Morality. Cambridge U. P. (2005)
27. Wallach, W., Allen, C.: Moral Machines: Teaching Robots Right from Wrong. Oxford U. P. (2009)
28. XSB. The XSB system version 3.2 vol. 2: Libraries, interfaces and packages (March 2009)

Towards Algorithmic Cut-Introduction*

Stefan Hetzl[1], Alexander Leitsch[2], and Daniel Weller[1]

[1] Institut für Diskrete Mathematik und Geometrie, Technische Universität Wien
[2] Institut für Computersprachen, Technische Universität Wien

Abstract. We describe a method for abbreviating an analytic proof in classical first-order logic by the introduction of a lemma. Our algorithm is based on first computing a compressed representation of the terms present in the analytic proof and then a cut-formula that realizes such a compression. This method can be applied to the output of automated theorem provers, which typically produce analytic proofs.

1 Introduction

Computer-generated proofs are typically analytic, i.e. they only contain logical material that also appears in the theorem shown. This is due to the fact that analytic proof systems have a considerably smaller search space which makes proof-search practically feasible. In the case of the sequent-calculus, proof-search procedures work on the cut-free fragment only. But also resolution is essentially analytic as all clauses derive from the formula that is shown.

One interesting property of non-analytic proofs is their considerably smaller length. The exact difference depends on the logic (or theory) under consideration, but it is typically enormous. In (classical and intuitionistic) first-order logic there are proofs with cut of length n whose theorems have only cut-free proofs of length 2_n (where $2_0 = 1$ and $2_{n+1} = 2^{2_n}$). The length of a proof plays an important role in many situations such as human readability, space requirements and time requirements for proof checking (also in applications such as proof carrying code). For most of these situations general-purpose data compression methods cannot be used as the compressed representation is not a proof any more. It is therefore of high practical interest to develop proof-search methods which produce non-analytic and hence potentially much shorter proofs. The difficulty in devising such methods is that it seems impossible to come up with a method for finding useful cut-formulas *during proof search*. In this paper we take a different angle at the problem: we start with a cut-free proof and abbreviate it by computing useful cuts based on a structural analysis of the cut-free proof.

There is another, more theoretical, motivation which derives from the foundations of mathematics: most of the central mathematical notions have developed from the observation that many proofs share common structures and steps of

* This work was supported by a Marie Curie Intra European Fellowship within the 7th European Community Framework Programme and by the projects P-22028-N13 and I-603 N18 of the Austrian Science Fund (FWF).

N. Bjørner and A. Voronkov (Eds.): LPAR-18, LNCS 7180, pp. 228–242, 2012.

reasoning. Encapsulating those leads to a new abstract notion, like that of a group or a vector space. Such a notion then builds the base for a whole new theory whose importance stems from the pervasiveness of its basic notions in mathematics. From a logical point of view this is the introduction of cuts into an existing proof database. While we cannot claim to contribute much to the understanding of such processes by the current technical state of the art, this second motivation is still worthwhile to keep in mind, if only to remind ourselves that we are dealing with a difficult problem here.

Work on cut-introduction can be found at a number of different places in the literature. Closest to our approach are [14] which is an algorithm for the introduction of atomic cuts that is capable of exponential proof compression and the method [2] for propositional logic which is shown to never increase the size of proofs more than polynomially. The work [1] is studying a different approach to cut-introduction which is based on filling a so-called proof skeleton with formulas in order to obtain a proof with cuts. Yet another approach to the compression of first-order proofs by introduction of definitions is [13]. A way to use focusing to avoid proving atomic subgoals twice which results in a proof with atomic cuts can be found in [8].

In this paper we consider classical first-order logic and treat the problem of introducing a cut that contains a single quantifier. While this is a modest class, the present algorithm is to the best of our knowledge the first for the introduction of quantified non-analytic cuts. The class being simple has the further advantage of allowing a clear exposition of the basic principles of the algorithm. After some preparation in Sections 2 and 3 we describe in Section 4 a calculus that allows to compute compressed representation ("decompositions") of the terms present in a cut-free proof. In Section 5 we show how to find a cut-formula that realizes such a decomposition and in Section 6 we discuss how to further improve the choice of the cut-formula. Some of the proofs are left out from this paper; the reader interested in all details is referred to the technical report [6].

2 Proofs and Herbrand-Sequents

A *sequent* is an ordered pair of sets of formulas (Γ, Δ) written as $\Gamma \to \Delta$. We use the sequent calculus $\mathbf{G3c} + \mathrm{Cut}_{\mathrm{cs}}$[1] from [12] and denote it by **LK**. An *instance* of $\forall x_1 \cdots \forall x_n\, A$ or $\exists x_1 \cdots \exists x_n\, A$ (for A quantifier-free) is a formula of the form $A[x_1 \backslash t_1, \ldots, x_n \backslash t_n]$. A *strong quantifier* is a $\forall$ ($\exists$) quantifier with positive (negative) polarity. We distinguish some important subsets of sequents.

Definition 1. *A* prenex sequent *is a sequent containing only prenex formulas. A prenex sequent without strong quantifiers is called a* Σ_1-sequent. *A* Σ_1*-sequent in which every formula has at most one quantifier is called a* simple sequent.

The notion of instance extends in a straightforward way to Σ_1-sequents. In this section, we will primarily work with Σ_1-sequents which does not constitute

[1] $\mathbf{G3c} + \mathrm{Cut}_{\mathrm{cs}}$ has no structural rules and all its rules are invertible.

a substantial restriction as one can transform every sequent into a validity-equivalent Σ_1-sequent by skolemisation and prenexification.

Definition 2. *Let $\Gamma \to \Delta$ be a Σ_1-sequent. Then $\Gamma' \to \Delta'$ is called* Herbrand-sequent of $\Gamma \to \Delta$ *if it is a tautology and consists of instances of $\Gamma \to \Delta$. The* complexity of a Herbrand-sequent *is defined as $|\Gamma' \to \Delta'| = |\Gamma'| + |\Delta'|$, where $|\ |$ denotes cardinality.*

Example 1. Consider the language containing a constant symbol a, a unary function symbol f and a unary predicate symbol P and the sequent

$$Pa, \forall x\,(Px \supset Pfx) \to Pf^m a$$

in this language (we omit parentheses around the argument of a unary symbol). This sequent has a Herbrand-sequent

$$Pa, Pa \supset Pfa, \ldots, Pf^{m-1}a \supset Pf^m a \to Pf^m a$$

of complexity $m+2$. Note that this Herbrand-sequent is of minimal complexity.

The length of a proof π, written as $|\pi|$, is defined as the number of inferences. The following result is shown in [3].

Theorem 1. *Let $s\colon \Gamma \to \Delta$ be a Σ_1-sequent and π a cut-free proof of s. Then there is a Herbrand-sequent $s'\colon \Gamma' \to \Delta'$ of s s.t. $|s'| \leq |\pi|$.*

Note that given an Herbrand-sequent s' of s one can find a cut-free proof of s from s' by quantifier introductions. Combining this with a propositional proof of s' one obtains a cut-free proof of s. Assuming that π is cut-free is essential for the above theorem to hold. The well-known non-elementary growth of cut-elimination [9,11,10] shows that it cannot be true if π contains cuts. We generalize the concept of Herbrand-sequent to extended Herbrand-sequents which correspond to proofs with cuts (similarly to [4]): define a $\forall$-*cut* to be a cut with cut-formula $\forall x\ A$, where A is quantifier-free. There are efficient algorithms for extracting Herbrand-sequents from cut-free proofs, see e.g. [7].

Definition 3. *Let $\Gamma \to \Delta$ be a Σ_1-sequent, let A be a quantifier-free formula, α be a variable not appearing in $\Gamma \cup \Delta \cup \{A\}$ and $s_1, \ldots, s_k$ be terms. A sequent of the form*

$$A[x\backslash\alpha] \supset \bigwedge_{j=1}^{k} A[x\backslash s_j], \Gamma' \to \Delta'$$

is called extended Herbrand-sequent *if it is a tautology and $\Gamma' \to \Delta'$ consists of instances of $\Gamma \to \Delta$. The* complexity *of the above extended Herbrand-sequent s is defined as $|s| = k + |\Gamma'| + |\Delta'|$.*

Proposition 1. *If π is a proof of a Σ_1-sequent $\Gamma \to \Delta$ and the only cut of π is a $\forall$-cut, then there is an extended Herbrand-sequent s of $\Gamma \to \Delta$ with $|s| \leq |\pi|$.*

Proof. W.l.o.g. the strong universal quantifier in the cut is introduced from a single eigenvariable α. Obtain a propositional proof π' of an extended Herbrand-sequent by replacing the introductions of the weak universal quantifier by $\wedge_{\mathrm{l}}$-inferences and omitting all inferences that introduce quantifiers into the end-sequent.

Example 2. Consider the sequent $Pa, \forall x\,(Px \supset Pfx) \to Pf^{n^2}a$ for $n \geq 1$. It can be derived by a proof π_n using one $\forall$-cut as follows:

$$\dfrac{\dfrac{\dfrac{}{\forall x\,(Px \supset Pfx) \to P\alpha \supset Pf^n\alpha}\,(\chi_1^n)}{\forall x\,(Px \supset Pfx) \to \forall x\,(Px \supset Pf^n x)}\,\forall_{\mathrm{r}} \quad \dfrac{}{\forall x\,(Px \supset Pf^n x), Pa \to P(f^{n^2}a)}\,(\chi_2^n)}{Pa, \forall x\,(Px \supset Pfx) \to Pf^{n^2}a}\,\mathrm{cut}$$

where χ_1^n uses the instances $\alpha, f\alpha, \ldots, f^n\alpha$ of the successor-axiom to prove the cut formula and χ_2^n uses instances $a, f^n a, \ldots, f^{(n-1)n}a$ of the cut formula to prove the claim. The extended Herbrand-sequent of this proof is

$$C, Pa, P\alpha \supset Pf\alpha, \ldots, Pf^{n-1}\alpha \supset Pf^n\alpha \to Pf^{n^2}a$$

where

$$C \;=\; (P\alpha \supset Pf^n\alpha) \supset \bigwedge_{j=0}^{n-1} (Pf^{jn}a \supset Pf^{(j+1)n}a)$$

so it has complexity $2(n+1)$.

We have seen in example 1 that the complexity of the minimal Herbrand-sequent is n^2+2. So by introducing the cut above we get a quadratic compression. For the case of a single universal quantifier this bound is sharp. As in the cut-free case, one can construct a proof with cut from an extended Herbrand-sequent.

Lemma 1. *If π is a proof of $A \supset B, \Gamma \to \Delta$, then there are proofs π_1 of $B, \Gamma \to \Delta$ and π_2 of $\Gamma \to \Delta, A$ with $|\pi_1| \leq |\pi|$ and $|\pi_2| \leq |\pi|$.*

Proof. For obtaining π_1, replace all ancestors of $A \supset B$ by B and the introducing inferences

$$\dfrac{\Pi \to \Lambda, A \quad B, \Pi \to \Lambda}{A \supset B, \Pi \to \Lambda}\,\supset_{\mathrm{l}} \qquad \text{by} \qquad B, \Pi \to \Lambda\ .$$

For π_2 proceed analogously.

Proposition 2. *Let $\Gamma \to \Delta$ be a Σ_1-sequent, let $s = A[x\backslash\alpha] \supset \bigwedge_{j=1}^{k} A[x\backslash s_j]$, $\Gamma' \to \Delta'$ be an extended Herbrand-sequent of $\Gamma \to \Delta$, ψ be a proof of s and l be the maximal length of a quantifier prefix in $\Gamma \to \Delta$. Then there is a proof π of $\Gamma \to \Delta$ having exactly one $\forall$-cut, s as extended Herbrand-sequent and satisfies $|\pi| = \mathrm{O}(|\psi| + |\Gamma \to \Delta| \cdot l)$.*

Proof. By introducing weak quantifiers to derive $\Gamma \to \Delta$ from $\Gamma' \to \Delta'$ and by replacing $\bigwedge_{j=1}^{k}$ by universal quantifiers, obtain a proof π_1 of $A[x\backslash\alpha] \supset \forall x\, A, \Gamma \to \Delta$ with $|\pi_1| \leq |\psi| + |\Gamma \to \Delta| \cdot l + 1$. By Lemma 1 there are proofs π_2 of $\Gamma \to \Delta, A[x\backslash\alpha]$ and π_3 of $\forall x\, A, \Gamma \to \Delta$ with $|\pi_2| \leq |\pi_1|$ and $|\pi_3| \leq |\pi_1|$. Define π as

$$\dfrac{\dfrac{\overset{(\pi_2)}{\Gamma \to \Delta, A[x\backslash\alpha]}}{\Gamma \to \Delta, \forall x\, A}\,\forall_{\mathrm{r}} \qquad \overset{(\pi_3)}{\forall x\, A, \Gamma \to \Delta}}{\Gamma \to \Delta}\,\mathrm{cut}\,,$$

and observe $|\pi| = |\pi_2| + |\pi_3| + 2 \leq 2(|\psi| + |\Gamma \to \Delta| \cdot l + 2)$.

3 Cut-Elimination and Cut-Introduction

Let $\mathcal{S}\colon \{\sigma_1, \ldots, \sigma_n\}$ be a set of substitutions; then by $(\Gamma \to \Delta)\mathcal{S}$ we denote the sequent $\Gamma\sigma_1, \ldots, \Gamma\sigma_n \to \Delta\sigma_1, \ldots, \Delta\sigma_n$.

Proposition 3. *Let $A[x\backslash\alpha] \supset \bigwedge_{j=1}^{k} A[x\backslash s_j], \Gamma' \to \Delta'$ be an extended Herbrand sequent of a Σ_1-sequent $\Gamma \to \Delta$, then $(\Gamma' \to \Delta')\{[\alpha\backslash s_j] \mid 1 \leq j \leq k\}$ is a Herbrand-sequent of $\Gamma \to \Delta$.*

Proof. Consider the proof π of $\Gamma \to \Delta$ constructed in the proof of Proposition 2: by reducing the universal quantifier of the cut using any standard method for cut-reduction we obtain a proof π' having only quantifier-free cuts whose Herbrand sequent is $(\Gamma' \to \Delta')\{[\alpha\backslash s_j] \mid 1 \leq j \leq k\}$.

So we have seen in the previous section that the first-order structure of a Herbrand-sequent corresponds to that of a cut-free proof and that of an extended Herbrand-sequent to that of a proof with a single $\forall$-cut. These observations, together with Proposition 3 that describes cut-elimination on these structures motivates the following statement of the

Cut-introduction Problem for a Single $\forall$-cut: Given a simple sequent $\Gamma \to \Delta$ and a Herbrand-sequent $\Gamma' \to \Delta'$ of $\Gamma \to \Delta$, find an extended Herbrand-sequent $s = A[x\backslash\alpha] \supset \bigwedge_{j=1}^{k} A[x\backslash s_j], \Gamma'' \to \Delta''$ of $\Gamma \to \Delta$ s.t.

$$\Gamma' \to \Delta' \;=\; (\Gamma'' \to \Delta'')\{[\alpha\backslash s_j] \mid 1 \leq j \leq k\}.$$

In order to describe our solution of the above problem, we first give some definitions. For a sequence of terms $\boldsymbol{t} = t_1, \ldots, t_n$ and a formula $F(x)$ (which may or may not contain x), we write $F(\boldsymbol{t})$ for the sequence of formulas $F(t_1), \ldots, F(t_n)$.

For the rest of this section, we fix a simple sequent $s = \forall x\, F_1(x), \ldots, \forall x\, F_n(x) \to \exists x\, F_{n+1}(x), \ldots, \exists x\, F_m(x)$ and a Herbrand-sequent $s' = F_1(\boldsymbol{t_1}), \ldots, F_n(\boldsymbol{t_n}) \to F_{n+1}(\boldsymbol{t_{n+1}}), \ldots, F_m(\boldsymbol{t_m})$ of s where $\boldsymbol{t_i} = t_{i,1}, \ldots, t_{i,n_i}$.

We define the *termset* $T(s, s') = \{t_{i,j} \mid i \leq m, j \leq n_i\}$. The following result shows how the termset can give rise to a solution to the cut-introduction problem.

Proposition 4. *Let* $U = \{u_1, \ldots, u_\ell\}$ *and* $S = \{s_1, \ldots, s_k\}$ *be sets of terms such that* $T(s, s') = \{u_i[\alpha\backslash s_j] \mid i \leq \ell, j \leq k\}$. *Then*

$$s' = (F_1(\boldsymbol{u}), \ldots, F_n(\boldsymbol{u}) \to F_{n+1}(\boldsymbol{u}), \ldots, F_m(\boldsymbol{u}))\{[\alpha\backslash s_j] \mid 1 \leq j \leq k\}$$

Proof. Let s, s' be as above. Since $T(s, s') = \{u_i[\alpha\backslash s_j] \mid i \leq \ell, j \leq k\}$, for every $i \leq m, j \leq n_i$ there exist p, q such that $t_{i,j} = u_p[\alpha\backslash s_q]$. Inversely, for every p, q there are i, j s.t. $t_{i,j} = u_p[\alpha\backslash s_q]$.

The first phase of our approach to cut-introduction for simple sequents consists in determining sets U, S as in Proposition 4. Such sets then induce a schematic extended Herbrand-sequent.

Definition 4. *Let* U, S *be as in Proposition 4. Then the* induced schematic extended Herbrand-sequent *is*

$$X\alpha \supset \bigwedge_{j=1}^{k} Xs_j, F_1(\boldsymbol{u}), \ldots, F_n(\boldsymbol{u}) \to F_{n+1}(\boldsymbol{u}), \ldots, F_m(\boldsymbol{u})$$

where X *is a monadic second-order variable.*

Let s'' be the induced schematic Herbrand-sequent corresponding to s, s', U, S. The second phase is then to determine a substitution $\sigma = [X\backslash\lambda x.\psi]$ s.t. $s''\sigma$ is a tautology. We will show that such a substitution σ always exists.

4 A Calculus of Decompositions

We will now describe our algorithmic solution for the first phase. For this whole section we fix a variable α and a set of ground terms $T = \{t_1, \ldots, t_n\}$.

Definition 5. *A* decomposition *of* T *is a pair of sets of terms, written as* $U \circ S$, *s.t.* $T = \{u[\alpha\backslash s] \mid u \in U, s \in S\}$.

Of course, every T possesses a trivial decomposition by letting $U = \{\alpha\}$ and $S = T$. Keeping our aim of proof compression in mind we are looking for a decomposition $U \circ S$ with $|U| + |S| < |T|$. We will develop a calculus of decompositions along similar lines as a resolution calculus. For a set of terms W and a term v we write $W[x\backslash v]$ for $\{w[x\backslash v] \mid w \in W\}$ and $v[x\backslash W]$ for $\{v[x\backslash w] \mid w \in W\}$ and $\mathrm{V}(v)$ for the set of variables occurring in v.

Definition 6. *We define the following axioms and rules for the manipulation of decompositions. Axioms are of the form*

$$\frac{}{\{\alpha\} \circ \{t\}}\ \mathrm{ax} \quad \textit{if } t \in T.$$

The rules are

$$\frac{U_1 \circ S \quad U_2 \circ S}{(U_1 \cup U_2) \circ S}\ \mathrm{R} \qquad \frac{U \circ S_1 \quad U \circ S_2}{U \circ (S_1 \cup S_2)}\ \mathrm{L} \qquad \frac{U[\alpha\backslash v] \circ S}{U \circ v[\alpha\backslash S]}\ \to \qquad \frac{U \circ v[\alpha\backslash S]}{U[\alpha\backslash v] \circ S}\ \leftarrow$$

for any term v *with* $\alpha \in \mathrm{V}(v)$.

To simplify the notation, we often omit the braces of singleton sets. The above calculus is sound in the sense that it only derives decompositions of subsets of T. More interestingly, it is also complete in the following sense:

Proposition 5. *If T has a decomposition $U \circ S$ then $U \circ S$ is derivable using* $\mathrm{ax}, \leftarrow, \mathrm{R}, \mathrm{L}$.

Proof. Let $T = \{u_i[\alpha\backslash s_j] \mid 1 \leq i \leq m, 1 \leq j \leq k\}$. First, observe that $u_i \circ s_j$ is derivable by a single left-shift. Secondly, we have

$$\frac{u_1 \circ s_j \quad \cdots \quad u_m \circ s_j}{\{u_1, \ldots, u_m\} \circ s_j}\ \mathrm{R}\cdots\mathrm{R} \quad \text{and finally}$$

$$\frac{\{u_1, \ldots, u_m\} \circ s_1 \quad \cdots \quad \{u_1, \ldots, u_m\} \circ s_k}{\{u_1, \ldots, u_m\} \circ \{s_1, \ldots, s_k\}}\ \mathrm{L}\cdots\mathrm{L}\ .$$

A search for decompositions in this calculus is not very efficient due to the indeterministic nature of the $\leftarrow$-inferences. Fortunately, it is possible to work with most general forms of decompositions, thereby getting (almost) rid of the shift-rules. The rest of this section is devoted to the development of such a most general calculus and a search procedure for it.

Definition 7. *A decomposition $U \circ S$ is called* right normal *if $U = U'[\alpha\backslash v]$ implies $v = \alpha$.*

A first but essential result is that right normal forms are unique. To show this we need some auxiliary notions.

Definition 8. *For terms t, v with $\alpha \in \mathrm{V}(v) \cap \mathrm{V}(t)$ we write $t \geq v$ if there is w s.t. $t = w[\alpha\backslash v]$.*

It will be convenient to work with an inductive definition of the set of right shift terms of a term.

Definition 9. *Let $\alpha \in \mathrm{V}(t)$ and define a set* $\mathrm{rsterms}(t)$ *as follows:* $\mathrm{rsterms}(\alpha) = \{\alpha\}$ *and* $\mathrm{rsterms}(f(t_1, \ldots, t_n)) = \{\alpha, f(t_1, \ldots, t_n)\} \cup \bigcap_{i=1, \alpha \in \mathrm{V}(t_i)}^{n} \mathrm{rsterms}(t_i)$.

Example 3. $f(c, g(\alpha)) \geq g(\alpha)$ because $f(c, g(\alpha)) = f(c, \alpha)[\alpha\backslash g(\alpha)]$ but on the other hand $f(\alpha, g(\alpha)) \not\geq g(\alpha)$ because every w with $f(\alpha, g(\alpha)) = w[\alpha\backslash g(\alpha)]$ would have to start with f whose first argument can then no longer be filled. Furthermore $t \geq t$ and $t \geq \alpha$ for all terms t. We have $\mathrm{rsterms}(f(c, g(\alpha))) = \{\alpha, g(\alpha), f(c, g(\alpha))\}$ and $\mathrm{rsterms}(f(\alpha, g(\alpha))) = \{\alpha, f(\alpha, g(\alpha))\}$.

Lemma 2. *Let $\alpha \in \mathrm{V}(t)$. Then $v \in \mathrm{rsterms}(t)$ iff $t \geq v$.*

Lemma 3. *$\geq$ is a partial order of the set of terms containing α.*

Note that $\geq$ is not a total order on the set of terms containing α. For example, consider the terms $f(\alpha), g(\alpha)$, then clearly $\alpha \leq f(\alpha)$ and $\alpha \leq g(\alpha)$ but $f(\alpha)$ and $g(\alpha)$ are incomparable. On the other hand:

Lemma 4. *Let $\alpha \in \mathrm{V}(t)$, then $\geq$ is a total order of* $\mathrm{rsterms}(t)$.

For a non-empty set of terms U we define $\mathrm{rsterms}(U) = \bigcap_{u \in U} \mathrm{rsterms}(u)$. Note that $\geq$ on $\mathrm{rsterms}(U)$ is total as well because it is a substructure of $\geq$ on $\mathrm{rsterms}(u)$ for any $u \in U$.

Proposition 6. *Every decomposition has a unique right normal form.*

Proof. Let $U \circ S$ be a decomposition with two different right normal forms $U_1 \circ S_1$ and $U_2 \circ S_2$. Then there are terms v_1, v_2 s.t. $U = U_1[\alpha \backslash v_1] = U_2[\alpha \backslash v_2]$ and

$$\frac{U_1[\alpha \backslash v_1] \circ S}{U_1 \circ v_1[\alpha \backslash S]} \to \quad \text{and} \quad \frac{U_2[\alpha \backslash v_2] \circ S}{U_2 \circ v_2[\alpha \backslash S]} \to$$

where $S_1 = v_1[\alpha \backslash S]$ and $S_2 = v_2[\alpha \backslash S]$. As $v_1, v_2 \in \mathrm{rsterms}(U)$ we can apply Lemma 4 to obtain w.l.o.g. $v_1 \geq v_2$. As $U_1 \circ S_1 \neq U_2 \circ S_2$ we have $v_1 \neq v_2$ hence $v_1 > v_2$, i.e. there is a $w \neq \alpha$ s.t. $v_1 = w[\alpha \backslash v_2]$. Therefore $U = U_1[\alpha \backslash v_1] = U_1[\alpha \backslash w][\alpha \backslash v_2] = U_2[\alpha \backslash v_2]$ hence $U_2 = U_1[\alpha \backslash w]$ which is not in right normal form.

In light of the above proposition we will henceforth speak about *the* right normal form of a decomposition. Note that the right normal form of a term t can be obtained from using the maximal element of $\mathrm{rsterms}(t)$ as a right shift term.

Lemma 5 (Lifting Lemma for L**).** *If*

$$\frac{U \circ S_1 \quad U \circ S_2}{U \circ (S_1 \cup S_2)}\ \mathrm{L}$$

and $U' \circ S'_1$ is the right normal form of $U \circ S_1$ and $U' \circ S'_2$ is the right normal form of $U \circ S_2$, then

$$\frac{U' \circ S'_1 \quad U' \circ S'_2}{U' \circ (S'_1 \cup S'_2)}\ \mathrm{L}$$

where $U' \circ (S'_1 \cup S'_2)$ is the right normal form of $U \circ (S_1 \cup S_2)$.

Proof. Being in right normal form depends only on the U-part of the decomposition. Therefore, if $U' \circ S'_i$ is in right normal form so is $U' \circ (S'_1 \cup S'_2)$.

Right normality is more problematic when it comes to the R-rule. Consider the two decompositions $\{f_1(\alpha), f_2(\alpha)\} \circ \{f(g(c))\}$ and $\{\alpha\} \circ \{h(g(c))\}$. Both are right normal and they cannot be combined with a R-rule. However shifting both to the left gives $\{f_1(f(\alpha)), f_2(f(\alpha))\} \circ \{g(c)\}$ and $\{h(\alpha)\} \circ \{g(c)\}$ which can be combined with R yielding $\{f_1(f(\alpha)), f_2(f(\alpha)), h(\alpha)\} \circ \{g(c)\}$ which is again right normal. Note that shifting by $f(g(\alpha))$ and $h(g(\alpha))$ instead would give $\{f_1(f(g(\alpha))), f_2(f(g(\alpha)))\} \circ \{c\}$ and $\{h(g(\alpha))\} \circ \{c\}$ whose combination by R would no longer be right normal. So if a R-combination is made possible by applying left shifts before, the minimal such left shifts are most general in the sense that they yield a right normal conclusion of the R-rule. Let us make this precise:

Definition 10. *For right normal decompositions D_1, D_2, D_3, abbreviate*

$$\dfrac{\dfrac{D_1}{D_1'} \leftarrow \quad \dfrac{D_2}{D_2'} \leftarrow}{D_3}\ \mathrm{R} \qquad \textit{by} \qquad \dfrac{D_1 \quad D_2}{D_3}\ \mathrm{R_{mg}}\ .$$

Lemma 6 (Lifting Lemma for R). *If* $\dfrac{U_1 \circ S \quad U_2 \circ S}{(U_1 \cup U_2) \circ S}\ \mathrm{R}$ *and* $U_1' \circ S_1$ *is the right normal form of* $U_1 \circ S$ *and* $U_2' \circ S_2$ *is the right normal form of* $U_2 \circ S$*, then* $\dfrac{U_1' \circ S_1 \quad U_2' \circ S_2}{V \circ T}\ \mathrm{R_{mg}}$ *where* $V \circ T$ *is the right normal form of* $(U_1 \cup U_2) \circ S$*.*

The calculus consisting of $\mathrm{ax}, \mathrm{R_{mg}}, \mathrm{L}$ is sound in the sense that only right normal forms of subsets of T are derived and complete in the following sense:

Theorem 2. *If T has a decomposition $U \circ S$ then the right normal form of $U \circ S$ is derivable using* $\mathrm{ax}, \mathrm{R_{mg}}, \mathrm{L}$.

Proof. By Proposition 5, there exists a derivation of the right normal form of $U \circ S$ using $\mathrm{ax}, \leftarrow, \mathrm{R}, \mathrm{L}$. We convert this derivation inductively to one using only $\mathrm{ax}, \mathrm{R_{mg}}, \mathrm{L}$ of the same structure bringing every line into right normal form by leaving out the $\leftarrow$-inferences and applying Lemmas 5 and 6 for the L- and R-inferences respectively.

We can observe that w.r.t. the generality of a derivation, the calculus $(\mathrm{ax}, \mathrm{R_{mg}}, \mathrm{L})$ behaves like resolution and $(\mathrm{ax}, \leftarrow, \mathrm{R}, \mathrm{L})$ like ground resolution. It is useful to observe the following algorithmic

Corollary 1. *Let A be the axioms induced by T, let B be the* $\mathrm{R_{mg}}$*-closure of A and let C be the* L*-closure of B. Then C contains the right normal forms of all decompositions of T.*

Proof. By inspection of the completeness proof.

Example 4. The sequent $Pa, \forall x\,(Px \supset Pfx) \to Pf^{n^2}a$ has a Herbrand-sequent $Pa, Pa \supset Pfa, \ldots, Pf^{n^2-1}a \supset Pf^{n^2}a \to Pf^{n^2}a$ of size n^2 as in Example 1. For abbreviating it we have to find a decomposition of $T = \{a, fa, \ldots, f^{n^2-1}a\}$. Observe that

$$\dfrac{\alpha \circ f^{in+0}a \quad \cdots \quad \alpha \circ f^{in+n-1}a}{\{\alpha, f\alpha, \ldots, f^{n-1}\alpha\} \circ f^{in}a}\ \mathrm{R_{mg}}, \ldots, \mathrm{R_{mg}}$$

for all $i \in \{0, \ldots, n-1\}$ and that

$$\dfrac{\{\alpha, f\alpha, \ldots, f^{n-1}\alpha\} \circ a \quad \cdots \quad \{\alpha, f\alpha, \ldots, f^{n-1}\alpha\} \circ f^{(n-1)n}a}{\{\alpha, f\alpha, \ldots, f^{n-1}\alpha\} \circ \{a, f^n a, \ldots, f^{(n-1)n}a\}}\ \mathrm{L}, \ldots, \mathrm{L}$$

which shows that this final decomposition is in C. This decomposition induces the schematic extended Herbrand-sequent

$$X\alpha \supset \bigwedge_{j=0}^{n-1} Xf^{jn}a, Pa, P\alpha \supset Pf\alpha, \ldots, Pf^{n-1}\alpha \supset Pf^n\alpha \to Pf^{n^2}a$$

which has complexity $2(n+1)$ and has the structure of the extended Herbrand-sequent of the proof π_n from Example 2.

5 Computing the Propositional Structure

Let

$$s^\star\colon \Gamma, X\alpha \supset \bigwedge_{i=1}^{n} Xs_i \to \Delta$$

be an induced schematic extended Herbrand-sequent (see Definition 4) for some fixed sequents s, s' and a term decomposition of $T(s,s')$ by $U \circ W$. The solution of the second phase consists in finding a substitution $\vartheta\colon \{X \leftarrow \lambda x.F(x)\}$ (where $F(x)$ is a quantifier-free formula which may contain the variable x but no other variable) s.t. the β-normal form of $s^\star\vartheta$ is a valid sequent.

The problem of finding a solution can be simplified by applying our (invertible) version of **LK** to s and decompose the formulas in s down to a set of two sequents of the form

$$\mathcal{S}\colon \{\Gamma \to \Delta, X\alpha;\ Xw_1, \ldots, Xw_n, \Gamma \to \Delta\}.$$

where Γ and Δ are sets of ground formulas, $W\colon \{w_1, \ldots, w_n\}$ is a set of ground terms and α is a constant which does not occur in W. Note that α is basically an eigenvariable, but in this context can be considered as a constant.

Definition 11. *Let s be a sequent, s' a corresponding Herbrand sequent and*

$$s^\star\colon\ \Gamma, X\alpha \supset \bigwedge_{i=1}^{n} Xw_i \to \Delta$$

be a schematic extended Herbrand sequent corresponding to the term decomposition $\mathcal{T}$ of $T(s,s')$ by $U \circ W$ for $W = \{w_1, \ldots, w_n\}$. Then the set of sequents $\mathcal{S}\colon \{s_1, s_2\}$ for

$$s_1 = Xw_1, \ldots, Xw_n, \Gamma \to \Delta,\ s_2 = \Gamma \to \Delta, X\alpha,$$

is called a cut-introduction problem *(CIP) w.r.t. $\mathcal{T}$. s_1 is called the W-sequent, and s_2 the α-sequent of $\mathcal{S}$. The sequent $\mathcal{S}_{\text{const}}\colon \Gamma \to \Delta$ is called the constant part of $\mathcal{S}$.*

Definition 12. *Let $\mathcal{S}$ be a CIP w.r.t. a term decomposition $\mathcal{T}$ and $F(x)$ be a quantifier-free formula s.t. $V(F(x)) \subseteq \{x\}$ and α does not occur in $F(x)$ (we call $F(x)$* admissible *for $\mathcal{S}$). The substitution $\vartheta\colon \{X \leftarrow \lambda x.F(x)\}$ is called a* solution *of $\mathcal{S}$ if $s_1\vartheta\downarrow$ and $s_2\vartheta\downarrow$ are both valid (where $\downarrow$ denotes normalization under β-reduction). $\mathcal{S}$ is called* solvable *if there exists a solution of $\mathcal{S}$.*

Remark 1. The restriction that α does not occur in $F(x)$ is necessary as, in case of solvability, the formula $(\forall x)F(x)$ is the cut-formula of the cut-introduction problem. As α is the eigenvariable of the quantifier-introduction on the left side of the cut, α may not appear in $(\forall x)F(x)$.

From now on we denote by $\mathcal{S}$ a CIP w.r.t. $\mathcal{T}$ where $\mathcal{T}$ is a decomposition of $T(s, s')$ by $U \circ W$ for $W = \{w_1, \ldots, w_n\}$, and by $F(x)$ an admissible formula for $\mathcal{S}$.

Definition 13. *Let* $s_1 = Xw_1, \ldots, Xw_n, A_1, \ldots, A_n \to B_1, \ldots, B_m$ *and* $s_2 = A_1, \ldots, A_n \to B_1, \ldots, B_m, X\alpha$ *and* $\mathcal{S}\colon \{s_1, s_2\}$ *be a CIP.*
The formula $G\colon\ A_1 \wedge \cdots \wedge A_n \wedge \neg B_1 \wedge \cdots \wedge \neg B_m$ *is called the* characteristic formula *of* $\mathcal{S}$.
The system $\mathcal{S}'\colon \{Xw_1, \ldots, Xw_n, G \to;\ G \to X\alpha\}$ *is called the* characteristic normal form *of* $\mathcal{S}$.

Lemma 7. *ϑ is a solution of a CIP $\mathcal{S}$ iff ϑ solves the characteristic normal form of $\mathcal{S}$.*

Proof. Trivial.

Lemma 8. *Let $\mathcal{S}$ be a CIP w.r.t. $\mathcal{T}$, and let G be the characteristic formula of $\mathcal{S}$. Then $G(w_1), \ldots, G(w_n) \to$ is valid.*

Proof. Let $W = \{w_1, \ldots, w_n\}$ and $\mathcal{S} = \{s_1, s_2\}$ such that $s_1 = Xw_1, \ldots, Xw_n$, $\Gamma' \to \Delta'$. By Proposition 4, for the original Herbrand-sequent $\Gamma'' \to \Delta''$ we have

$$\Gamma'' \to \Delta'' \subseteq (\Gamma' \to \Delta')\{\alpha \leftarrow w \mid w \in W\}.$$

Let $\Gamma' = A_1, \ldots, A_n$ and $\Delta' = B_1, \ldots, B_m$, then $G(\alpha) = A_1 \wedge \cdots A_n \wedge \neg B_1 \wedge \cdots \wedge \neg B_m$. The sequent $G(w_1), \ldots, G(w_n) \to$ can be transformed (via substitution application, and applying $\wedge\colon l$ and $\neg\colon l$ rules backwards) to the equivalent sequent

$$s_1''\colon (A_1, \ldots, A_k)\{\alpha \leftarrow w_1\}, \ldots, (A_1, \ldots, A_k)\{\alpha \leftarrow w_n\} \to$$
$$(B_1, \ldots, B_m)\{\alpha \leftarrow w_1\}, \ldots, (B_1, \ldots, B_m)\{\alpha \leftarrow w_n\} =$$
$$(\Gamma' \to \Delta')\{\alpha \leftarrow w \mid w \in W\}.$$

But

$$\Gamma'' \to \Delta'' = (\Gamma' \to \Delta')\{\alpha \leftarrow w \mid w \in W\} = s_1'',$$

as $\Gamma'' \to \Delta''$ is a Herbrand-sequent s_1'' is valid. Therefore $G(w_1), \ldots, G(w_n) \to$ is valid.

Theorem 3. *Let $\mathcal{S}$ be a system in characteristic normal form and let G be the characteristic formula. Then $\mathcal{S}$ is solvable and $\{X \leftarrow \lambda x.G\{\alpha \leftarrow x\}\}$ is a solution of $\mathcal{S}$.*

Proof. Let $\mathcal{S}\colon \{s_1, s_2\}$ be a cut-introduction problem for $s_1 = Xw_1, \ldots, Xw_n, \Gamma' \to \Delta'$, $s_2 = \Gamma' \to \Delta', X\alpha$, and $\Gamma' = A_1, \ldots, A_k,\ \Delta' = B_1, \ldots, B_m$, and G be the characteristic formula of the problem. We prove that $\theta = \{X \leftarrow \lambda x.G\{\alpha \leftarrow x\}\}$ is a solution of $\mathcal{S}$.

(a) $s_2'\colon s_2\theta \downarrow$ is valid. In fact,

$$s_2' = A_1, \ldots, A_k \to B_1, \ldots, B_m, A_1 \wedge \cdots \wedge A_k \wedge \neg B_1 \wedge \cdots \wedge \neg B_m.$$

Note that $(X\alpha)\{X \leftarrow \lambda x.G\{\alpha \leftarrow x\}\} \downarrow = G$.

(b) $s'_1 : s_1\theta\downarrow$ is valid:

$$s'_1 = (Xw_1)\theta\downarrow, \ldots, (Xw_n)\theta\downarrow, \Gamma' \to \Delta' =$$
$$(\lambda x.G\{\alpha \leftarrow x\})w_1\downarrow, \ldots (\lambda x.G\{\alpha \leftarrow x\})w_n\downarrow, \Gamma' \to \Delta' =$$
$$G(w_1), \ldots, G(w_n), \Gamma' \to \Delta'.$$

Since $G(w_1), \ldots, G(w_n) \to$ is valid by Lemma 8, s'_1 is valid.

Corollary 2. *Every cut-introduction problem is solvable.*

Proof. By Lemma 7 and Theorem 3.

In fact, once we have a decomposition of the substitution terms we find a canonical solution for the cut-formula. Roughly speaking this solution encodes the whole sequent $\Gamma' \to \Delta'$.

6 Improving the Canonical Solution

In the previous section, we have shown in Theorem 3 that for any CIP $\mathcal{S}$ there exists a solution $\{X \leftarrow \lambda x.G\{\alpha \leftarrow x\}\}$, where G is the characteristic formula of $\mathcal{S}$, such that $|G| = O(|\mathcal{S}|)$. Still for practical application of the method to the structuring of proofs, it will be important to further simplify the solution if possible, since the solution of the CIP corresponds to the cut-formula that is used to structure the proof. As a motivating example, consider the following.

Example 5. Let $\mathcal{S}$ be the CIP of the running example. Then the characteristic formula of $\mathcal{S}$

$$G(\alpha) = Pa \wedge \bigwedge_{0 \leq i < n} (Pf^i\alpha \supset Pf^{i+1}\alpha) \wedge \neg Pf^{n^2}a$$

gives rise to a solution. But there also exists a solution of constant logical complexity, using

$$H(\alpha) = P\alpha \supset Pf^n\alpha$$

which is preferable over the canonical solution based on $G(\alpha)$. Note that $G(\alpha) \models H(\alpha)$ but $H(\alpha) \not\models G(\alpha)$ and that $H(\alpha)$ only contains atoms that contain α.

We will now show that the observations from this example can be generalized and used to simplify the canonical solution. We will focus on characteristic formulas which are in conjunctive normal form. We will first give a sufficient criterion for simplification of such characteristic formulas, and then present an algorithm based on propositional resolution and validity checking that, given a solution, searches for a smaller one. First, note that the canonical solution is most general.

Proposition 7. *Let $\mathcal{S}$ be a CIP and $\vartheta = \{X \leftarrow \lambda x.F\}$ be a solution for $\mathcal{S}$. Then $G\{\alpha \leftarrow x\} \models F$, where G is the characteristic formula of $\mathcal{S}$.*

Proof. By Lemma 7, ϑ is a solution to the characteristic normal form of $\mathcal{S}$, and hence $(G \supset X(\alpha))\{X \leftarrow \lambda x.F\} = G \supset F\{x \leftarrow \alpha\}$ is valid. Therefore $(G \supset F\{x \leftarrow \alpha\})\{\alpha \leftarrow x\} = G\{\alpha \leftarrow x\} \supset F$ is valid.

Note that the converse does not hold: in general, $G \models \top$ but $\{X \leftarrow \lambda x.\top\}$ is not a solution of the CIP of our running example.

Proposition 8. *Let G be a characteristic formula of the CIP $\mathcal{S}$ and assume that G is in conjunctive normal form. Let G' be obtained from G by removing all clauses that do not contain α. Then $\{X \leftarrow \lambda x.G'\{\alpha \leftarrow x\}\}$ is a solution for $\mathcal{S}$.*

Let F be a formula in conjunctive normal form, i.e. $F = \bigwedge_{i\in\{1,\dots,m\}} C_i$, with clauses $C_i = \bigvee_{j\in\{1,\dots,n_i\}} L_{i,j}$, where the $L_{i,j}$ are literals. By $\overline{L}$ we denote the dual of a literal L. For two clauses C_i, C_j, if there exists exactly one pair (k,ℓ) such that $L_{i,k} = \overline{L_{j,\ell}}$, we define their *resolvent*

$$\mathrm{res}(C_i, C_j) = \bigvee_{r\in\{1,\dots,n_i\}\setminus k} L_{i,r} \vee \bigvee_{q\in\{1,\dots,n_j\}\setminus \ell} L_{j,q}$$

and leave $\mathrm{res}(C_i, C_j)$ undefined otherwise.

Then define

$$\mathcal{R}(F) = \{\mathrm{res}(C_i, C_j) \wedge \bigwedge_{k\in\{1,\dots,m\}\setminus\{i,j\}} C_k \mid \mathrm{res}(C_i, C_j) \text{ defined}\}.$$

Note that if $G \in \mathcal{R}(F)$ then $|G| < |F|$. Since $C_i \wedge C_j \supset \mathrm{res}(C_i, C_j)$, we have

Lemma 9. *If $H \in \mathcal{R}(F)$ then $F \supset H$ is valid.*

This directly translates to a result on CIPs:

Proposition 9. *Let $\mathcal{S} = \{Xw_1, \dots, Xw_n, \Gamma \to \Delta,\ \Gamma \to \Delta, X\alpha\}$ be a CIP and $H \in \mathcal{R}(F)$.*

(1) If $F(w_1), \dots, F(w_n), \Gamma \to \Delta$ is not valid, then $\{X \leftarrow \lambda x.H(x)\}$ is not a solution for $\mathcal{S}$.
(2) If $\Gamma \to \Delta, F(\alpha)$ and $H(w_1), \dots, H(w_n), \Gamma \to \Delta$ are valid, then $\{X \leftarrow \lambda x.H(x)\}$ is a solution for $\mathcal{S}$.

Proof. For showing (1), assume that $H(w_1), \dots, H(w_n), \Gamma \to \Delta$ is valid. Then by Lemma 9, $F(w_1), \dots, F(w_n), \Gamma \to \Delta$ is valid.

For (2), it suffices to show that $\Gamma \to \Delta, H(\alpha)$ is valid, which follows from the same Lemma.

Propositions 7, 8 and 9 suggest a resolution-based method to find more efficient solutions for a CIP $\{Xw_1, \dots, Xw_n, \Gamma \to \Delta,\ \Gamma \to \Delta, X\alpha\}$, starting from a canonical solution G in conjunctive normal form: First, apply Proposition 8 to remove unnecessary clauses from G to obtain G'. Then, compute $\mathcal{R}(G')$. Since G' yields a solution, we have $\Gamma \to \Delta, G'(\alpha)$ and hence it suffices to check for

$F \in \mathcal{R}(G')$ whether $F(w_1), \ldots, F(w_n), \Gamma \to \Delta$ is valid to determine whether F yields a solution. If it is valid, we iterate the procedure on F. If it is not valid, then we know that no iteration of $\mathcal{R}$ on F will yield a solution, so we can abort the search on this branch of the search tree. Since on each branch of the search tree, the size of solutions decreases, the search terminates.

Example 6. Let $\mathcal{S}$ be the CIP of the running example for $n = 2$, which has the characteristic formula, written in conjunctive normal form,

$$G(\alpha)\colon\ Pa \wedge (\neg P\alpha \vee Pf\alpha) \wedge (\neg Pf\alpha \vee Pf^2\alpha) \wedge \neg Pf^4a.$$

Application of Proposition 8 yields

$$G'(\alpha)\colon\ (\neg P\alpha \vee Pf\alpha) \wedge (\neg Pf\alpha \vee Pf^2\alpha).$$

We have $\mathcal{R}(G') = \{\neg P\alpha \vee Pf^2\alpha\}$. By (2) of Proposition 9, it suffices to check whether

$$Pa, \neg Pa \vee Pf^2a, \neg Pf^2a \vee Pf^4a \to Pf^4a$$

is valid, which is the case. Since $\mathcal{R}(\neg P\alpha \vee Pf^2\alpha) = \emptyset$, search terminates and we have found a smaller solution. In general, the algorithm obtains the solution $\neg P\alpha \vee Pf^n\alpha$ after a linear number of iterations.

7 Conclusion

We have presented a method for cut-introduction which computes a quantified cut-formula from a structural analysis of a cut-free proof. This paper is a first step towards algorithmically feasible proof compression by cut-introduction.

As further work we plan to extend the method: the introduction of an arbitrary number of $\forall$-cuts can be dealt with based on the results in [5] using a decomposition calculus where lines have a flexible width. The extension from single quantifiers to blocks of quantifiers consists in replacing a single variable by a vector of variables. The treatment of cuts with quantifier alternations first requires a description of the structure of Herbrand-sequents obtained from such proofs (along the lines of Proposition 3) which is an interesting theoretical problem.

In order to study this method in a realistic context we plan to implement it within the existing `gapt`-project[2] and to apply it to the output of automated theorem provers.

References

1. 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)

[2] `http://code.google.com/p/gapt/`

2. Finger, M., Gabbay, D.: Equal Rights for the Cut: Computable Non-analytic Cuts in Cut-based Proofs. Logic Journal of the IGPL 15(5–6), 553–575 (2007)
3. Gentzen, G.: Untersuchungen über das logische Schließen. Mathematische Zeitschrift 39, 176–210, 405–431 (1934–1935)
4. Hetzl, S.: Describing proofs by short tautologies. Annals of Pure and Applied Logic 159(1–2), 129–145 (2009)
5. Hetzl, S.: Applying Tree Languages in Proof Theory. In: Dediu, A.-H., Martín-Vide, C. (eds.) LATA 2012. LNCS, vol. 7183, pp. 301–312. Springer, Heidelberg (2012)
6. Hetzl, S., Leitsch, A., Weller, D.: Towards Algorithmic Cut-Introduction. technical report, http://www.logic.at/people/hetzl/
7. 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/Calculemus/MKM 2008. LNCS (LNAI), vol. 5144, pp. 462–477. Springer, Heidelberg (2008)
8. 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)
9. Orevkov, V.P.: Lower bounds for increasing complexity of derivations after cut elimination. Zapiski Nauchnykh Seminarov Leningradskogo Otdeleniya Matematicheskogo Instituta 88, 137–161 (1979)
10. Pudlák, P.: The Lengths of Proofs. In: Buss, S. (ed.) Handbook of Proof Theory, pp. 547–637. Elsevier (1998)
11. Statman, R.: Lower bounds on Herbrand's theorem. Proceedings of the American Mathematical Society 75, 104–107 (1979)
12. Troelstra, A.S., Schwichtenberg, H.: Basic Proof Theory, 2nd edn. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press (2000)
13. Vyskočil, J., Stanovský, D., Urban, J.: Automated Proof Compression by Invention of New Definitions. In: Clarke, E.M., Voronkov, A. (eds.) LPAR-16 2010. LNCS, vol. 6355, pp. 447–462. Springer, Heidelberg (2010)
14. Woltzenlogel Paleo, B.: Atomic Cut Introduction by Resolution: Proof Structuring and Compression. In: Clarke, E.M., Voronkov, A. (eds.) LPAR-16 2010. LNCS, vol. 6355, pp. 463–480. Springer, Heidelberg (2010)

Conflict Anticipation in the Search for Graph Automorphisms

Hadi Katebi, Karem A. Sakallah, and Igor L. Markov

EECS Department, University of Michigan
{hadik,karem,imarkov}@umich.edu

Abstract. Effective search for graph automorphisms allows identifying symmetries in many discrete structures, ranging from chemical molecules to microprocessor circuits. Using this type of structure can enhance visualization as well as speed up computational optimization and verification. Competitive algorithms for the graph automorphism problem are based on efficient partition refinement augmented with group-theoretic pruning techniques. In this paper, we improve prior algorithms for the graph automorphism problem by introducing *simultaneous refinement of multiple partitions*, which enables the anticipation of future conflicts in search and leads to significant pruning, reducing overall runtimes. Empirically, we observe an exponential speedup for the family of Miyazaki graphs, which have been shown to impede leading graph-automorphism algorithms.

1 Introduction

An *automorphism* (*symmetry*) of a graph is a *permutation* of the graph's vertices that preserves the graph's edge relation. The set of all symmetries of a graph forms a *group*[1] under functional composition. The *graph automorphism problem* seeks a generating set for the automorphism group of a graph. Closely related to graph automorphism is the problem of *canonical labeling* which assigns a unique signature to a graph that is invariant under all possible labelings of its vertices. Graph automorphisms and canonical labelings are related to the functional properties of the combinatorial objects in question. In a representative application developed in [3,2], a CNF (conjunctive normal form) formula is modeled by a graph and passed to a symmetry detection program. During subsequent symmetry-breaking, these symmetries are used to augment the formula with a set of symmetry-breaking predicates. These predicates do not change the formula's satisfiability, but help SAT solvers prune away symmetric portions of the search space.

Graph symmetry and canonical labeling have been extensively studied over the past five decades. The **nauty** program [18,19], developed by McKay in 1981,

[1] A group is an algebraic structure comprising a non-empty set of elements with a binary operation that is *associative*, admits an *identity* element, and is *invertible*. For example, the set of integers with addition forms a group. A *generating set* of a group is a subset of the group's elements whose combinations under the group operation generate the entire group.

N. Bjørner and A. Voronkov (Eds.): LPAR-18, LNCS 7180, pp. 243–257, 2012.

pioneered the first high-performance algorithms that inspired all subsequent tools. Almost two decades later, Darga et al [9] observed that the use of an adjacency matrix in **nauty** could lead to asymptotic inefficiencies in dealing with sparse graphs. This motivated the development of a new tool called **saucy** [9,10,16], which was limited to just finding a set of symmetry generators, but was three orders of magnitude faster than **nauty** on very large and very sparse graphs. Closely following **nauty**'s canonical labeling algorithms were two other tools, namely, **bliss** [13,14] and **nishe** [22]. The search routines in **bliss** improved the handling of large and sparse graphs, and the branching heuristics in **nishe** facilitated a polynomial-time solution for the Miyazaki graphs [20], a family of graphs that **nauty** requires exponential time to process.

Since the emergence of the first version of **saucy** in 2004 (**saucy** 1.1) [9], different algorithmic enhancements improved **saucy**'s performance over a wide range of graphs with both theoretical and practical interest. The second version of **saucy** (**saucy** 2.0) [10] incorporated the observation that the symmetry generators of sparse graphs were mostly sparse. The major algorithmic changes that were introduced in **saucy** 2.0 separated the search for symmetries from the search for a canonical labeling. Further improvements to **saucy**'s data structures and algorithms were reported in **saucy** 2.1 [16].

In this paper, we present **saucy** 3.0 which performs *simultaneous partition refinement* to anticipate and avoid possible future conflicts. The procedure augments the method introduced in **saucy** 2.1 whereby nodes in the search tree represent sets of vertex permutations encoded by an *ordered partition pair (OPP)* of graph vertices. The basic idea of the new procedure is to refine the top and bottom partitions of an OPP at the same time, making sure that the two partitions conform to each other (according to the graph's edge relation) after each refinement step. We implemented this enhancement in **saucy** 3.0 and tested its performance on a wide variety of graph benchmarks. Our experimental evaluation shows that this enhancement can significantly prune the search tree for many graph families, such as the Miyazaki graphs. Furthermore, the concept of simultaneous refinement helps us better understand and explain the validity of some of the algorithms that were previously presented in **saucy** 2.1.

In the remainder, we first review some preliminaries in Section 2. Then, we discuss **saucy**'s baseline algorithms in Section 3. The new partitioning algorithm based on the concept of simultaneous refinement is presented in Section 4. Section 5 establishes the correctness of "matching OPP" pruning (this pruning mechanism was presented in **saucy** 2.1). The results of our experimental study are provided in Section 6. Finally, we discuss conclusions in Section 7.

2 Preliminaries

We assume familiarity with basic notions from group theory, including such concepts as groups, subgroups, group generators, cosets, orbit partition, etc. Information on different group theoretic concepts is available in many abstract algebra texts such as [11]. In this paper, we focus on the automorphisms of an n-vertex *colored graph* G whose vertex is $V = \{0, 1, ..., n-1\}$. A *permutation*

of V is a bijection from V to V, and a *symmetry* of G is a permutation of V that preserves G's edge relation. Permutation α, when applied to G, produces the permuted graph G^α. Every graph has a trivial symmetry, called the *identity*, that maps each vertex to itself. The set of symmetries of G forms a *group* under functional composition. This group is the *symmetry group* of G, and is denoted by $Aut(G)$. Given G, the objective of any symmetry detection tool is to find a set of *group generators* for $Aut(G)$.

An *ordered partition* $\pi = [W_1|W_2|\cdots|W_m]$ of V is an ordered list of non-empty pair-wise disjoint subsets of V whose union is V. The subsets W_i are called the *cells* of the partition. Ordered partition π is *unit* if $m = 1$ (i.e., $W_1 = V$) and *discrete* if $m = n$ (i.e., $|W_i| = 1$ for $i = 1, \cdots, n$). An *ordered partition pair (OPP)* π is specified as

$$\Pi = \begin{bmatrix} \pi_T \\ \pi_B \end{bmatrix} = \begin{bmatrix} T_1 \,|T_2\,| \cdots |T_m \\ B_1 \,|B_2\,| \cdots |B_k \end{bmatrix}$$

with π_T and π_B referred to, respectively, as the top and bottom ordered partitions of π. OPP π is *isomorphic* if $m = k$ and $|T_i| = |B_i|$ for $i = 1, \cdots, m$; otherwise it is *non-isomorphic*. In other words, an OPP is isomorphic if its top and bottom partitions have the same number of cells, and corresponding cells have the same cardinality. An isomorphic OPP is *matching* if its corresponding non-singleton cells are *identical*. We will refer to an OPP as discrete (resp. unit) if its top and bottom partitions are discrete (resp. unit).

OPPs lie at the heart of **saucy**'s symmetry detection algorithms, since each OPP compactly represents a set of permutations. This set of permutations might be empty (non-isomorphic OPP), might have only one permutation (discrete OPP), or might consist of up to $n!$ permutations (unit OPP). Several OPP examples and the permutation set encoded by them are provided below.

- Discrete OPP: $\left[\begin{array}{c|c|c} 2 & 0 & 1 \\ 1 & 2 & 0 \end{array}\right] = \{(0\ 2\ 1)\}$
- Unit OPP: $\begin{bmatrix} 0,1,2 \\ 0,1,2 \end{bmatrix} = \{\iota, (0\ 1), (0\ 2), (1\ 2), (0\ 1\ 2), (0\ 2\ 1)\}$
- Isomorphic OPP: $\left[\begin{array}{c|c} 2 & 0,1 \\ 1 & 2,0 \end{array}\right] = \{(1\ 2), (0\ 2\ 1)\}$
- Matching OPP: $\left[\begin{array}{c|c|c} 1 & 0,2,4 & 3 \\ 3 & 0,2,4 & 1 \end{array}\right] = (1\ 3) \circ S_3(\{0,2,4\})$
- Non-isomorphic OPPs: $\left[\begin{array}{c|c} 0,2 & 1 \\ 1 & 2,0 \end{array}\right] = \emptyset$, $\left[\begin{array}{c|c|c} 2 & 0 & 1 \\ 1 & 2,0 \end{array}\right] = \emptyset$

3 Baseline Algorithms

Similar to other combinatorial search algorithms, **saucy** explores the space of permutations by building a search tree and systematically traversing it. However, the representation of search nodes as OPPs in **saucy** is unique. The root of the tree is a unit OPP which is initially *refined* based on the colors and degrees of the

vertices of the input graph. The depth-first traversal of the permutation space is started by choosing a *target* vertex from a non-singleton cell of the top partition and mapping it to all the vertices of the corresponding cell of the bottom partition. To propagate the *constraints of the graph* (i.e. the graph's edge relation), partition refinement is invoked after each mapping decision. The mapping procedure continues until the OPP becomes discrete, matching, or non-isomorphic (the latter is referred to as a *conflict*). In either case, **saucy** backtracks one level up, and maps the target vertex to the remaining candidate vertices. The search ends when all possible mappings are exhausted.

In addition to partition refinement, **saucy** exploits two types of pruning mechanisms: *group-theoretical* and *OPP-based*. To enable group-theoretical pruning, namely *coset* and *orbit* pruning, the left-most path of the tree should correspond to a sequence of *subgroup stabilizers* ending in the identity. In other words, the decisions along the left-most path maps each vertex to itself. This phase of the search is called *subgroup decomposition*. Note that no such requirement is needed in the remaining parts of the search tree. In contrast, OPP-based pruning mechanisms are optional techniques that assist **saucy**'s algorithms to avoid unnecessary search. Two of these techniques, embedded in **saucy** 2.1, are *non-isomorphic OPP* and *matching OPP* pruning.

In this paper, we introduce an enhanced partition refinement procedure that refines the top and bottom partitions of an OPP simultaneously. Our simultaneous refinement anticipates the conflicts that might arise in a certain subtree, and prunes the entire subtree without exploring it. The idea here is to capture conflicts that might be overlooked by the conventional refinement procedure.

4 Conflict Anticipation via Simultaneous Refinement

Partition refinement in **saucy** is adapted from **nauty**, and **nauty**'s refinement is based on the concept of *equitable* partitions. Partition $\pi = [W_1|W_2|\cdots|W_m]$ is equitable (with respect to graph G) if, for all $v_1, v_2 \in W_i$ $(1 \leq i \leq m)$, the number of neighbors of v_1 in W_j $(1 \leq j \leq m)$ is equal to the number of neighbors of v_2 in W_j. Although **saucy**'s partition refinement is adapted from **nauty**, the search tree in **saucy** is completely different from that in **nauty**. The nodes of **nauty**'s tree are **single ordered partitions**, while the nodes of **saucy**'s tree are **ordered partition pairs**. In **nauty**, an equitable partition is obtained by invoking partition refinement after each vertex *individualization*. Extending this to OPPs, the refinement procedure in **saucy** refines both partitions of an OPP *simultaneously* after each mapping decision, until 1) both partitions become equitable and the resulting OPP is isomorphic, or 2) the resulting OPP is non-isomorphic indicating an empty set of permutations, i.e., a conflict. In **saucy** 2.1 and earlier, simultaneous refinement was basically an algorithmic enhancement that detected conflicts (if any existed) earlier during refinement, without fully establishing an equitable OPP (an OPP whose top and bottom partitions are both equitable), and then examining the resulting OPP to see whether it was isomorphic/non-isomorphic. In implementation, **saucy** first refines the top

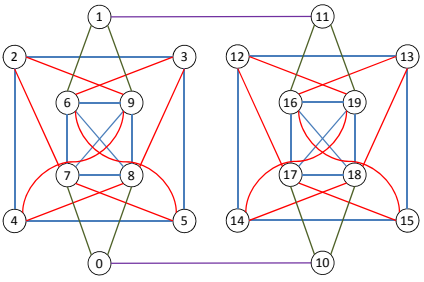


Fig. 1. A 20-vertex 46-edge graph with symmetry group of size 32

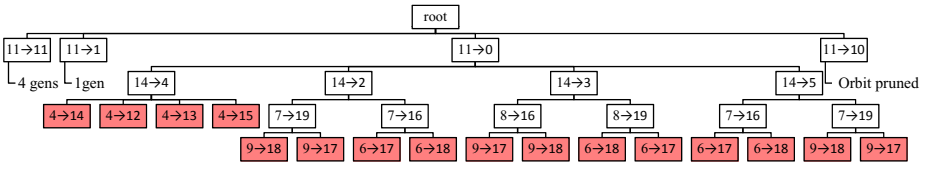


Fig. 2. The search tree constructed by **saucy** 2.1 for the graph in Figure 1

partition until it becomes equitable, records where the cell splits occur, then starts refining the bottom partition, and compares the splitting locations of the bottom to the top whenever a new split occurs (i.e., checks the isomorphism of the two partitions after each split).

In this section, we argue that the significance of simultaneous refinement is not limited to the early detection of "non-isomorphic equitable OPPs". In particular, we demonstrate cases where the resulting equitable OPP is isomorphic, but the OPP still violates the edge relation of the graph. We illustrate such a case, and explain why conventional refinement fails to detect the conflict in that case. We then present an *enhanced simultaneous refinement* procedure that detects such cases and does not explore them. We discuss the impact of our proposed refinement procedure on the search tree constructed for our example.

Consider the 20-vertex 46-edge graph shown in Figure 1. The search tree generated by **saucy** 2.1 for this graph is shown in Figure 2. This search tree produces 16 conflicts (non-isomorphic OPPs), indicated by red-shaded nodes. In the remainder of this section, we focus on the path from the root that maps $11 \mapsto 0$ and then $14 \mapsto 4$. The OPPs in Figure 3a, labeled with (1), (2) and (3), represent the nodes of the search tree at the root, after mapping $11 \mapsto 0$, and after mapping $14 \mapsto 4$, respectively.

In **saucy** 2.1, the isomorphic OPP (3), obtained after mapping $14 \mapsto 4$, is not considered to be a conflict node and triggers further vertex mappings (namely, $4 \mapsto 14$, $4 \mapsto 12$, $4 \mapsto 13$, and $4 \mapsto 15$). However, this OPP violates the edge relation of the graph in Figure 1. To see this, consider the edge that connects 13 to 16. This edge, according to OPP (3), should be mapped to another edge

$$\left[\begin{array}{l|l|l} 11,10,1,0 & 15,12,14,13,5,2,4,3 & 18,19,17,16,8,9,7,6 \\ 11,10,1,0 & 15,12,14,13,5,2,4,3 & 18,19,17,16,8,9,7,6 \end{array}\right] \quad (1)$$

$$\left[\begin{array}{l|l|l|l|l|l|l|l|l} 0 & 10 & 1 & 11 & 14,12,13,15 & 2,4,5,3 & 17,18 & 8,7 & 6,9 \\ 11 & 1 & 10 & 0 & 4,3,5,2 & 13,14,12,15 & 9,6 & 19,16 & 18,17 \end{array}\right] \quad (2)$$

$$\left[\begin{array}{l|l|l|l|l|l|l|l|l|l|l|l} 0 & 10 & 1 & 11 & 13 & 12 & 15 & 14 & 2,4,5,3 & 17 & 18 & 8,7 & 6,9 & 16 & 19 \\ 11 & 1 & 10 & 0 & 3 & 2 & 5 & 4 & 13,14,12,15 & 6 & 9 & 19,16 & 18,17 & 7 & 8 \end{array}\right] \quad (3)$$

Fig. 3a. The search nodes of the tree in Figure 2. OPP (1) is at the root, OPP (2) is after mapping $11 \mapsto 0$, and OPP (3) is after mapping $14 \mapsto 4$.

$$[0|10|1|11|12,13,15|14|2,4,5,3|17,18|8,7|6,9|16,19] \quad (4)$$

$$[0|10|1|11|13|12,15|14|2,4,5,3|18|17|8,7|6,9|16|19] \quad (5)$$

$$[0|10|1|11|13|12|15|14|2,4,5,3|18|17|8,7|6,9|16|19] \quad (6)$$

Fig. 3b. The refinement of the top partition of OPP (2) to get OPP (3)

$$[11|1|10|0|3,5,2|4|13,14,12,15|9,6|19,16|18,17|7,8] \quad (7)$$

$$[11|1|10|0|3|5,2|4|13,14,12,15|9|6|19,16|18,17|7|8] \quad (8)$$

$$[11|1|10|0|3|2|5|4|13,14,12,15|9|6|19,16|18,17|7|8] \quad (9)$$

Fig. 3c. The refinement of the bottom partition of OPP (2) to get OPP (3)

that connects 3 to 7, since OPP (3) maps $13 \mapsto 3$, and $16 \mapsto 7$. Nevertheless, no such edge exists between 3 and 7 in Figure 1, and hence, OPP (3) is a conflict.

The question now is why the refinement procedure failed to detect the above conflict? Or, in other words, why was OPP (3) found to be isomorphic? To answer this question, we should follow the trace of the refinement procedure which is performed on OPP (2) to get OPP (3) after mapping $14 \mapsto 4$. As elaborated earlier, **saucy** first refines the top partition until it becomes equitable, then refines the bottom partition and checks the isomorphism of the bottom to the top whenever a new split occurs. The step by step refinement of the top and bottom partitions when $14 \mapsto 4$ is shown in Figure 3b and Figure 3c, respectively.

The refinement on the top starts by first making 14 a singleton cell (partition (4)). According to the graph of Figure 1, 14 is connected to 12,15,18 and 19, but not to 13, 17 and 16. Hence, refinement separates 12 and 15 from 13 (this makes 13 a singleton cell), 18 from 17, and 19 from 16 (partition (5)). The refinement continues by looking at the connections of one of the newly created cells. Here, **saucy** picks the singleton cell 16. According to the graph, 16 is connected to 11,13,15,17,18 and 19. This separates 15 from 12 (partition (6)). The top partition is now equitable, i.e., no further refinement is implied.

After refining the top partition, **saucy** starts refining the bottom partition. This is done by first making 4 a singleton cell (partition (7)). Since 4 is connected to 2,5,8 and 9, refinement separates 2 and 5 from 3 (this makes 3 a singleton cell), 9 from 6, and 8 from 7 (partition (8)). Note that, at this point, partition

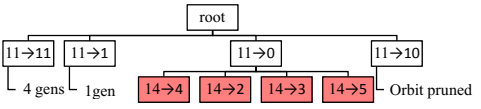


Fig. 4. The search tree constructed by **saucy** 3.0 for the graph in Figure 1

(8) is isomorphic to partition (5), i.e., no conflict is detected. This time **saucy** picks the singleton cell 7, since it had previously chosen 16 from the top, and 7 is at the same index on the bottom as 16 on the top. According to the graph, 7 is connected to 0,2,5,6,8 and 9. Since 7 is connected to both 2 and 5, no further refinement is implied. At this point, **saucy** should *detect the conflict that 16 on the top separated 15 from 12, but 7 on the bottom did not distinguish 2 from 5.* However, since no new cell is created on the bottom, **saucy** does not invoke the isomorphism check, and falsely assumes that the bottom stays isomorphic to the top. Note that the failure to detect this conflict is not a bug in refinement, since **nauty**'s (and essentially **saucy**'s) refinement procedure refines one partition at a time, and checks isomorphism once both partitions are equitable. After refining based on 7, **saucy** refines based on 6. Vertex 6 is connected to 1,3,5,7,8 and 9. Since 6 is connected to 5 but not 2, it separates 5 from 2 (partition 9). The bottom partition is now equitable and isomorphic to the top.

After the refinement procedure ends, **saucy** builds isomorphic OPP (3), and starts exploring it by mapping 4 to 14, 12, 13, and 15. However, this phase of the search is superfluous, since we know that OPP (3) violates the graph's edge relation, and its further exploration will always result in conflicts. Another case of a conflicting isomorphic OPP is when two corresponding *singleton cells* of the top and bottom partitions have different connections to the other *singleton cells* of their own partition. In this case, the conflict is again overlooked by **saucy**'s conventional refinement procedure, since singleton cells cannot be partitioned to smaller cells (i.e., no new cell splitting occurs), and hence, the top and bottom partitions remain isomorphic after this step of refinement.

To detect the conflicts that might remain undetected during partition refinement, we enhanced **saucy**'s partition refinement in two ways; 1) the isomorphism of the bottom partition to the top is checked *after each refinement step, rather than after each time a new split occurs*, and 2) in addition to the isomorphism check, we also ensure that *the connections of each newly created cell on the bottom match the connections of its corresponding cell on the top.* These two new checks verify that the top and bottom partitions remain isomorphic and conforming (according to the graph's edge relation) after each refinement step. In our implementation, the overhead of the first check is negligible, as it is performed within the main refinement loop, but the second check requires an extra iteration over the outgoing edges of the vertices of the newly created cells. We would like to emphasize that our enhancement is *enabled* by the OPP-encoding of permutations that is unique to **saucy**'s search for automorphisms.

Figure 4 shows the search tree for the graph in Figure 1 when our new simultaneous refinement is invoked. Comparing this search tree to that in Figure 2, the number of conflicts is reduced from 16 to 4.

5 The Validity of Matching OPP Pruning

When matching OPP π is encountered in the search, **saucy** "constructs" a permutation α from π by mapping the vertices in matching cells identically. It then uses α to prune the entire subtree rooted at this OPP in one of two ways; either 1) α is an automorphism of the graph, which means that the subtree is a coset of the stabilizer subgroup, and α is a coset representative, or 2) α is not an automorphism, which indicates that the subtree is not a coset, and the search for a coset representative in that subtree will always fail. In this section, we show that, if π is found to be matching by our enhanced simultaneous refinement (described in Section 4), the second case cannot occur, i.e., α must always be an automorphism of the graph. The proof of this claim is presented next.

Assume that π is an OPP that is found matching by our enhanced refinement procedure. This means that π is equitable, isomorphic, matching, and conforming according to G's edge relation. Let α be the permutation that corresponds to π, i.e., the permutation that maps the vertices in π's non-singleton cells identically. To show by contradiction that α is a symmetry of G, assume that it is not. Then, there must be an edge in G^α that does not exist in G (or vice versa). Assume that this edge connects v_1 to v_2. Trivially, both v_1 and v_2 cannot be mapped identically in α, otherwise, an edge between v_1 and v_2 in G would map to the exact same edge in G^α. Hence, permutation α either maps v_1 to v_1' ($v_1 \neq v_1'$), or v_2 to v_2' ($v_2 \neq v_2'$), or both. We first consider the case where v_1 is mapped to v_1' but v_2 is mapped identically (this is similar to the case where v_2 is mapped to v_2' but v_1 is mapped identically). This case contradicts our assumption that π is equitable, since v_1 and v_1' were both singleton cells of π, and having an edge between v_1 and v_2 but not between v_1' and v_2 would imply further refinement on π. Now consider the case where v_1 is mapped to v_1' and v_2 to v_2'. This case contradicts our assumption that π is conforming according to G's edge relation, since v_1, v_2, v_1' and v_2' were all singleton cells of π, and having an edge between v_1 and v_2 but not between v_1' and v_2' would violate G's edge relation.

6 Experimental Evaluation

We implemented our simultaneous partition refinement technique in **saucy** 3.0, and tested its performance on 1445 graph benchmarks drawn from a wide variety of domains. Our experiments were conducted on a SUN workstation equipped with a 3GHz Intel Dual-Core CPU, a 6MB cache and an 8GB RAM, running the 64-bit version of Redhat Linux. A time-out of 1000 seconds was applied. Table 1 lists the benchmark families used in our experiments. For these families, the name, the number of instances, the size of the smallest and largest instances, and a short description are provided. The families are divided into

Table 1. Benchmark families

Family	Instances	Smallest Instance vertices	Smallest Instance edges	Largest Instance vertices	Largest Instance edges	Description
`mz` [20,15]	25	40	60	1,000	1,500	Original Miyazaki graphs
`cmz` [15]	46	120	90	200	1,900	(`mz`), and their variants
`mz-aug` [15]	25	40	92	1,000	2,300	designed to mislead the
`mz-aug2` [15]	24	96	152	1,200	1,900	**bliss** cell selector
`circuit` [23,1]	33	3,575	14,625	4,406,950	8,731,076	**saucy** benchmarks from
`router` [7,12]	3	112,969	181,639	284,805	428,624	place-route, verification,
`roadnet` [6]	56	1,158	1,008	1,679,418	2,073,394	routers & road networks
`application` [8]	300	464	2,066	32,813,545	65,487,132	SAT 2011 application,
`crafted` [8]	300	105	320	776,820	3,575,337	crafted and random
`random` [8]	600	1,165	5,375	310,000	680,000	CNF instances
`binnet` [17,4]	33	1,000	720	6,000,000	4,391,515	binary networks

four categories. These categories were chosen based on the general construction of the graphs, considering metrics such as the number of vertices and edges, connectivity and sparsity. The first category is the Miyazaki graphs [20,15], which **nauty** takes exponential time to process. The second category contains benchmarks used to test earlier versions of **saucy**. It represents graphs from various domains, such as logic circuits and their physical layouts [23,1], internet routers [7,12], and road networks in the US states and its territories [6]. The third category includes CNF benchmarks from the international SAT 2011 competition [8]. The fourth category consists of graphs not previously reported in graph automorphism or satisfiability research. These graphs were proposed for testing community-detection algorithms [17,4] [2].

Figure 5 compares the number of conflicts produced by **saucy** 3.0 and **saucy** 2.1. If a benchmark is not processed within the time-out, the number of conflicts encountered right before termination is reported. The results show that **saucy** 3.0 always produces fewer or the same number of conflicts. This is expected, as our proposed refinement procedure anticipates and avoids certain conflicts that might arise in **saucy** 2.1. Of all the benchmark families, `mz-aug` and `mz-aug2` benefit most from the new refinement procedure. For these two families, the highest number of conflicts reported by **saucy** 3.0 was 696 (for `mz-aug-50`). In contrast, the number of conflicts reported by **saucy** 2.1 was at least 10,000 for 46 out of 49 `mz-aug` and `mz-aug2` instances. Of the remaining two Miyazaki families, `mz` did not experience any change in its number of conflicts, and `cmz` showed a slight improvement for 5 out of its 46 instances (8 fewer conflicts were reported for those 5 instances). Of the graphs from circuits, internet routers,

[2] We used the implementation of the algorithm described in [17] (available at [4]) to generate 33 undirected and unweighted binary networks. We set the number of nodes to $\{1, ..., 9\} \times \{10^3, 10^4, 10^5\}$ and $\{1, ..., 6\} \times 10^6$ (generating larger networks required more than 8GB RAM), and fixed the remaining parameters in all instances. Specifically, we set the average degree to 2, the max degree to 4, the mixing parameter to 0.1, the minimum community size to 20, and the maximum community size to 50.

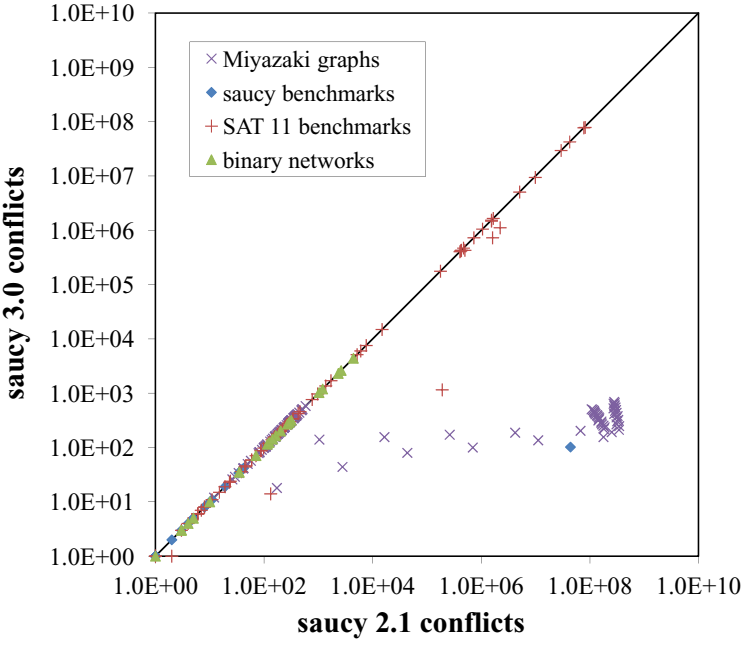


Fig. 5. Number of conflicts returned by **saucy** 3.0 versus **saucy** 2.1

and road networks, only one instance (from `circuit`) showed significant conflict reduction (from 43 million to only 102). The remaining instances produced the same number of conflicts (not more than 42) in **saucy** 3.0 and **saucy** 2.1. Of the 1200 CNF benchmarks, only 72 (15 from application and 57 from crafted) encountered conflicts in **saucy** 2.1, and only 12 (all from crafted) experienced a reduction in the number of conflicts. The smallest reduction was 1 and the largest was 2.9 million. The `binnet` instances also produced the same results in both versions of **saucy**. The reported number of conflicts for those instances ranged from no conflicts to 4,412.

Figure 6 shows the distribution of *depth* of the conflicts that were captured and avoided by **saucy** 3.0. Recall that the new refinement procedure in **saucy** 3.0 prunes some subtrees that are explored by **saucy** 2.1. Suppose that one such subtree is found to be conflicting at level l in **saucy** 3.0, but leads to c conflicts in **saucy** 2.1, where the n-th conflict ($1 \leq n \leq c$) occurs at level l_n. Trivially, $l_n \geq l$. We define the depth of the n-th conflict as $d = l_n - l$. If $d = 0$, both **saucy** 3.0 and **saucy** 2.1 capture the conflict at the same time. If $d > 0$, **saucy** 3.0 anticipates and avoids the conflict d levels sooner than it occurs in **saucy** 2.1. We use conflict depth as a numeric criterion to evaluate the effectiveness of our new refinement procedure. The results in Figure 6 show that the deepest conflicts captured by **saucy** 3.0 occur in the instances of Miyazaki families. The greatest reported depth was 98, which occurred 2.8×10^8 times for `mz-aug-50`. The only benchmark from the `circuit` family that had significant

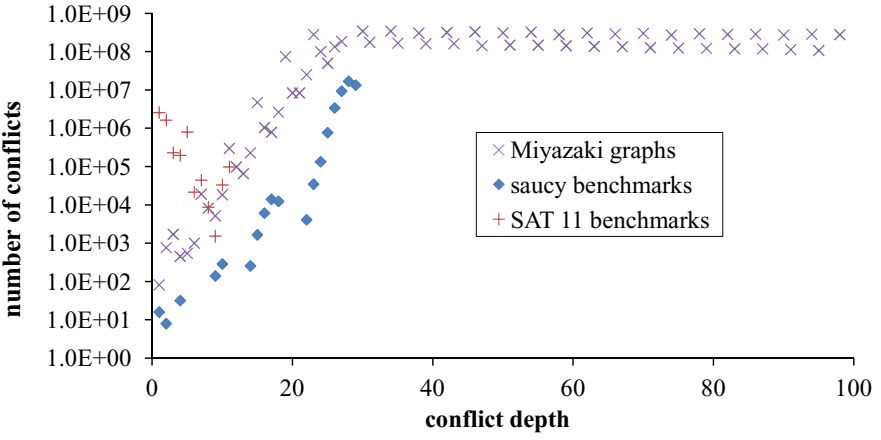


Fig. 6. Histogram of the conflict depths captured by **saucy** 3.0

conflict reduction produced conflict depth of up to 29, where the largest conflict depth happened 1.3×10^7 times. For the CNF benchmarks, the deepest reported conflict had a depth of 11, and occurred roughly 10^5 times. The histogram in Figure 6 excludes the results for binary networks, since all those conflicts were reported at depth 0.

The runtime comparison between **saucy** 3.0 and **saucy** 2.1 is depicted in Figure 7. For the families of `mz-aug` and `mz-aug2`, we observed an exponential speedup when our proposed refinement procedure was invoked. Of the 49 instances in these two families, **saucy** 3.0 solved all in less than a second, while **saucy** 2.1 failed to process 39 within the time-out limit. For the `mz` and `cmz` families, **saucy** 2.1 and 3.0 had comparable runtimes. The instances of `router`, `roadnet`, and `binnet` did not experience much change either. For the `circuit` family, the results were comparable, except for one benchmark that was solved by **saucy** 3.0 in a second but remained unsolved in **saucy** 2.1. Interestingly enough, we did not observe any major improvement in the runtimes of the SAT 11 CNF benchmarks, although conflict reduction of up to 2.9 million was reported for some of those instances. Our further analysis revealed that high reduction in the number of conflicts was reported for instances that timed out in both **saucy** 3.0 and **saucy** 2.1, and the reduction in the remaining instances was not significant enough to reflect a major improvement in runtimes. Note that the runtimes reported in Figure 7 match with the number of conflicts reported in Figure 5. In fact, fewer conflicts generally led to better runtimes.

In order to evaluate the performance of **saucy** 3.0 versus state-of-the-art graph automorphism tools, we ran **bliss** (version 0.72, available at [5]) on all the 1445 benchmarks listed in Table 1, and compared its runtimes to those obtained from **saucy** 3.0. This comparison is shown in Figure 8. Of the four Miyazaki graph families, **bliss** showed difficulties in processing the instances of `cmz` (took up to 856 seconds to complete all those instances), but processed the

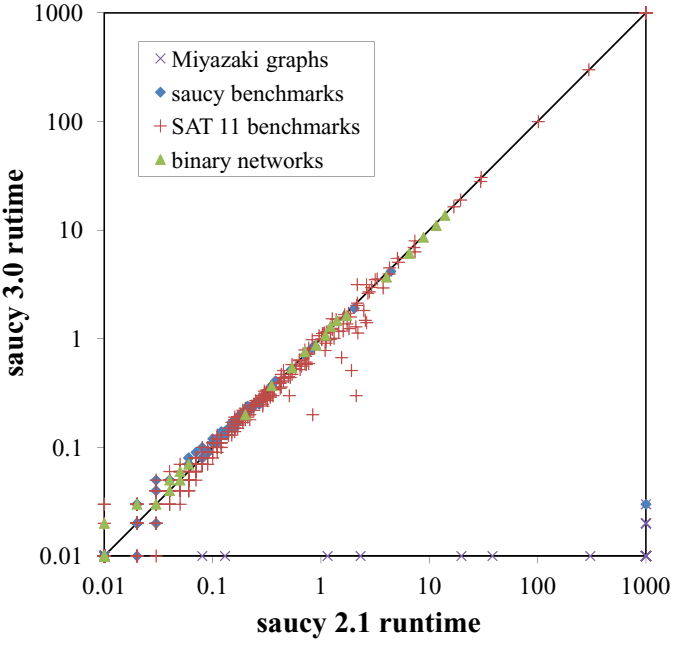


Fig. 7. Runtime of **saucy** 3.0 versus **saucy** 2.1 (timeout is 1000 seconds)

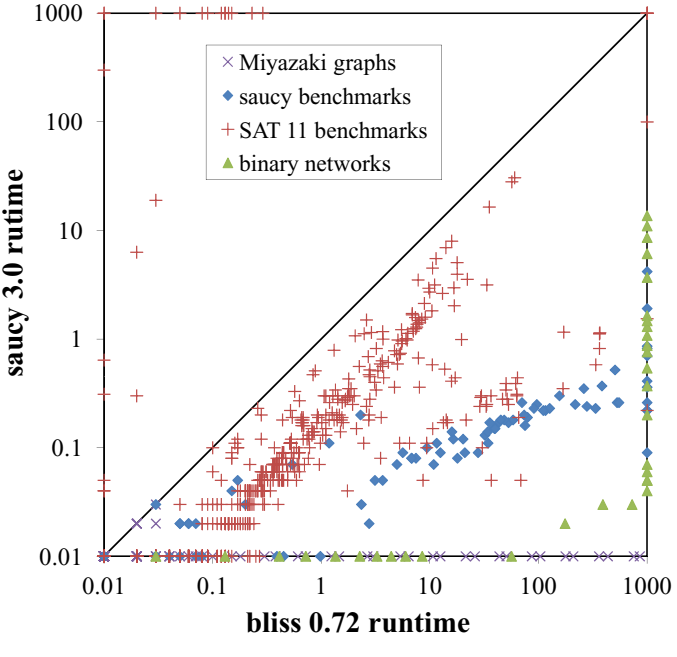


Fig. 8. Runtime of **saucy** 3.0 versus **bliss** 0.72 (timeout is 1000 seconds)

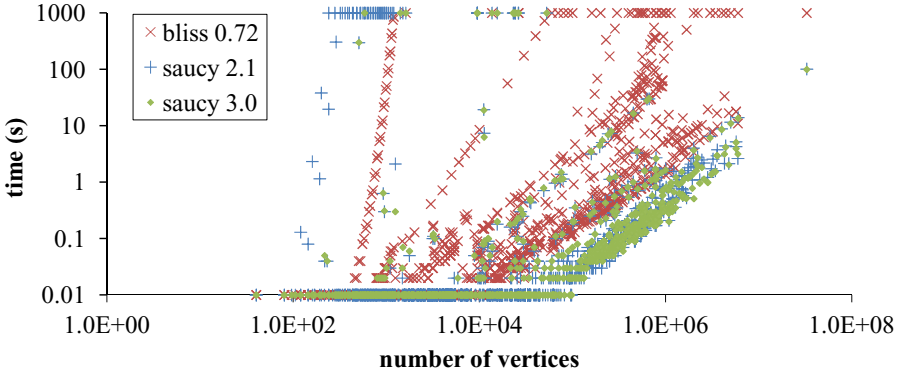


Fig. 9. Runtimes of **saucy** 3.0, **saucy** 2.1, and **bliss** 0.72 as a function of graph size

remaining three families in less than a second. In contrast, **saucy** solved all Miyazaki graphs in less than a second. Furthermore, **bliss** timed out on 8 and 3 out of 33 and 56 instances of the `circuit` and `roadnet` families, respectively, but solved the remaining instances of those two families and all 3 instances of `router` in 550 seconds. This was while **saucy** solved all the 92 instances of these three families in 5 seconds (processed 90 in less than a second). For the CNF benchmarks, **saucy** and **bliss** showed mixed results. Of the 600 crafted and application instances, **bliss** failed to process 4 crafted and 3 application instances, whereas, **saucy** failed to process 17 crafted instances, but solved all application instances. The 4 crafted benchmarks that were unsolved by **bliss** were also unsolved by **saucy**. This means that **bliss** solved 13 crafted instances that **saucy** failed to process, and **saucy** solved 3 application instances that **bliss** did not solve. Of the remaining crafted and application benchmarks, **bliss** solved 541 in less than 10 seconds, and 52 in 366 seconds, while **saucy** solved 577 in less than 10 seconds, and 6 in 300 seconds. Both **saucy** and **bliss** solved all random benchmarks in less than a second. Overall, the results in Figure 8 indicate that **saucy** outperformed **bliss** on the majority of SAT 11 benchmarks. For binary networks, **saucy** consistently produced better results. Specifically, **saucy** solved all 33 instances of `binnet` in 14 seconds (the largest runtime was 13.67 seconds which was reported for the largest instance of this family with 6×10^6 vertices), but **bliss** timed out on 19, and solved the remaining in 727 seconds.

As part of our study, we also ran **nishe** 0.1 [21] on all the graph benchmarks in our suite, and compared its results to **saucy** 3.0. In general, we observed that the runtimes of **nishe** and **saucy** were comparable for the Miyazaki graphs. For the remaining benchmarks, however, **nishe** exhibited poor performance compared to **saucy** and **bliss**. In particular, it failed to process (either timed out or had a segmentation fault) 59 out of 92 **saucy** benchmarks, 950 out of 1200 CNF instances, and 24 out of 33 binary networks.

Figure 9 shows the runtimes of **saucy** 3.0, **saucy** 2.1, and **bliss** 0.72 as a function of graph size for all the 1445 benchmarks listed in Table 1. As this

figure suggests, the smaller instances seem to be more challenging for **saucy**. This is particularly not true of **bliss**, as **bliss** tends to produce larger runtimes for larger instances. The smallest instance that **saucy** 3.0 timed out on had 583 vertices, and the largest had 52,786 vertices, while these numbers were respectively reported to be 1,620 and 33 million for **bliss** 0.72. Of the 446 benchmarks with more than 52,786 vertices, **saucy** 3.0 solved 389 in less than a second, and processed the rest in 100 seconds, while **bliss** 0.72 solved 213 in less than a second, took up to 550 seconds to process 200, and timed out on 33. On the other hand, of the 999 benchmarks that had less than 52,786 vertices, **saucy** 3.0 solved 979 in less than a second, timed out on 17, and took up to 550 seconds to process the rest, whereas, **bliss** 0.72 processed 946 in less than a second, timed out on 4, and processed the remaining in 856 seconds. To investigate the reason why **saucy** 3.0 did not perform as expected on relatively small instances, we examined the effect of different decision heuristics on the 17 benchmarks that **saucy** failed to process. Interestingly, 4 out of those 17 benchmarks were solved in less than a second with an alternative decision heuristic. Of those 4, one was reported to be unsolved by **bliss** 0.72. These results suggest that branching decisions play a crucial role in minimizing the time for automorphism search. We plan to pursue the effect of decision heuristics in our future research.

7 Conclusions

In this work, we have advanced the state of the art in algorithms for solving graph automorphism, which finds applications in many fields. Our technique takes advantage of a unique feature in the **saucy** algorithm — the representation of partial permutations (search nodes) in terms of ordered partition pairs. Previously, these partitions were refined one at a time, but we have now developed *simultaneous partition refinement*, which allows **saucy** to anticipate possible future conflicts and prune the search tree early. This optimization significantly improves runtime on several benchmark families, including the ones suggested by Miyazaki [20] for further study because **nauty** provably requires exponential time on these benchmarks. Our empirical comparisons show that our implementation **saucy** 3.0 outperforms the competition on most available benchmarks. Our ongoing work is focused on several benchmarks where **saucy** 3.0 is outperformed by **bliss** 0.72. Preliminary analysis suggests that these benchmarks tend to be small, which may be due to subtle inefficiencies in our implementation rather than asymptotic bottlenecks. We hope that our future research will shed additional light on this.

References

1. ISPD (2005), http://archive.sigda.org/ispd2005/contest.htm
2. Aloul, F.A., Markov, I.L., Sakallah, K.A.: Shatter: Efficient symmetry-breaking for boolean satisfiability. In: Proc. 40th IEEE/ACM Design Automation Conference (DAC), Anaheim, California, pp. 836–839 (2003)

3. Aloul, F.A., Ramani, A., Markov, I.L., Sakallah, K.A.: Solving difficult sat instances in the presence of symmetry. In: Proc. 39th IEEE/ACM Design Automation Conference (DAC), New Orleans, Louisiana, pp. 731–736 (2002)
4. binary networks, https://sites.google.com/site/santofortunato/inthepress2
5. bliss 0.72 (2011), http://www.tcs.hut.fi/Software/bliss/bliss-0.72.zip
6. U. S. Census Bureau, http://www.census.gov/geo/www/tiger/tigerua/ua_tgr2k.html.
7. Cheswick, B., Burch, H., Branigan, S.: Mapping and visualizing the internet. In: USENIX Annual Technical Conference, pp. 1–13 (2000)
8. SAT Competition, http://www.satcompetition.org
9. Darga, P.T., Liffiton, M.H., Sakallah, K.A., Markov, I.L.: Exploiting structure in symmetry detection for CNF. In: Proc. 41st IEEE/ACM Design Automation Conference (DAC), San Diego, California, pp. 530–534 (2004)
10. Darga, P.T., Sakallah, K.A., Markov, I.L.: Faster symmetry discovery using sparsity of symmetries. In: Proc. 45th IEEE/ACM Design Automation Conference (DAC), Anaheim, California, pp. 149–154 (2008)
11. Fraleigh, J.B.: A First Course in Abstract Algebra, 6th edn. Addison Wesley Longman, Reading (2000)
12. Govindan, R., Tangmunarunkit, H.: Heuristics for internet map discovery. In: IEEE INFOCOM, pp. 1371–1380 (2000)
13. Junttila, T., Kaski, P.: Engineering an efficient canonical labeling tool for large and sparse graphs. In: Ninth Workshop on Algorithm Engineering and Experiments (ALENEX 2007), New Orleans, LA (2007)
14. Junttila, T., Kaski, P.: Conflict Propagation and Component Recursion for Canonical Labeling. In: Marchetti-Spaccamela, A., Segal, M. (eds.) TAPAS 2011. LNCS, vol. 6595, pp. 151–162. Springer, Heidelberg (2011)
15. Kaski, P.: http://www.tcs.hut.fi/Software/bliss/benchmarks/index.shtml
16. Katebi, H., Sakallah, K.A., Markov, I.L.: Symmetry and satisfiability: An update. In: Proc. Satisfiability Symposium (SAT), Edinburgh, Scotland (2010)
17. Lancichinetti, A., Fortunato, S.: Benchmarks for testing community detection algorithms on directed and weighted graphs with overlapping communities. Phys. Rev. E 80, 016118 (2009)
18. McKay, B.D.: nauty user's guide (version 2.2), http://cs.anu.edu.au/~bdm/nauty/nug.pdf
19. Brendan, D.: McKay. Practical graph isomorphism. Congressus Numerantium 30, 45–87 (1981)
20. Miyazaki, T.: The complexity of McKay's canonical labeling algorithm, p. 239. Amer. Mathematical Society (1997)
21. nishe 0.1., http://gregtener.com/media/upload/nishe-0.1.tar.bz2
22. Tener, G., Deo, N.: Efficient isomorphism of miyazaki graphs. In: 39th Southeastern International Conference on Combinatorics, Graph Theory, and Computing, Boca Raton, FL (2008)
23. Velev, M.N., Bryant, R.E.: Effective use of boolean satisfiability procedures in the formal verification of superscalar and vliw microprocessors. In: Proc. Design Automation Conference (DAC), New Orleans, Louisiana, pp. 226–231 (2001)

Confluence of Non-Left-Linear TRSs via Relative Termination*

Dominik Klein and Nao Hirokawa

School of Information Science
Japan Advanced Institute of Science and Technology, Japan
{dominik.klein,hirokawa}@jaist.ac.jp

Abstract. We present a confluence criterion for term rewrite systems by relaxing termination requirements of Knuth and Bendix' confluence criterion, using joinability of extended critical pairs. Because computation of extended critical pairs requires equational unification, which is undecidable, we give a sufficient condition for testing joinability automatically.

1 Introduction

Applications in various domains [16,20,26], resulted in an interest in proving confluence of term rewrite systems (TRSs) *automatically* [3,10,27,28]. Knuth and Bendix [16] showed that confluence of terminating TRSs is decidable by testing joinability of critical pairs, which are induced by *overlaps*. In the case of non-termination, several powerful techniques have been developed for proving confluence of left-linear systems [10,23,24]. Still, proving confluence of both non-left-linear and non-terminating TRSs remains challenging.

Results that tackle this setting can be roughly classified into three categories: First, by generalizing the notion of overlaps, one can formulate *direct criteria* that guarantee confluence [8,9]. The second approach is to *decompose* a TRS into smaller ones, show confluence of each of them by existing criteria, and formulate modularity conditions to ensure that the union remains confluent [1,19,22]. The third approach is to generalize Knuth and Bendix' confluence criterion by relaxing termination requirements to *relative termination*. A famous result here is Jouannaud and Kirchner's criterion for the Church-Rosser modulo property [13] based on extended critical pairs. Geser [6] analyzed their proof to derive confluence criteria based only on syntactical critical pairs.

We present a new confluence criterion that also relies on relative termination, and can be applied for non-left-linear TRSs. The criterion requires to check joinability of extended critical pairs, but we show that under certain conditions, joinability can be concluded from joinability of syntactical critical pairs. With it, we are able to prove confluence of several non-terminating, non-left-linear TRSs fully automatically, for which no known criteria exist.

* This work is supported by the Grant-in-Aids for Young Scientists (B) 22700009 and Scientific Research (B) 23300005 of the Japan Society for the Promotion of Science.

N. Bjørner and A. Voronkov (Eds.): LPAR-18, LNCS 7180, pp. 258–273, 2012.

This paper is structured as follows: In Section 2 we recall notions from rewriting, unification, and the decreasing diagram technique, which will be used in proofs later. Our main result is presented in Section 3. Since it requires joinability of uncomputable extended critical pairs, we explain in Section 4 how to automate it, and then report on experiments in Section 5. In Section 6, we compare our criterion with related works, and finally conclude with an outlook on future work in Section 7.

2 Preliminaries

We assume familiarity with the basics of term rewriting ([4,21]).

Term Rewriting. Terms are inductively defined over a set $\mathcal{F}$ of fixed-arity *function symbols*, and a set $\mathcal{V}$ of *variables*. For given term t, the set of variables occurring in t is denoted by $\mathcal{V}\mathsf{ar}(t)$. The set of (variable, function) *positions* in t is denoted by $\mathcal{P}\mathsf{os}(t)$ ($\mathcal{P}\mathsf{os}_{\mathcal{V}}(t), \mathcal{P}\mathsf{os}_{\mathcal{F}}(t)$). Here positions are expressed by sequences on natural numbers, and the *root* position ε is the empty sequence. Given positions p, q, and o, we write $p\backslash q$ for o if $p = qo$. We write $\rhd$ for the proper superterm relation. The domain $\mathcal{D}\mathsf{om}(\sigma)$ of a substitution σ is the set $\{x \in \mathcal{V} \mid x \neq x\sigma\}$. A rewrite rule $\ell \to r$ is a pair (ℓ, r) of terms with $\mathcal{V}\mathsf{ar}(r) \subseteq \mathcal{V}\mathsf{ar}(\ell)$ and $\ell \notin \mathcal{V}$. A TRS is a collection of rewrite rules. A rewrite rule is *left-linear* if no variable occurs more than once in ℓ. Likewise, a TRS is left-linear, if all of its rules are. An *extended rewrite rule* is a pair (ℓ, r) of terms with $\ell \notin \mathcal{V}$, and an *extended TRS* (*eTRS*) is a set of extended rewrite rules. A rewrite step of $\mathcal{R}$ at position p is denoted by $\xrightarrow{p}_{\mathcal{R}}$. We write $\downarrow_{\mathcal{R}}$ for the *join* relation $\to_{\mathcal{R}}^* \cdot {}_{\mathcal{R}}^*\!\leftarrow$. We write $\to_1/\to_2$ for $\to_2^* \cdot \to_1 \cdot \to_2^*$, and $\to_{\mathcal{R}/\mathcal{S}}$ for $\to_{\mathcal{R}}/\to_{\mathcal{S}}$. $\mathcal{R}$ is *relatively terminating* over a TRS $\mathcal{S}$ or $\mathcal{R}/\mathcal{S}$ is terminating, if $\to_{\mathcal{R}/\mathcal{S}}$ is so.

Unification. We briefly recapitulate some notions from unification theory. An equality $s \approx t$ is the ordered pair (s, t) of terms. Let $\mathcal{E}$ and $\mathcal{S}$ be sets of equalities, and X the set of all variables in $\mathcal{E}$. Given a substitution σ, we write $\mathcal{E}\sigma$ for $\{s\sigma \approx t\sigma | s \approx t \in \mathcal{E}\}$. An *$\mathcal{S}$-unifier* of $\mathcal{E}$ is a substitution σ such that $\mathcal{E}\sigma \subseteq \leftrightarrow_{\mathcal{S}}^*$. A substitution σ is *more general* than a substitution σ' on X ($\sigma \lesssim_{\mathcal{S}}^X \sigma'$), if there exists a substitution τ such that $x\sigma' \leftrightarrow_{\mathcal{S}}^* x\sigma\tau$ for all $x \in X$. Let $\mathcal{U}$ be a set of $\mathcal{S}$-unifiers of $\mathcal{E}$. We say that $\mathcal{U}$ is *complete* if for every $\mathcal{S}$-unifier of $\mathcal{E}$ there is a more general element in $\mathcal{U}$. If in addition all elements in $\mathcal{U}$ are minimal with respect to $\lesssim_{\mathcal{S}}^X$, we call $\mathcal{U}$ *minimal complete*. A substitution σ is an *$\mathcal{S}$-most general unifier* (*$\mathcal{S}$-mgu*) of $\mathcal{E}$, if $\{\sigma\}$ is a minimal complete set of $\mathcal{S}$-unifiers of $\mathcal{E}$. In the special case of $\mathcal{S} = \varnothing$, we simply speak of (syntactic) unification, unifiers and mgu's. A set of equalities $\mathcal{E} = \{x_1 \approx t_1, \ldots, x_n \approx t_n\}$ is in *solved form*, if x_i are pairwise distinct variables, and no x_i occurs in t_i. For $\mathcal{E}$ in solved form, we write $\overrightarrow{\mathcal{E}}$ for the induced substitution $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$. Note that in general, $\mathcal{S}$-unifiability does not ensure presence of an $\mathcal{S}$-mgu, except for $\mathcal{S} = \varnothing$.

Critical Pairs. Conditions for confluence are often based on the notion of overlaps and critical pairs. Let $\mathcal{R}_1, \mathcal{R}_2, \mathcal{S}$ be eTRSs. An $\mathcal{S}$-overlap $(\ell_1 \to r_1, p, \ell_2 \to r_2)_\sigma$ of $\mathcal{R}_1$ on $\mathcal{R}_2$ consists of a variant $\ell_1 \to r_1$ of a rule in $\mathcal{R}_1$ and $\ell_2 \to r_2$ of a rule in $\mathcal{R}_2$, a position $p \in \mathcal{P}\mathsf{os}_{\mathcal{F}}(\ell_2)$ and a substitution σ, such that $\ell_1\sigma \leftrightarrow^*_{\mathcal{S}} \ell_2|_p\sigma$. If $p = \varepsilon$, then $\ell_1 \to r_1$ and $\ell_2 \to r_2$ may not be variants of each other. The pair $(\ell_2\sigma[r_1\sigma]_p, r_2\sigma)$ induced from the overlap is an *$\mathcal{S}$-extended critical pair* (or simply *$\mathcal{S}$-critical pair*) of $\ell_1 \to r_1$ and $\ell_2 \to r_2$ at p, written $\ell_2\sigma[r_1\sigma]_p \; {}_{\mathcal{R}_1}\!\leftarrow_{\mathcal{S}}\!\!\circ\!\!\to_{\mathcal{R}_2} r_2\sigma$. We write ${}_{\mathcal{R}_1}\!\leftarrow_{\mathcal{S}}\!\!\circ\!\!\to_{\mathcal{R}_2}$ for ${}_{\mathcal{R}_1}\!\leftarrow_{\mathcal{S}}\!\!\circ\!\!\to_{\mathcal{R}_2} \cup \; {}_{\mathcal{R}_2}\!\leftarrow_{\mathcal{S}}\!\!\circ\!\!\to_{\mathcal{R}_1}$. We remark that our definition of ($\mathcal{S}$-)critical pairs includes pairs originating from non-minimal unifiers, which are usually excluded from the definition to guarantee finiteness of critical pairs.

Let $\mathsf{REN}(t)$ denote a linear term resulting from replacing in t each variable occurrence by a fresh variable. We write $\widehat{\mathcal{R}}$ for the eTRS $\{\mathsf{REN}(\ell) \to r \mid \ell \to r \in \mathcal{R}\}$. A TRS $\mathcal{S}$ is *strongly non-overlapping* on $\mathcal{R}$ if $\widehat{\mathcal{S}}$ has no overlaps on $\widehat{\mathcal{R}}$. We write $\mathsf{SNO}(\mathcal{R}, \mathcal{S})$ if both $\mathcal{S}$ is strongly non-overlapping on $\mathcal{R}$, and $\mathcal{R}$ is strongly non-overlapping on $\mathcal{S}$. Left-linear TRSs without critical pairs are called *orthogonal*. Orthogonal TRSs are confluent. Moreover, Knuth and Bendix' criterion [16] states that ${}_{\mathcal{R}}\!\leftarrow_{\varnothing}\!\!\circ\!\!\to_{\mathcal{R}} \subseteq \downarrow_{\mathcal{R}}$ implies confluence of a terminating TRS $\mathcal{R}$.

Decreasing Diagrams. Van Oostrom showed a powerful confluence criterion for abstract rewrite systems (ARSs), called the *decreasing diagram* technique [25]. Let $\mathcal{A} = (A, \langle \to_\alpha \rangle_{\alpha \in I})$ be an ARS and $>$ a proper order on I. For every $\alpha \in I$ we write $\overset{\vee}{\to}_\alpha$ for $\{\to_\beta \mid \beta \in I \text{ and } \beta < \alpha\}$, and write $\overset{\vee}{\to}^*_\alpha$ for $(\overset{\vee}{\to}_\alpha)^*$. The union of $\overset{\vee}{\to}_\alpha$ and ${}_\alpha\overset{\vee}{\leftarrow}$ is denoted by $\overset{\vee}{\leftrightarrow}_\alpha$. For $\alpha, \beta \in I$, the union of $\overset{\vee}{\to}_\alpha$ and $\overset{\vee}{\to}_\beta$ is written as $\overset{\vee}{\to}_{\alpha\beta}$. Two labels α and β are *decreasing* with respect to $>$ if

$$ {}_\alpha\!\leftarrow \cdot \to_\beta \;\subseteq\; \overset{\vee}{\leftrightarrow}{}^*_\alpha \cdot \to^{=}_\beta \cdot \overset{\vee}{\leftrightarrow}{}^*_{\alpha\beta} \cdot {}^{=}_\alpha\!\leftarrow \cdot \; {}^*_\beta\overset{\vee}{\leftrightarrow} $$

An ARS $\mathcal{A} = (A, \langle \to_\alpha \rangle_{\alpha \in I})$ is *decreasing* if there exists a well founded order $>$ such that all two labels in I are decreasing with respect to $>$.

Theorem 1 ([25]). *A decreasing ARS is confluent.* □

3 Confluence Criterion

First we state our main theorem, which is a proper generalization (when $\mathcal{S} \neq \varnothing$) of Knuth and Bendix' confluence criterion.

Theorem 2. *Suppose that $\mathcal{S}$ is confluent, $\mathcal{R}/\mathcal{S}$ is terminating, and $\mathsf{SNO}(\mathcal{R}, \mathcal{S})$. The union $\mathcal{R} \cup \mathcal{S}$ of the TRSs is confluent if and only if ${}_{\mathcal{R}}\!\leftarrow_{\mathcal{S}}\!\!\circ\!\!\to_{\mathcal{R}} \subseteq \downarrow_{\mathcal{R}\cup\mathcal{S}}$.*

In the rest of this section we first prove our main theorem, and afterwards give examples of its application. Let $\mathcal{R}$ and $\mathcal{S}$ be TRSs. We introduce an *intermediate* relation $\to\!\!\!\!\triangleright$, such that $\to_{\mathcal{R}\cup\mathcal{S}} \subseteq \to\!\!\!\!\triangleright \subseteq \to^*_{\mathcal{R}\cup\mathcal{S}}$. Confluence of this intermediate

relation readily implies confluence of $\mathcal{R} \cup \mathcal{S}$. The relation $\twoheadrightarrow$ is defined as the union of $\to_{\mathcal{R}_\mathcal{S}}$ and $\to_\mathcal{S}^*$, where $\mathcal{R}_\mathcal{S}$ is the TRS

$$\{\ell'\sigma \to r\tau \mid \ell'\rho \to r \in \mathcal{R} \text{ and } \sigma \to_\mathcal{S}^* \rho\tau \text{ for some substitution } \rho \text{ on } \mathcal{V}\ \}$$

In the above set $\sigma \to_\mathcal{S}^* \tau$ means that $x\sigma \to_\mathcal{S}^* x\tau$ for all variables x. It is important to note that in the definition linearity of ℓ' can be assumed without loss of generality, and that the inclusions $\to_\mathcal{R} \subseteq \to_{\mathcal{R}_\mathcal{S}} \subseteq \to_\mathcal{S}^* \cdot \to_\mathcal{R}$ hold.

We show confluence of $\twoheadrightarrow$ by the decreasing diagram technique with the predecessor labeling [25]: We write $b \twoheadrightarrow_a c$ if $a \twoheadrightarrow^* b \twoheadrightarrow c$. Labels are compared with respect to $\to^+_{\mathcal{R}/\mathcal{S}}$, denoted by $>$. Since termination of $\mathcal{R}/\mathcal{S}$ is presupposed in the theorem, the relation $>$ forms a well-founded order. The next lemma states a property of rewriting in substitutions.

Lemma 3. *If $t\sigma \xrightarrow{p}_\mathcal{R} u$ and $p \notin \mathcal{P}\mathsf{os}_\mathcal{F}(t)$ then $u \to_\mathcal{R}^* t\tau$ for some τ with $\sigma \to_\mathcal{R}^= \tau$.*

Proof. Suppose $t\sigma \xrightarrow{p}_\mathcal{R} u$ and $p \notin \mathcal{P}\mathsf{os}_\mathcal{F}(t)$. Then there exists a variable position $q \in \mathcal{P}\mathsf{os}_\mathcal{V}(t)$ with $q \leqslant p$ and $u = (t\sigma)[u|_q]_q$. Let Q be the set of all variable occurrences of $t|_q$ in t. Since $u|_{q'} \to_\mathcal{R} u|_q$ holds for all $q' \in Q \setminus \{q\}$, we have $u \to_\mathcal{R} (t\sigma)[u|_q]_{q' \in Q \setminus \{q\}}$. The latter term is identical to $(t\sigma)[u|_q]_{q' \in Q}$. We define the substitution τ as follows:

$$\tau(x) = \begin{cases} u|_q & \text{if } x = t|_q \\ x\sigma & \text{otherwise} \end{cases}$$

One can verify $\sigma \to_\mathcal{R}^= \tau$ and $(t\sigma)[u|_q]_{q' \in Q} = t\tau$. Hence $u \to_\mathcal{R}^* t\tau$. □

We analyze peaks of the form $\twoheadleftarrow \cdot \twoheadrightarrow$. According to the definition of $\twoheadrightarrow$, they fall into the three cases: (a) ${}^*_\mathcal{S}\!\leftarrow \cdot \to_\mathcal{S}^*$, (b) ${}_{\mathcal{R}_\mathcal{S}}\!\leftarrow \cdot \to_\mathcal{S}^*$, and (c) ${}_{\mathcal{R}_\mathcal{S}}\!\leftarrow \cdot \to_{\mathcal{R}_\mathcal{S}}$. For case (a) we can apply confluence of $\mathcal{S}$ to show decreasingness of the peak. The remaining cases are more complicated. We start with a localized version of (b). In the next Lemmata 4, 6, and 7 we assume $\mathsf{SNO}(\mathcal{R}, \mathcal{S})$ and confluence of $\mathcal{S}$.

Lemma 4. *If $t\ {}_{\mathcal{R}_\mathcal{S}}\!\leftarrow s \to_\mathcal{S} u$ then $t \to_{\mathcal{R}\cup\mathcal{S}}^* \cdot\ {}^*_{\mathcal{R}_\mathcal{S}}\!\leftarrow u$.*

Proof. We perform induction on s. Suppose $t\ {}_{\mathcal{R}_\mathcal{S}}\!\xleftarrow{p} s \xrightarrow{q}_\mathcal{S} u$. By the definition of $\mathcal{R}_\mathcal{S}$ we may assume $\ell_1\rho \to r_1 \in \mathcal{R}$ for some linear term ℓ_1 and $\rho : \mathcal{V} \to \mathcal{V}$, $s|_p = \ell_1\sigma$, $t|_p = r_1\tau$, and $\sigma \to_\mathcal{S}^* \rho\tau$, as well as $\ell_2 \to r_2 \in \mathcal{S}$, $s|_q = \ell_2\mu$, and $u|_q = r_2\mu$. Due to $\mathsf{SNO}(\mathcal{R}, \mathcal{S})$, neither $p \backslash q \in \mathcal{P}\mathsf{os}_\mathcal{F}(\ell_2)$ nor $q \backslash p \in \mathcal{P}\mathsf{os}_\mathcal{F}(\ell_1)$ holds. We distinguish several cases concerning the relation of p and q.

– Suppose $p = \varepsilon$. Then there is a variable position q_1 of x_1 in ℓ_1 with $q_1 \leqslant q$. Since $x_1\rho\tau\ {}^*_\mathcal{S}\!\leftarrow x_1\sigma \to_\mathcal{S} u|_{q_1}$ holds, we have $x_1\rho\tau \to_\mathcal{S}^* v\ {}^*_\mathcal{S}\!\leftarrow u|_{q_1}$ for some v by confluence of $\mathcal{S}$. We define the substitutions μ_1 and ν as follows:

$$\mu_1(x) = \begin{cases} u|_{q_1} & \text{if } x = x_1 \\ x\sigma & \text{otherwise} \end{cases} \qquad \nu(x) = \begin{cases} v & \text{if } x = x_1\rho \\ x\tau & \text{otherwise} \end{cases}$$

We have $\tau \to_\mathcal{S}^* \nu$, and also $u = \ell_1\mu_1$ by linearity of ℓ_1. Moreover, $\mu_1 \to_\mathcal{S}^* \rho\nu$ because $x\mu_1 \to_\mathcal{S}^* v = x_1\rho\nu = x\rho\nu$ if $x = x_1$, and $x\mu = x\sigma \to_\mathcal{S}^* x\nu$ otherwise. Therefore, we obtain $t = r_1\tau \to_\mathcal{S}^* r_1\nu\ {}_{\mathcal{R}_\mathcal{S}}\!\leftarrow \ell_1\mu_1 = u$.

- Suppose $q = \varepsilon$. We may presume $\mathcal{V}\mathsf{ar}(\ell_1) \cap \mathcal{V}\mathsf{ar}(\ell_2) = \varnothing$, and thus $\sigma = \mu$ can be assumed. Since $\ell_2\sigma \to_{\mathcal{R}_\mathcal{S}} t$ holds, by Lemma 3 we obtain $t \to^*_{\mathcal{R}_\mathcal{S}} \ell_2\nu$ for some ν with $\sigma \to^=_{\mathcal{R}_\mathcal{S}} \nu$. Thus, $t \to^*_{\mathcal{R}_\mathcal{S}} \ell_2\nu \to_\mathcal{S} r_2\nu \; {}^*_{\mathcal{R}_\mathcal{S}}\!\leftarrow r_2\sigma = r_2\mu = u$.
- If $p = ip'$ and $q = jq'$ for some $i, j \in \mathbb{N}$ with $i \neq j$, one can easily verify $t \overset{q}{\to}_\mathcal{S} \cdot \mathcal{R}_\mathcal{S} \overset{p}{\leftarrow} u$.
- Otherwise, $p = ip'$ and $q = iq'$ for some $i \in \mathbb{N}$. Since $t|_i \; \mathcal{R}_\mathcal{S}\!\leftarrow s|_i \to_\mathcal{S} u|_i$ holds, the induction hypothesis yields $t|_i \to^*_{\mathcal{R}\cup\mathcal{S}} \cdot \; {}^*_{\mathcal{R}_\mathcal{S}}\!\leftarrow u|_i$. Therefore $t \to^*_{\mathcal{R}\cup\mathcal{S}} \cdot \; {}^*_{\mathcal{R}_\mathcal{S}}\!\leftarrow u$. □

In order to handle peaks of shape $\mathcal{R}_\mathcal{S}\!\leftarrow \cdot \to^*_\mathcal{S}$ we show an auxiliary lemma for ARSs. In the next lemma $\to$ stands for $\to_1 \cup \to_2$ and $>$ for $(\to_1/\to_2)^+$, and we write $b \to_a c$ if $a \to^* b \to c$. We will freely use the next two facts: (1) for all a, b, c with $a > b$, we have that $b \overset{\vee}{\leftrightarrow}{}^*_a \cdot \to^* c$ implies $b \overset{\vee}{\leftrightarrow}{}^*_a c$, and (2) $b \to^=_1 \cdot \overset{\vee}{\to}{}^*_a c$ whenever $a \to^* b \to^*_1 c$.

Lemma 5. *Let* ${}_1\!\leftarrow \cdot \to_2 \subseteq \to^* \cdot \; {}^*_1\!\leftarrow$. *If* $b \; {}_1\!\leftarrow a \to^*_2 c$ *then* $b \overset{\vee}{\leftrightarrow}{}^*_a \cdot \; {}^=_1\!\leftarrow c$.

Proof. Let $b \; {}_1\!\leftarrow a \to^n_2 c$. We show the claim by induction on n. If $n = 0$ then trivially the claim holds. Otherwise, $a \to^{n-1}_2 d \to_2 c$ for some d. The induction hypothesis yields $b \overset{\vee}{\leftrightarrow}{}^*_a e \; {}^=_1\!\leftarrow d$ for some e. We distinguish two cases.

- If $d = e$ then $b \overset{\vee}{\leftrightarrow}{}^*_a e = d \to_1 c$. Thus $b \overset{\vee}{\leftrightarrow}{}^*_a c$ by (1).
- Suppose $d \to_1 e$. Because we have $e \; {}_1\!\leftarrow d \to_2 c$, by the assumption $e \to^* f \; {}^*_1\!\leftarrow c$ for some f. Since $a \to^*_2 d \to_1 e \to^* f$ holds, we obtain $e \overset{\vee}{\to}{}^*_a f$ by (2). Moreover, $c \to^*_1 f$ implies $c \to^=_1 \cdot \overset{\vee}{\to}{}^*_a f$ by (2). Hence, $b \overset{\vee}{\leftrightarrow}{}^*_a \cdot \; {}^=_1\!\leftarrow c$. □

Lemma 6. *If* $t \; \mathcal{R}_\mathcal{S}\!\leftarrow s \to^*_\mathcal{S} u$ *then* $t \overset{\vee}{\leftarrow\!\rightarrow}{}^*_s \cdot \; {}^=_{\mathcal{R}_\mathcal{S}}\!\leftarrow u$.

Proof. By Lemma 4 we have that $t \; \mathcal{R}_\mathcal{S}\!\leftarrow s \to_\mathcal{S} u$ implies $t \to^*_{\mathcal{R}\cup\mathcal{S}} \cdot \; {}^*_{\mathcal{R}_\mathcal{S}}\!\leftarrow u$. The claim follows by instantiating Lemma 5 with $\to_1$ as $\to_{\mathcal{R}_\mathcal{S}}$ and $\to_2$ as $\to_\mathcal{S}$. □

Lastly, peaks of case (c), of shape $\mathcal{R}_\mathcal{S}\!\leftarrow \cdot \to_{\mathcal{R}_\mathcal{S}}$, are considered.

Lemma 7. *If* $t \; \mathcal{R}_\mathcal{S}\!\leftarrow s \to_{\mathcal{R}_\mathcal{S}} u$ *then* $t \overset{\vee}{\leftarrow\!\rightarrow}{}^*_s u$ *or* $t \to^*_\mathcal{S} \cdot \mathcal{R}\!\leftarrow_\mathcal{S} \infty \to_\mathcal{R} \cdot \; {}^*_\mathcal{S}\!\leftarrow u$.

Proof. We perform induction on s. Suppose $t \; \mathcal{R}_\mathcal{S} \overset{p}{\leftarrow} s \overset{q}{\to}_{\mathcal{R}_\mathcal{S}} u$. By the definition of $\mathcal{R}_\mathcal{S}$ we can assume $\ell_1\rho_1 \to r_1, \ell_2\rho_2 \to r_2 \in \mathcal{R}$ for some linear terms ℓ_1, ℓ_2 and $\rho_1, \rho_2 : \mathcal{V} \to \mathcal{V}$, and

$$s|_p = \ell_1\sigma_1 \qquad t|_p = r_1\tau_1 \qquad \sigma_1 \to^*_\mathcal{S} \rho_1\tau_1$$
$$s|_q = \ell_2\sigma_2 \qquad u|_q = r_2\tau_2 \qquad \sigma_2 \to^*_\mathcal{S} \rho_2\tau_2$$

Except for symmetric cases, the relation of p and q falls into the next four cases:

- Suppose $q = \varepsilon$, and $p \in \mathcal{P}\mathsf{os}_\mathcal{F}(\ell_2)$. We have $\ell_1\rho_1\tau_1 \; {}^*_\mathcal{S}\!\leftarrow s \to^*_\mathcal{S} \ell_2|_p\rho_2\tau_2$. Without loss of generality $\mathcal{V}\mathsf{ar}(\ell_1\rho_1) \cap \mathcal{V}\mathsf{ar}(\ell_2\rho_2) = \varnothing$, and thus we may assume $\tau = \tau_1 \cup \tau_2$ is a well-defined substitution. The substitution τ is an $\mathcal{S}$-unifier of $\ell_1\rho_1$ and $\ell_2\rho_2|_p$. Because $x\sigma_2 \to^*_\mathcal{S} x\rho_2\tau$ holds for all $x \in \mathcal{V}\mathsf{ar}(\ell_2)$,

$$t = (\ell_2\sigma_2)[r_1\tau]_p \to^*_\mathcal{S} (\ell_2\rho_2\tau)[r_1\tau]_p \; \mathcal{R}\!\leftarrow_\mathcal{S} \infty \to_\mathcal{R} r_2\tau = u$$

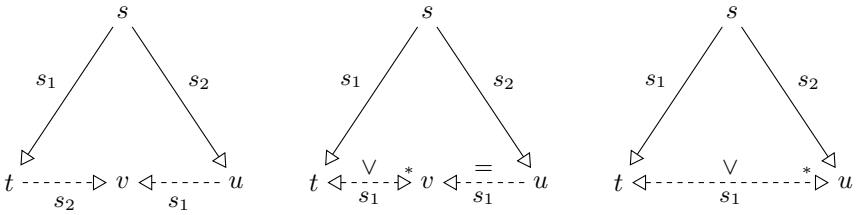


Fig. 1. Decreasingness of $\twoheadrightarrow$

– Suppose $q = \varepsilon$, and $p \notin \mathcal{P}\mathsf{os}_{\mathcal{F}}(\ell_2)$ and p_2 is a variable occurrence of x_2 in ℓ_2 with $p_2 \leqslant p$. Since $t|_{p_2}\ \mathcal{R}_{\mathcal{S}}\leftarrow x_2\sigma_2 \rightarrow^*_{\mathcal{S}} x_2\rho_2\tau_2$, Lemma 6 yields $t|_{p_2} \overset{\vee}{\leftrightarrow}^*_{s|_{p_2}} v\ \overset{=}{\mathcal{R}_{\mathcal{S}}}\leftarrow x_2\rho_2\tau_2$ for some v. Because $s = \ell_2\sigma_2$, $t|_{p_2} \overset{\vee}{\leftrightarrow}^*_{s|_{p_2}} v$, and $\sigma_2 \rightarrow^*_{\mathcal{S}} \rho_2\tau_2$ hold, by closure under contexts of rewrite relations and $>$ we obtain

$$t = (\ell_2\sigma_2)[t|_{p_2}]_{p_2} \overset{\vee}{\leftrightarrow}^*_s (\ell_2\sigma_2)[v]_{p_2} \rightarrow^*_{\mathcal{S}} (\ell_2\rho_2\tau_2)[v]_{p_2}$$

Thus, $t \overset{\vee}{\leftrightarrow}^*_s (\ell_2\rho_2\tau_2)[v]_{p_2}$. Since $x_2\rho_2\tau_2 \overset{=}{\rightarrow}_{\mathcal{R}_{\mathcal{S}}} v$ holds and ℓ_2 is linear,

$$\ell_2\rho_2\tau_2 \overset{=}{\rightarrow}_{\mathcal{R}_{\mathcal{S}}} (\ell_2\rho_2\tau_2)[v]_{p_2}$$

is deduced. Here we distinguish two cases. If $\ell_2\rho_2\tau_2 = (\ell_2\rho_2\tau_2)[v]_{p_2}$, we obtain

$$t \overset{\vee}{\leftrightarrow}^*_s \ell_2\rho_2\tau_2 \rightarrow_{\mathcal{R}} u$$

Otherwise, $\ell_2\rho_2\tau_2 \rightarrow_{\mathcal{R}_{\mathcal{S}}} (\ell_2\rho_2\tau_2)[v]_{p_2}$. Since by Lemma 3 there exists ν with $\tau_2 \overset{=}{\rightarrow}_{\mathcal{R}_{\mathcal{S}}} \nu$ such that $(\ell_2\rho_2\tau_2)[v]_{p_2} \rightarrow^*_{\mathcal{R}_{\mathcal{S}}} \ell_2\rho_2\nu$, finally we obtain

$$t \overset{\vee}{\leftrightarrow}^*_s (\ell_2\rho_2\tau_2)[v]_{p_2} \rightarrow^*_{\mathcal{R}_{\mathcal{S}}} \ell_2\rho_2\nu \rightarrow_{\mathcal{R}} r_2\nu\ \overset{*}{\mathcal{R}_{\mathcal{S}}}\leftarrow r_2\tau_2 = u$$

Because $s > t$ and $s > u$ hold, in both cases $t \overset{\vee}{\leftrightarrow}^*_s u$ is concluded.

– If $p = ip'$ and $q = jq'$ for some $i, j \in \mathbb{N}$ with $i \neq j$, one can easily verify $t \overset{q}{\rightarrow}_{\mathcal{R}_{\mathcal{S}}} \cdot\ _{\mathcal{R}_{\mathcal{S}}}\overset{p}{\leftarrow} u$, which implies $t \overset{\vee}{\leftrightarrow}^*_s u$.
– Otherwise, $p = ip'$ and $q = iq'$ for some $i \in \mathbb{N}$. Since $t|_i\ \mathcal{R}_{\mathcal{S}}\leftarrow s|_i \rightarrow_{\mathcal{R}_{\mathcal{S}}} u|_i$ holds, by induction hypothesis $t|_i \overset{\vee}{\leftrightarrow}^*_{s|_i} u|_i$ or $t|_i \rightarrow^*_{\mathcal{S}} \cdot\ \mathcal{R}\leftarrow_{\mathcal{S}}\infty\rightarrow_{\mathcal{R}} \cdot\ \overset{*}{\mathcal{S}}\leftarrow u|_i$ is deduced. Thus, $t \overset{\vee}{\leftrightarrow}^*_s u$ or $t \rightarrow^*_{\mathcal{S}} \cdot\ \mathcal{R}\leftarrow_{\mathcal{S}}\infty\rightarrow_{\mathcal{R}} \cdot\ \overset{*}{\mathcal{S}}\leftarrow u$ is concluded. □

Now we are ready to prove the main theorem.

Proof (of Theorem 2). Suppose that $\mathcal{S}$ is confluent, $\mathcal{R}/\mathcal{S}$ is terminating, and $\mathsf{SNO}(\mathcal{R}, \mathcal{S})$. We show that $\mathcal{R} \cup \mathcal{S}$ is confluent if and only if $\mathcal{R}\leftarrow_{\mathcal{S}}\infty\rightarrow_{\mathcal{R}} \subseteq \downarrow_{\mathcal{R}\cup\mathcal{S}}$. Since the "only if"-direction is trivial, we only show the "if"-direction. Assume $\mathcal{R}\leftarrow_{\mathcal{S}}\infty\rightarrow_{\mathcal{R}} \subseteq \downarrow_{\mathcal{R}\cup\mathcal{S}}$. Because confluence of $\twoheadrightarrow$ implies confluence of $\mathcal{R}\cup\mathcal{S}$, according to Theorem 1, it is enough to show decreasingness of $\twoheadrightarrow$. Let $t\ _{s_1}\leftarrow s \twoheadrightarrow_{s_2} u$. As mentioned, following the definition of $\twoheadrightarrow$, we distinguish three cases.

(a) If $t \,{}^{*}_{\mathcal{S}}\!\leftarrow s \rightarrow^{*}_{\mathcal{S}} u$ then $t \rightarrow^{*}_{\mathcal{S}} v \,{}^{*}_{\mathcal{S}}\!\leftarrow u$ for some v by confluence of $\mathcal{S}$.
(b) If $t \,{}_{\mathcal{R}_\mathcal{S}}\!\leftarrow s \rightarrow^{*}_{\mathcal{S}} u$ then $t \overset{*}{\underset{\vee}{\leftrightarrow}}_{s} v \,{}^{=}_{\mathcal{R}_\mathcal{S}}\!\leftarrow u$ for some v by Lemma 6.
(c) If $t \,{}_{\mathcal{R}_\mathcal{S}}\!\leftarrow s \rightarrow_{\mathcal{R}_\mathcal{S}} u$ then $t \overset{*}{\underset{\vee}{\leftrightarrow}}_{s} u$ for some v by Lemma 7 and joinability of $\mathcal{S}$-critical pairs.

In all cases decreasingness is established, as seen in Figure 1. □

The next examples illustrate Theorem 2. Note that no existing powerful tool can prove their confluence automatically (see Section 5).

Example 8. Consider the TRS

$$1\colon\ \mathsf{f}(x,x) \to (x+x)+x \qquad\qquad 2\colon\ x+y \to y+x$$

Take $\mathcal{R} = \{1\}$ and $\mathcal{S} = \{2\}$. One can easily verify $\mathsf{SNO}(\mathcal{R},\mathcal{S})$. Termination of $\mathcal{R}/\mathcal{S}$ can be established using a termination tool such as $\mathsf{T_TT_2}$ v1.06 [17][1], and confluence of $\mathcal{S}$ follows from orthogonality. Because of ${}_{\mathcal{R}}\!\leftarrow_{\mathcal{S}}\!\circ\!\rightarrow_{\mathcal{R}} = \varnothing \subseteq \downarrow_{\mathcal{R}\cup\mathcal{S}}$, we conclude confluence by Theorem 2.

Example 9. Consider the TRS

$$1\colon\ \mathsf{f}(x,x) \to \mathsf{s}(\mathsf{s}(x)) \qquad\qquad 2\colon\ \infty \to \mathsf{s}(\infty)$$

Take $\mathcal{R} = \{1\}$ and $\mathcal{S} = \{2\}$. As in Example 8, one can easily verify the conditions of Theorem 2, including ${}_{\mathcal{R}}\!\leftarrow_{\mathcal{S}}\!\circ\!\rightarrow_{\mathcal{R}} = \varnothing \subseteq \downarrow_{\mathcal{R}\cup\mathcal{S}}$. Hence the TRS is confluent.

Example 10. Consider the TRS

$$1\colon\ \mathsf{eq}(\mathsf{s}(n), x:xs, x:ys) \to \mathsf{eq}(n,xs,ys) \qquad 3\colon\ \mathsf{nats} \to 0 : \mathsf{inc}(\mathsf{nats})$$
$$2\colon\ \mathsf{eq}(n,xs,xs) \to \mathsf{T} \qquad 4\colon\ \mathsf{inc}(x:xs) \to \mathsf{s}(x) : \mathsf{inc}(xs)$$

Take $\mathcal{R} = \{1,2\}$ and $\mathcal{S} = \{3,4\}$. Again, $\mathsf{SNO}(\mathcal{R},\mathcal{S})$, termination of $\mathcal{R}/\mathcal{S}$ and confluence of $\mathcal{S}$ is established. Moreover, one can show

$${}_{\mathcal{R}}\!\leftarrow_{\mathcal{S}}\!\circ\!\rightarrow_{\mathcal{R}} \;=\; \{(\mathsf{eq}(s,t,u),\mathsf{T}) \mid s,\, t,\, u \text{ are terms and } t \leftrightarrow^{*}_{\mathcal{S}} u\}$$

and thus the set is included in $\downarrow_{\mathcal{R}\cup\mathcal{S}}$ because of confluence of $\mathcal{S}$. Hence by using Theorem 2 we conclude that $\mathcal{R}\cup\mathcal{S}$ is confluent.

We conclude this section by mentioning that all conditions of Theorem 2 are essential. One cannot drop $\mathsf{SNO}(\mathcal{R},\mathcal{S})$ nor termination of $\mathcal{R}/\mathcal{S}$, and even replacing joinability of $\mathcal{S}$-critical pairs by joinability of syntactical critical pairs makes the theorem unsound.

Example 11. Consider Huet's example [11]

$$1\colon\ \mathsf{f}(x,x) \to \mathsf{a} \qquad 2\colon\ \mathsf{f}(x,\mathsf{g}(x)) \to \mathsf{b} \qquad 3\colon\ \mathsf{c} \to \mathsf{g}(\mathsf{c})$$

which is known to be non-confluent. If one takes $\mathcal{R} = \{1\}$ and $\mathcal{S} = \{2,3\}$ then $\mathcal{R}/\mathcal{S}$ is terminating, $\mathcal{S}$ is confluent, and ${}_{\mathcal{R}}\!\leftarrow_{\mathcal{S}}\!\circ\!\rightarrow_{\mathcal{R}} = \varnothing \subseteq \downarrow_{\mathcal{R}\cup\mathcal{S}}$. If one takes $\mathcal{R} = \{3\}$ and $\mathcal{S} = \{1,2\}$ then, $\mathsf{SNO}(\mathcal{R},\mathcal{S})$, $\mathcal{S}$ is confluent, and there are no $\mathcal{S}$-critical pairs of $\mathcal{R}$. Furthermore, if one takes $\mathcal{R} = \{1,2\}$ and $\mathcal{S} = \{3\}$ then $\mathsf{SNO}(\mathcal{R},\mathcal{S})$, $\mathcal{R}/\mathcal{S}$ is terminating, $\mathcal{S}$ is confluent, and there are no syntactical critical pairs of $\mathcal{R}$, although $\mathcal{S}$-critical pairs are present.

[1] http://colo6-c703.uibk.ac.at/ttt2/

4 Joinability of $\mathcal{S}$-Critical Pairs

The biggest challenge in applying Theorem 2 is to check $\mathcal{R}{\leftarrow_{\mathcal{S}}}{\rtimes}{\rightarrow_{\mathcal{R}}} \subseteq {\downarrow_{\mathcal{R}\cup\mathcal{S}}}$ automatically. The standard approach is to compute a minimal complete set of $\mathcal{S}$-unifiers for ℓ_1 and $\ell_2|_p$ for each combination of rules $\ell_1 \to r_1$, $\ell_2 \to r_2$ and a position $p \in \mathcal{P}\mathsf{os}_{\mathcal{F}}(\ell_2)$. Then, joinability of its induced critical pairs ensures joinability for all $\mathcal{S}$-unifiers. However, depending on $\mathcal{S}$, the computation of minimal complete sets varies, and worse, minimal complete sets may not even exist for $\mathcal{S}$-unifiable terms. In this section we give sufficient conditions for the joinability and non-joinability of $\mathcal{S}$-critical pairs without performing specific equational unification algorithms.

For the first we show that a most general unifier of strongly $\mathcal{S}$-stable terms is always a most general $\mathcal{S}$-unifier. As the next lemma shows, this allows us to compute $\mathcal{S}$-critical pairs by means of syntactic unification. Here a term t is *strongly $\mathcal{S}$-stable* if for every position $p \in \mathcal{P}\mathsf{os}_{\mathcal{F}}(t)$ there are no term u and substitution σ such that $t|_p\sigma \to_{\mathcal{S}}^* \cdot \xrightarrow{\varepsilon}_{\mathcal{S}} u$. Note that $t\sigma$ is strongly $\mathcal{S}$-stable if t and $x\sigma$ are strongly $\mathcal{S}$-stable for all variables x.

Lemma 12. *If $\mathsf{SNO}(\mathcal{R},\mathcal{S})$ then ℓ is strongly $\mathcal{S}$-stable for all $\ell \to r \in \mathcal{R}$.* □

In order to show the claim on mgu's, we recall the standard inference rules for syntactic unification from [4]. These rules are defined over sets of equalities on terms.

$$\textsc{Eliminate}\quad \frac{\{x \approx t\} \uplus \mathcal{E}}{\{x \approx t\} \cup \mathcal{E}\{x \mapsto t\}} \quad \text{if } x \notin \mathcal{V}\mathsf{ar}(t)$$

$$\textsc{Orient}\quad \frac{\{t \approx x\} \uplus \mathcal{E}}{\{x \approx t\} \cup \mathcal{E}} \quad \text{if } t \notin \mathcal{V}$$

$$\textsc{Delete}\quad \frac{\{t \approx t\} \uplus \mathcal{E}}{\mathcal{E}}$$

$$\textsc{Decompose}\quad \frac{\{f(s_1,\dots,s_n) \approx f(t_1,\dots,t_n)\} \uplus \mathcal{E}}{\{s_1 \approx t_1,\dots,s_n \approx t_n\} \cup \mathcal{E}}$$

We write $\Longrightarrow$ for a derivation by the inferences. The following lemma states that a most general unifier can be computed by a sequence of derivations.

Lemma 13 ([4]). *If s and t are unifiable, there exists $\mathcal{E}$ in solved form such that $\{s \approx t\} \Longrightarrow^* \mathcal{E}$ and $\overrightarrow{\mathcal{E}}$ is an mgu of s and t.* □

The next lemma shows that the inferences of syntactic unification preserve strong $\mathcal{S}$-stability and $\mathcal{S}$-unifiability. We say that a set $\mathcal{E}$ of equalities is strongly $\mathcal{S}$-stable if s and t are strongly $\mathcal{S}$-stable for all $s \approx t \in \mathcal{E}$.

Lemma 14. *Let $\mathcal{S}$ be a confluent TRS. If $\mathcal{E}_1$ is strongly $\mathcal{S}$-stable, $\mathcal{E}_1\sigma \subseteq {\downarrow_{\mathcal{S}}}$, and $\mathcal{E}_1 \Longrightarrow \mathcal{E}_2$, then $\mathcal{E}_2\sigma \subseteq {\downarrow_{\mathcal{S}}}$ and $\mathcal{E}_2$ is strongly $\mathcal{S}$-stable.*

Proof. Suppose $\mathcal{E}_1$ is strongly $\mathcal{S}$-stable, $\mathcal{E}_1\sigma \subseteq \downarrow_{\mathcal{S}}$, and $\mathcal{E}_1 \Longrightarrow \mathcal{E}_2$. We distinguish the inference of $\mathcal{E}_1 \Longrightarrow \mathcal{E}_2$. Because the cases of DELETE and ORIENT are trivial, below we only consider the other two cases:

- ELIMINATE: Suppose $\mathcal{E}_1 = \{x \approx t\} \uplus \mathcal{E}'$ and $\mathcal{E}_2 = \{x \approx t\} \cup \mathcal{E}'\mu$, where $\mu = \{x \mapsto t\}$ and $x \notin \mathcal{V}\mathsf{ar}(t)$. We claim $\mu\sigma \leftrightarrow_{\mathcal{S}}^* \sigma$. Actually it follows from the assumption $x\sigma \downarrow_{\mathcal{S}} t\sigma$. We now prove $\mathcal{E}_2\sigma \subseteq \downarrow_{\mathcal{S}}$. It is sufficient to show $u\mu\sigma \downarrow_{\mathcal{S}} v\mu\sigma$ for an arbitrary $u \approx v \in \mathcal{E}'$. Because $u\sigma \downarrow_{\mathcal{S}} v\sigma$ by assumption, the claim yields $u\mu\sigma \leftrightarrow_{\mathcal{S}}^* v\mu\sigma$. Therefore $u\mu\sigma \downarrow_{\mathcal{S}} v\mu\sigma$ is concluded from confluence of $\mathcal{S}$. To show strong $\mathcal{S}$-stability of $\mathcal{E}_2$, fix $u \approx v \in \mathcal{E}'$. Since u, v, and $x\mu$ are strongly $\mathcal{S}$-stable, so are $u\mu$ and $v\mu$.
- DECOMPOSE: Suppose $\mathcal{E}_1 = \{s \approx t\} \uplus \mathcal{E}'$ and $\mathcal{E}_2 = \{s_1 \approx t_1, \ldots, s_n \approx t_n\} \cup \mathcal{E}'$ with $s = f(s_1, \ldots, s_n)$ and $t = f(t_1, \ldots, t_n)$. Since $\mathcal{E}$ is strongly $\mathcal{S}$-stable, and thus s and t are, s_i and t_i are also strongly $\mathcal{S}$-stable for all $1 \leqslant i \leqslant n$. Furthermore, due to strong $\mathcal{S}$-stability of s and t, $s\sigma \downarrow_{\mathcal{S}} t\sigma$ implies $s_i\sigma \downarrow_{\mathcal{S}} t_i\sigma$ for all $1 \leqslant i \leqslant n$. Therefore, the claim holds. □

We arrive at the aforementioned sufficient condition.

Theorem 15. *Let $\mathcal{S}$ be a confluent TRS. An mgu of strongly $\mathcal{S}$-stable terms s and t is an $\mathcal{S}$-mgu of s and t.*

Proof. Let μ be an arbitrary mgu of strongly $\mathcal{S}$-stable terms s and t. Since μ is trivially an $\mathcal{S}$-unifier of s and t, it is enough to show that μ is more general than an arbitrary $\mathcal{S}$-unifier σ of s and t. By using Lemma 13 there is an $\mathcal{E}$ in solved form such that $\{s \approx t\} \Longrightarrow^* \mathcal{E}$ and $\overrightarrow{\mathcal{E}}$ is an mgu of s and t. Because $s\sigma \leftrightarrow_{\mathcal{S}}^* t\sigma$ and $\mathcal{S}$ is confluent, we have $\{s \approx t\}\sigma \subseteq \downarrow_{\mathcal{S}}$, and thus $\mathcal{E}\sigma \subseteq \downarrow_{\mathcal{S}}$ is obtained by induction on the length of $\Longrightarrow^*$ using Lemma 14. Since $\mathcal{E}$ is in solved form, $x\sigma \downarrow_{\mathcal{S}} x\overrightarrow{\mathcal{E}}\sigma$ holds for all $x \in \mathcal{D}\mathsf{om}(\overrightarrow{\mathcal{E}})$. This means $\sigma \leftrightarrow_{\mathcal{S}}^* \overrightarrow{\mathcal{E}}\sigma$. Since μ is an mgu, there is a substitution ρ with $\overrightarrow{\mathcal{E}} = \mu\rho$. Thus $\sigma \leftrightarrow_{\mathcal{S}}^* \mu\rho\sigma$. Hence μ is more general than σ. □

When automating Theorem 2, confluence of $\mathcal{S}$ and $\mathsf{SNO}(\mathcal{R}, \mathcal{S})$ can be assumed. Therefore, according to Theorem 15 and Lemma 12, a syntactical overlap by an mgu μ is also an $\mathcal{S}$-overlap by $\mathcal{S}$-mgu μ. Thus joinability of its syntactical critical pairs implies joinability of $\mathcal{S}$-critical pairs induced by any $\mathcal{S}$-unifier.

Example 16 (continued from Example 10). We consider again the example with $\mathcal{R} = \{1, 2\}$ and $\mathcal{S} = \{3, 4\}$. Take the first and second rules renamed:

$$1\colon\ \mathsf{eq}(\mathsf{s}(n), x : xs, x : ys) \to \mathsf{eq}(n, xs, ys) \qquad 2\colon\ \mathsf{eq}(m, zs, zs) \to \mathsf{T}$$

We know that there is an overlap between 1 and 2 at root position with the mgu $\mu = \{m \mapsto \mathsf{s}(n), zs \mapsto x : xs, ys \mapsto xs\}$. Elsewhere, even $\mathcal{S}$-overlaps cannot occur. The induced critical pair $(\mathsf{eq}(n, xs, xs), \mathsf{T})$ is trivially joinable by the second rule. Hence $\mathcal{R}\leftarrow_{\mathcal{S}}\mathcal{S}\rightarrow_{\mathcal{R}} \subseteq \downarrow_{\mathcal{R}\cup\mathcal{S}}$ holds.

Confluence of $\mathcal{S}$ cannot be dropped in Theorem 15.

Example 17. Consider the TRS $\mathcal{S}$

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

The terms $\mathsf{f}(x_1,x_1)$ and $\mathsf{f}(x,y)$ are both strongly $\mathcal{S}$-stable, and the substitution $\mu = \{x \mapsto x_1, y \mapsto x_1\}$ is a most general unifier. However, μ is not an $\mathcal{S}$-mgu, because μ is not more general than the other $\mathcal{S}$-unifier $\{x_1 \mapsto x\}$.

Unjoinability of $\mathcal{S}$-critical pairs can be tested similarly to checking non-confluence of a TRS with the function TCAP ([27]).

Definition 18 ([7]). *Let t be a term, and $\mathcal{R}$ a TRS. We define $\mathsf{TCAP}_{\mathcal{R}}(t)$ inductively as a fresh variable, when t is a variable or when $t = f(t_1,\dots,t_n)$ and ℓ and u unify for some (renamed) rule $\ell \to r \in \mathcal{R}$, and u, otherwise. Here u stands for $f(\mathsf{TCAP}_{\mathcal{R}}(t_1),\dots,\mathsf{TCAP}_{\mathcal{R}}(t_n))$.*

Lemma 19. *Let $\ell_1 \to r_1$, $\ell_2 \to r_2 \in \mathcal{R}$ and $p \in \mathcal{P}\mathsf{os}_{\mathcal{F}}(\ell_2)$. If $\ell_1\sigma \leftrightarrow^*_{\mathcal{R}} \ell_2|_p\sigma$, and $\mathsf{TCAP}_{\mathcal{R}}(r_2)$ and $\mathsf{TCAP}_{\mathcal{R}}(\ell_2[r_1]_p)$ do not unify, $\mathcal{R}$ is not confluent.* □

Proof. Using the fact that if $s\sigma \downarrow_{\mathcal{R}} t\tau$ then $\mathsf{TCAP}_{\mathcal{R}}(s)$ and $\mathsf{TCAP}_{\mathcal{R}}(t)$ must unify (see [27]). □

Example 20 (continued from Example 11). Recall $\mathcal{R} = \{1,2\}$ and $\mathcal{S} = \{3\}$:

$$1\colon\ \mathsf{f}(x,x) \to \mathsf{a} \qquad 2\colon\ \mathsf{f}(y,\mathsf{g}(y)) \to \mathsf{b} \qquad 3\colon\ \mathsf{c} \to \mathsf{g}(\mathsf{c})$$

where variables are renamed in rule 2. We denote i-th rule by $\ell_i \to r_i$. While ℓ_1 and $\ell_2|_\varepsilon$ are $(\mathcal{R}\cup\mathcal{S})$-unifiable with $\{x,y \mapsto \mathsf{c}\}$, $\mathsf{TCAP}_{\mathcal{R}\cup\mathcal{S}}(\ell_2[r_1]_\varepsilon) = \mathsf{a}$ and $\mathsf{TCAP}_{\mathcal{R}\cup\mathcal{S}}(r_2) = \mathsf{b}$ do not unify. Thus, by Lemma 19, $\mathcal{R}\cup\mathcal{S}$ is not confluent.

In automation we need to test $\mathcal{S}$-unifiability of ℓ_1 and $\ell_2|_p$. This can be automated by first-order theorem provers (for unit equational problems, so-called UEQ) and indeed non-confluence of the above TRS can be proved automatically, see Section 5. Note that in contrast to [27] this approach only requires $\mathcal{S}$-unifiability but not $\mathcal{S}$-unifiers.

As a final remark, from the absence of a unifier we may not conclude non-existence of $\mathcal{S}$-critical pairs, as illustrated in Example 11.

5 Experiments

In order to assess feasibility of our methods, we implemented Theorem 2 together with Theorem 15 for confluence, and Lemma 19 for non-confluence. In the next subsections we mention details of our implementation and report on experimental data.

5.1 Implementation

In order to automate Theorem 2 we employed $\mathsf{T\!T_2}$ v1.06 [17] for checking relative termination $\mathcal{R}/\mathcal{S}$ and an extended version of Maxcomp [14] for testing $\mathcal{S}$-unifiability, using ordered completion. To check confluence of $\mathcal{S}$, we used the existing three state-of-the-art confluence provers: ACP v0.20 [3][2], CSI v0.1 [27][3], and Saigawa v1.2 [10][4]. Since termination of $\mathcal{R} \cup \mathcal{S}$ cannot be assumed, we only test joinability of $\mathcal{S}$-critical pairs by at most four step rewriting for each term.

We give a brief overview of our procedure. Given a TRS $\mathcal{P}$, we output either `YES` ($\mathcal{P}$ is confluent), `NO` ($\mathcal{P}$ is not confluent), or `MAYBE` (confluence of $\mathcal{P}$ is neither proven nor disproven). We enumerate all possible partitions $\mathcal{P} = \mathcal{R} \uplus \mathcal{S}$, and then for each $(\mathcal{R}, \mathcal{S})$, we test whether $\mathsf{SNO}(\mathcal{R}, \mathcal{S})$, termination of $\mathcal{R}/\mathcal{S}$, and confluence of $\mathcal{S}$ holds. If one of these conditions does not hold, we continue with the next partition; if none is left, we return `MAYBE`. Otherwise, to check the last remaining condition of Theorem 2, namely ${}_{\mathcal{R}}\!\leftarrow_{\mathcal{S}} \circ \rightarrow_{\mathcal{R}} \subseteq \downarrow_{\mathcal{R}\cup\mathcal{S}}$, we proceed in the following way: For all tuples $(\ell_1 \to r_1, p, \ell_2 \to r_2)$ where $\ell_1 \to r_1$ and $\ell_2 \to r_2$ are rules from $\mathcal{R}$ and $p \in \mathcal{P}\mathsf{os}_{\mathcal{F}}(\ell_2)$, we test in the following order:

1. If $\mathsf{REN}(\ell_1)$ and $\mathsf{REN}(\ell_2|_p)$ are not syntactically unifiable, then no $\mathcal{S}$-overlap exists, and we continue with the next tuple. Otherwise,
2. if ℓ_1 and $\ell_2|_p$ are syntactically unifiable with σ, the current tuple forms an $\mathcal{S}$-overlap, so we test joinability of the induced critical pair.
   (a) If joinability holds, we continue with the next tuple.
   (b) If joinability cannot be established, we test whether $\mathsf{TCAP}_{\mathcal{R}\cup\mathcal{S}}(r_2\sigma)$ and $\mathsf{TCAP}_{\mathcal{R}\cup\mathcal{S}}(\ell_2\sigma[r_1\sigma]_p)$ syntactically unify. If they are not unifiable, return `NO`. Otherwise, return `MAYBE`
3. if ℓ_1 and $\ell_2|_p$ are not syntactically unifiable, we check $\mathcal{S} \models \ell_1 \approx \ell_2|_p$ by a theorem prover:
   (a) If unsatisfiability of the formula is detected, no $\mathcal{S}$-overlap exists, and we continue.
   (b) If satisfiability is detected, we test syntactic unifiability of $\mathsf{TCAP}_{\mathcal{R}\cup\mathcal{S}}(r_2)$ and $\mathsf{TCAP}_{\mathcal{R}\cup\mathcal{S}}(\ell_2[r_1]_p)$. If they are not unifiable, return `NO`. If they unify, return `MAYBE`.
   (c) Lastly, if the theorem prover does not provide a conclusive answer, return `MAYBE`

If no tuple remains, we have established ${}_{\mathcal{R}}\!\leftarrow_{\mathcal{S}} \circ \rightarrow_{\mathcal{R}} \subseteq \downarrow_{\mathcal{R}\cup\mathcal{S}}$ and return `YES`. Correctness of the whole procedure can be established using Theorems 2, 15 and Lemmata 12, 19.

5.2 Experimental Results

We tested the implementation on a collection of 32 TRSs, consisting of 29 non-left-linear non-terminating TRSs in the Confluence Problem Database (Cops Nos. 1–116)[5] and Examples 8, 9 and 10. Note that Example 11 is part of the 29 TRSs.

[2] http://www.nue.riec.tohoku.ac.jp/tools/acp/
[3] http://cl-informatik.uibk.ac.at/software/csi/
[4] http://www.jaist.ac.jp/project/saigawa/
[5] http://coco.nue.riec.tohoku.ac.jp/

Table 1. Summary of experimental results (32 TRSs)

	ACP	ACP*	CSI	CSI*	Saigawa	Saigawa*
YES	12	**19**	7	**15**	0	**10**
NO	3	**4**	3	**3**	0	**2**
MAYBE	17	9	17	9	32	20
timeout (60 sec)	0	0	5	5	0	0

The tests were single-threaded run on a system equipped with an Intel Core Duo L7500 with 1.6 GHz and 2 GB of RAM using a timeout of 60 seconds.

The results are depicted in Table 1. [6] Here columns ACP, CSI and Saigawa show results for running the respective tools, and ACP*, CSI* and Saigawa* show results when using the respective tool to show confluence of the $\mathcal{S}$-part in Theorem 2.

It should be noted, that the criteria implemented by Saigawa apply only to left-linear systems, whereas CSI is able to show confluence of non-left-linear systems by order-sorted decomposition [5], and the implementation of ACP includes criteria based on layer preserving [19] and persistency decompositions [1], and the criterion by Gomi et al. [9].

For overall results, there are twelve TRSs for which confluence can be shown by ACP, CSI and Saigawa combined, in fact however all twelve can be shown by ACP alone. Extending with Theorem 2, there are 19 TRSs, for which confluence can be shown by ACP*, CSI* or Saigawa* combined. Similar to the standalone-case, ACP* subsumes both other combinations. As for Example 8, 9 and 10, neither CSI, ACP nor Saigawa can show confluence, whereas all CSI*, ACP* and Saigawa* succeed. Out of the nine TRSs that ACP* missed, four TRSs (Cops Nos. 76, 77, 78, 109) contain AC rules, for which most likely the criterion in [13] applies if suitable equational unification algorithms were implemented (see Section 6), and five TRSs (Nos. 16, 24, 26, 27, 47) are variants of Huet's example (Example 11) or Klop's example [15]: $\{\mathsf{f}(x,x) \to \mathsf{a},\ \mathsf{g}(x) \to \mathsf{f}(x,\mathsf{g}(x)),\ \mathsf{c} \to \mathsf{g}(\mathsf{c})\}$.

6 Related Work

Among others, we compare our criterion with three well-known criteria capable of proving confluence of non-left-linear and non-terminating TRSs. Note that for the second criterion below we use *reversibility* [2] for comparison, because the original criterion requires equational systems for $\mathcal{S}$ rather than rewrite systems. We say that a TRS $\mathcal{S}$ is *reversible* if ${}_{\mathcal{S}}\!\leftarrow \subseteq \to_{\mathcal{S}}^{*}$.

– **Criteria by Non-E-Overlappingness.** The criterion by Gomi et al. [8], later extended in [9], is that a *root-E-overlapping* TRS, that is also strongly

[6] Detailed results are available at http://www.jaist.ac.jp/project/saigawa/

weight-preserving or strongly *depth-preserving*, is confluent. Here E-overlaps are a generalization of overlaps, and strong non-overlappingness plays a major role in deriving sufficient conditions to decide root-E-overlappingness.[7] A TRS is strongly depth preserving, if for any rewrite rule and any variable appearing in both sides, the minimal depth of the variable occurrences in the left-hand side is greater than or equal to the maximal depth of the right hand side's occurrences. Instead of comparing the depth of the variable directly, one can also assign weights to function symbols and compare the weight of the variable occurrence, where the weight is the sum of the function symbols from root to its occurrence. For details of the definitions we refer to [9]. Consider the following TRS:

$$\mathsf{f}(x,x) \to \mathsf{a} \qquad \mathsf{c} \to \mathsf{g}(\mathsf{c}) \qquad \mathsf{g}(x) \to \mathsf{f}(x,x)$$

Confluence of this TRS can be established, since it is depth-preserving and root-E-overlapping. However Theorem 2 cannot be applied, since the TRS cannot be partitioned into a non-empty $\mathcal{R}$ and $\mathcal{S}$, such that $\mathcal{R}/\mathcal{S}$ is terminating — except for $\mathcal{R} = \varnothing$. On the other hand, weight-preservation and depth-preservation impose strong syntactic restrictions on the variable positions. Consider for example the TRS

$$1\colon\ \mathsf{g}(x,x) \to \mathsf{f}(x) \qquad 2\colon\ \mathsf{f}(x) \to \mathsf{f}(\mathsf{f}(x))$$

By taking $\mathcal{R} = \{1\}$ and $\mathcal{S} = \{2\}$, Theorem 2 can be applied. However the second rule violates both strong depth and strong weight-preservation.

– **Criteria by Extended Critical Pairs.** In [13], based on the preliminary work in [12], Jouannaud and Kirchner show that the union of a TRS $\mathcal{R}$ and a reversible TRS $\mathcal{S}$ is confluent if $\mathcal{R}/\mathcal{S}$ and $\rhd/{\leftrightarrow_{\mathcal{S}}}$ are terminating and

$$\mathcal{R}{\leftarrow}_{\mathcal{S}}{\circ}{\rightarrow}_{\mathcal{R}\cup\mathcal{S}\cup\mathcal{S}^{-1}} \subseteq {\rightarrow}^*_{\mathcal{R},\mathcal{S}} \cdot {\leftrightarrow}^*_{\mathcal{S}} \cdot {}^*_{\mathcal{R},\mathcal{S}}{\leftarrow}$$

Here $s \to_{\mathcal{R},\mathcal{S}} t$ if there exist a rule $\ell \to r \in \mathcal{R}$, a position $p \in \mathcal{P}\mathsf{os}(s)$, and a substitution σ, such that $s|_p \leftrightarrow^*_{\mathcal{S}} \ell\sigma$ and $t = s[r\sigma]_p$. Note that $\mathcal{S}$ has a serious restriction: The two termination requirements prohibit application when $\mathcal{S}$ is erasing or collapsing, or even when $C[t] \leftrightarrow^*_{\mathcal{S}} t$. For instance, Examples 9 and 10 cannot be handled due to this restriction. On the other hand it is applicable for mutually overlapping TRSs $\mathcal{R}$ and $\mathcal{S}$, for example:

$$1\colon\ x + x \to x \qquad 2\colon\ x + y \to y + x \qquad 3\colon\ (x + y) + z \to x + (y + z)$$

By taking $\mathcal{R} = \{1\}$ and $\mathcal{S} = \{2, 3\}$, one can easily show confluence of $\mathcal{R}\cup\mathcal{S}$ by using their criterion. However, Theorem 2 cannot be applied because $\mathcal{R}$ and $\mathcal{S}$ overlap on each other. This criterion forms a foundation of AC-completion.

[7] $\mathcal{S}$-overlaps are sometime called $\mathcal{E}$-overlaps but should not be confused with the E-overlaps defined by Gomi et al. [8], originally introduced by Ogawa [18].

– **Criteria by Relative Termination.** Geser [6] introduced several pioneering applications of relative termination. A result of particular interest in this context is the following confluence criterion: A TRS $\mathcal{R} \cup \mathcal{S}$ is confluent if $\mathcal{R}$ is left-linear, $\mathcal{S}$ is confluent, and the following two inclusions hold:

$$\mathcal{S}{\leftarrow_\varnothing}{\rtimes}{\rightarrow_\mathcal{R}} \subseteq (\rightarrow_\mathcal{S}^* \cdot {}_{\mathcal{R}\cup\mathcal{S}}{\leftarrow}^*) \cup (\rightarrow_\mathcal{R} \cdot \downarrow_{\mathcal{R}\cup\mathcal{S}}) \qquad \mathcal{R}{\leftarrow_\varnothing}{\rtimes}{\rightarrow_\mathcal{R}} \subseteq \downarrow_{\mathcal{R}\cup\mathcal{S}}$$

In contrast to Theorem 2, overlaps between rules in $\mathcal{R}$ and $\mathcal{S}$ pose no problem. The following example, due to Geser, shows the power of his approach beyond pure left-linear systems:

$$1\colon\ \mathsf{c}(\mathsf{s}(x),\mathsf{s}(y)) \to \mathsf{c}(x,y) \qquad\qquad 2\colon\ \mathsf{c}(x,x) \to \mathsf{f}(\mathsf{c}(x,x))$$

Then confluence can be established by taking $\mathcal{R} = \{1\}$ and $\mathcal{S} = \{2\}$, whereas Theorem 2 is not applicable. [8] The reason for being able to handle overlaps between $\mathcal{R}$ and $\mathcal{S}$ is, that with the restriction of left-linearity of the $\mathcal{R}$-part, joinability of syntactical critical pairs suffices to establish confluence. On the other hand, the requirement of left-linearity prevents application for Examples 8, 9 and 10, except for choosing $\mathcal{R} = \varnothing$.

7 Conclusion

In this paper we showed a generalization of Knuth and Bendix' confluence criterion, which can deal with non-left-linear, non-terminating TRSs. Moreover we presented its automation technique. As seen in Section 6, conditions required in our criterion are related to the results by Jouannaud and Kirchner [13] and Geser [6]. Any of them exploits relative termination to overcome non-termination, however still relative termination poses a strict restriction. We anticipate that use of *critical pair steps* [10] relaxes this restriction.

Acknowledgements. We thank the anonymous referees for their valuable comments.

References

1. Aoto, T., Toyama, Y.: Persistency of confluence. Journal of Universal Computer Science 3(11), 1134–1147 (1997)
2. Aoto, T., Toyama, Y.: A Reduction-Preserving Completion for Proving Confluence of Non-Terminating Term Rewriting Systems. In: Proc. 22nd RTA. LIPIcs, vol. 10, pp. 91–106 (2011)
3. Aoto, T., Yoshida, J., Toyama, Y.: Proving Confluence of Term Rewriting Systems Automatically. In: Treinen, R. (ed.) RTA 2009. LNCS, vol. 5595, pp. 93–102. Springer, Heidelberg (2009)
4. Baader, F., Nipkow, T.: Term rewriting and all that. Cambridge University Press (1998)

[8] All current confluence tools fail to show confluence of the one rule TRS of rule 2.

5. Felgenhauer, B., Zankl, H., Middeldorp, A.: Layer systems for proving confluence. In: Proc. 31st FSTTCS. LIPIcs, vol. 13, pp. 288–299 (2011)
6. Geser, A.: Relative Termination. PhD thesis, Universität Passau, Available as technical report 91-03 (1990)
7. Giesl, J., Thiemann, R., Schneider-Kamp, P.: Proving and Disproving Termination of Higher-Order Functions. In: Gramlich, B. (ed.) FroCos 2005. LNCS (LNAI), vol. 3717, pp. 216–231. Springer, Heidelberg (2005)
8. Gomi, H., Oyamaguchi, M., Ohta, Y.: On the Church-Rosser property of non-E-overlapping and strongly depth-preserving term rewriting systems. Trans. IPSJ 37(12), 2147–2160 (1996)
9. Gomi, H., Oyamaguchi, M., Ohta, Y.: On the Church-Rosser property of root-E-overlapping and strongly depth-preserving term rewriting systems. Trans. IPSJ 39(4), 992–1005 (1998)
10. Hirokawa, N., Middeldorp, A.: Decreasing diagrams and relative termination. Journal of Automated Reasoning 47, 481–501 (2011)
11. Huet, G.: Confluent reductions: Abstract properties and applications to term rewriting systems: Abstract properties and applications to term rewriting systems. Journal of the ACM 27, 797–821 (1980)
12. Jouannaud, J.P.: Confluent and Coherent Equational Term Rewriting Systems: Application to Proofs in Abstract Data Types. In: Protasi, M., Ausiello, G. (eds.) CAAP 1983. LNCS, vol. 159, pp. 269–283. Springer, Heidelberg (1983)
13. Jouannaud, J.P., Kirchner, H.: Completion of a set of rules modulo a set of equations. SIAM Journal on Computing 15(4), 1155–1194 (1986)
14. Klein, D., Hirokawa, N.: Maximal completion. In: Proc. 22nd RTA. LIPIcs, vol. 10, pp. 71–80 (2011)
15. Klop, J.: Combinatory reduction systems. PhD thesis, Utrecht University (1980)
16. Knuth, D.E., Bendix, P.: Simple word problems in universal algebras. In: Computational Problems in Abstract Algebra, pp. 263–297 (1970)
17. 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)
18. Ogawa, M.: Chew's Theorem Revisited -Uniquely Normalizing Property of Nonlinear Term Rewriting Systems. In: Ibaraki, T., Iwama, K., Yamashita, M., Inagaki, Y., Nishizeki, T. (eds.) ISAAC 1992. LNCS, vol. 650, pp. 309–318. Springer, Heidelberg (1992)
19. Ohlebusch, E.: Modular properties of composable term rewriting systems. Journal of Symbolic Computation 20, 1–41 (1995)
20. Stump, A., Kimmell, G., Omar, R.E.H.: Type preservation as a confluence problem. In: Proc. 22nd RTA. LIPIcs, vol. 10, pp. 345–360 (2011)
21. TeReSe: Term Rewriting Systems. Cambridge Tracts in Theoretical Computer Science, vol. 55. Cambridge University Press (2003)
22. Toyama, Y.: On the Church-Rosser property for the direct sum of term rewriting systems. Journal of the ACM 34(1), 128–143 (1987)
23. Toyama, Y.: Commutativity of term rewriting systems. In: Programming of Future Generation Computers II, pp. 393–407. North-Holland (1988)

24. van Oostrom, V.: Developing developments. Theoretical Computer Science 175(1), 159–181 (1997)
25. van Oostrom, V.: Confluence by Decreasing Diagrams. In: Voronkov, A. (ed.) RTA 2008. LNCS, vol. 5117, pp. 306–320. Springer, Heidelberg (2008)
26. Yamamoto, A.: Completeness of Extending Unification Based on Basic Narrowing. In: Fujisaki, T., Nakata, I., Tanaka, H. (eds.) Logic Programming 1988. LNCS, vol. 383, pp. 1–10. Springer, Heidelberg (1989)
27. Zankl, H., Felgenhauer, B., Middeldorp, A.: CSI – A Confluence Tool. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS, vol. 6803, pp. 499–505. Springer, Heidelberg (2011)
28. Zankl, H., Felgenhauer, B., Middeldorp, A.: Labelings for decreasing diagrams. In: Proc. 22nd RTA. LIPIcs, pp. 377–392 (2011)

Regular Expressions for Data Words

Leonid Libkin and Domagoj Vrgoč

School of Informatics, University of Edinburgh

Abstract. In data words, each position carries not only a letter form a finite alphabet, as the usual words do, but also a data value coming from an infinite domain. There has been a renewed interest in them due to applications in querying and reasoning about data models with complex structural properties, notably XML, and more recently, graph databases. Logical formalisms designed for querying such data often require concise and easily understandable presentations of regular languages over data words.

Our goal, therefore, is to define and study regular expressions for data words. As the automaton model, we take register automata, which are a natural analog of NFAs for data words. We first equip standard regular expressions with limited memory, and show that they capture the class of data words defined by register automata. The complexity of the main decision problems for these expressions (nonemptiness, membership) also turns out to be the same as for register automata. We then look at a subclass of these regular expressions that can define many properties of interest in applications of data words, and show that the main decision problems can be solved efficiently for it.

1 Introduction

Data words are words that, in addition to a letter from a finite alphabet, have a *data value* from an infinite domain associated with each position. For example, $\binom{a}{1}\binom{b}{2}\binom{b}{1}$ is a data word over an alphabet $\Sigma = \{a, b\}$ and $\mathbb{N}$ as the domain of values. It can be viewed as the ordinary word abb in which the first and the third positions are equipped with value 1, and the second position with value 2.

These were introduced in [13] which proposed a natural extension of finite automata for them, called *register automata*. Data words have become an active subject of research lately due to their applications in XML, in particular in static analysis of logic and automata-based XML specifications, and in query evaluation tasks. Indeed, paths in XML trees should account not only for the labels (XML tags) but values of attributes, which can come from an infinite domain, such as $\mathbb{N}$. While logic and automata models are well-understood by now for the structural part of XML (i.e., trees) [15,17,22], adding data values required a concentrated effort for finding good logics and their associated automata [4,6,5,10,20,23]. Connections between logical and automata formalisms have been explored as well, usually with the focus on finding logics with decidable satisfiability problem. A well-known result of [5] shows that FO^2, the two-variable fragment of first-order logic extended by equality test for data values, is decidable over data words. Another account of this was given in [20], where various data word automata models are compared to fragments of FO and MSO with regard

N. Bjørner and A. Voronkov (Eds.): LPAR-18, LNCS 7180, pp. 274–288, 2012.

to their expressive power. Recently, the problem was studied in [3,8]; in particular it was shown that the guarded fragment of MSO defines data word languages that are recognized by non-deterministic register automata.

Data words appear in other areas as well, in particular verification, and querying databases. In several applications, one would like to deal with concise and easy-to-understand representations of languages of data words. These can be used, for example, in extending languages for XML navigation that take into account data values. Another possible example is in the field of verification, in particular from modeling infinite-state systems with finite control [9,12]. Here having a concise representation of system properties is much preferred to long and unintuitive specifications given by e.g. automata.

The need for a good representation mechanism for data word languages is particularly apparent in the area of querying graph databases [1], a data model that is increasingly common in applications including social networks, biology, Semantic Web, and RDF. Many properties of interest in such databases are expressed by regular path queries [18], asking for the existence of a path conforming to a given regular expression, or their extensions [7,2]. Typical queries are specified by the closure of atomic formulae $x \xrightarrow{L} y$ under $\wedge$ and $\exists$; the atoms ask for the existence of a path whose label is in a regular language L between x and y [7]. Typically, such logical languages have been studied without taking data values into account. Recently, however, logical languages that extend regular conditions from words to data words appeared [16]; for such languages we need a concise way of representing regular languages, which is most commonly done by regular expressions (as automata tend to be rather cumbersome to be used in a query language).

The most natural extension of the usual NFAs to data words is *register automata*, first introduced in [13] and studied, for example, in [9,21]. These are in essence finite state automata equipped with a set of registers that allow them to store data values and make a decision about their next step based not only on the current state and the letter in the current position, but also by comparing the current data value with the ones previously stored in registers. They were originally introduced as a mechanism to reason about words over an infinite alphabet (that is, without the finite part), but they easily extend to describe data word languages. Note that a variety of other automata formalisms for data words exist, for example, pebble automata [20,25], data automata [5], and class automata [6]. In this paper we concentrate on languages specified by register automata, since they are the most natural generalization of finite state automata to languages over data words.

As mentioned earlier, if we think of a specification of a data word language, register automata are not the most natural way of providing them: in fact, even over the usual words, regular languages are easier to describe by regular expressions than by NFAs. For example, in XML and graph database applications, specifying paths via regular expressions is completely standard. In many XML specifications (e.g., XPath), data value comparisons are fairly limited: for instance, one checks if two paths ends with the same value. On the other hand, in graph databases, one often needs to specify a path using both labels and data values that occur in it. For those purposes, we need a language for describing regular languages of data words, i.e., languages accepted by register automata. In [16] we started looking at such expressions, but in a context

slightly different from data words. Our goal now is to present a clean account of regular expressions for data words that would:

1. capture the power of register automata over data words, just as the usual regular expressions capture the power of regular languages;
2. have good algorithmic properties, at least matching those of register automata; and
3. admit expressive subclasses with very good (efficient) algorithmic properties.

Note that an attempt to find such regular expressions has been made in [14], but it fell short of even the first goal. In fact, the expressions of [14] are not very intuitive, and they fail to capture some very simple languages like, for example, the language $\{\binom{a}{d}\binom{a}{d'} \mid d \neq d'\}$. In our formalism this language will be described by a regular expression $(a{\downarrow}x) \cdot (a[x^{\neq}])$. This expression says: bind x to be the data value seen while reading a, move to the next position, and check that the symbol is a and that the data value differs from the one in x. The idea of binding is, of course, common in formal language theory, but here we do not bind a letter or a subword (as, for example, in regular expressions with backreferencing) but rather values from an infinite alphabet.

We shall call such expressions *regular expressions with memory*. We formally define their semantics, give examples, prove that they capture register automata and share their algorithmic properties. We then introduce a different kind of regular expressions, *regular expressions with equality*. The previous language, for example, will be captured by the expression $(aa)_{\neq}$, saying that the finite part of the data word reads aa, and the data values at the beginning and at the end are different. We show that such expressions are strictly weaker than expressions with memory, but enjoy nice algorithmic properties.

Organization. In Section 2 we define register automata, and list their closure properties and complexity results about nonemptiness and membership. In Section 3 we introduce regular expressions with memory and show that they define the same class of languages as register automata. In Section 4 we introduce regular expressions with equality, show that while they are strictly weaker than register automata, they admit faster algorithms for decision problems that are based on the close connection of these expressions with pushdown automata. Due to space limitations, some proofs are only sketched, and complete proofs will appear in the full version of the paper.

2 Register Automata over Data Words

A **data word** is simply a finite string over the alphabet $\Sigma \times \mathcal{D}$, where Σ is a finite set of letters and $\mathcal{D}$ an infinite set of data values. That is, in each position a data word carries a letter from Σ and a data value from $\mathcal{D}$. We will denote data words by $\binom{a_1}{d_1} \ldots \binom{a_n}{d_n}$, where $a_i \in \Sigma$ and $d_i \in \mathcal{D}$. The set of all data words over the alphabet Σ and set of data values $\mathcal{D}$ is denoted by $\Sigma[\mathcal{D}]^*$. A data word language is simply a subset $L \subseteq \Sigma[\mathcal{D}]^*$.

Register automata are an analog of NFAs for data words. They move from one state to another by reading the appropriate letter from the finite alphabet and comparing the data value to ones previously stored into the registers. Our version of register automata will use comparisons which are boolean combinations of atomic $=, \neq$ comparisons of data values.

To define such conditions formally, assume that, for each $k > 0$, we have variables $x_1, \ldots, x_k$. Then the set of conditions $\mathcal{C}_k$ is given by the grammar:

$$c \;:=\; \mathtt{tt} \mid \mathtt{ff} \mid x_i^{=} \mid x_i^{\neq} \mid c \wedge c \mid c \vee c \mid \neg c, \quad 1 \leq i \leq k.$$

The satisfaction is defined with respect to a data value $d \in \mathcal{D}$ and a tuple $\tau = (d_1, \ldots, d_k) \in \mathcal{D}^k$ as follows:

- $d, \tau \models \mathtt{tt}$ and $d, \tau \not\models \mathtt{ff}$;
- $d, \tau \models x_i^{=}$ iff $d = d_i$;
- $d, \tau \models x_i^{\neq}$ iff $d \neq d_i$;
- $d, \tau \models c_1 \wedge c_2$ iff $d, \tau \models c_1$ and $d, \tau \models c_2$ (and likewise for $c_1 \vee c_2$);
- $d, \tau \models \neg c$ iff $d, \tau \not\models c$.

In what follows, $[k]$ is a shorthand for $\{1, \ldots, k\}$.

Definition 1 (Register Data Word Automata). *Let Σ be a finite alphabet and k a natural number. A k-*register data word automaton *is a tuple $\mathcal{A} = (Q, q_0, F, T)$, where:*

- *Q is a finite set of states;*
- *$q_0 \in Q$ is the initial state;*
- *$F \subseteq Q$ is the set of final states;*
- *T is a finite set of transitions of the form $(q, a, c) \to (I, q')$, where q, q' are states, a is a label, $I \subseteq [k]$, and c is a condition in $\mathcal{C}_k$.*

Intuitively the automaton traverses a data word from left to right, starting in q_0, with all registers empty. If it reads $\binom{a}{d}$ in state q with register configuration τ, it may apply a transition $(q, a, c) \to (I, q')$ if $d, \tau \models c$; it then enters state q' and changes contents of registers i, with $i \in I$, to d.

To define acceptance formally we first define a configuration of a k-register data word automaton $\mathcal{A}$ on data word $w = \binom{a_1}{d_1} \ldots \binom{a_n}{d_n}$ as a triple (q, j, τ), where q is the current state of $\mathcal{A}$, j is the current position of the symbol in w that $\mathcal{A}$ reads and τ is the current state of the registers. We use the symbol $\perp$ to indicate that a register is unassigned; that is, τ is a k-tuple over $\mathcal{D}_\perp = \mathcal{D} \cup \{\perp\}$. The initial configuration is $(q_0, 1, \tau_0)$, where $\tau_0 = (\perp, \ldots, \perp)$, and any configuration (q, j, τ) with $q \in F$ is a final configuration.

From a configuration (q, j, τ) we can move to a configuration $(q', j + 1, \tau')$ if:

- $(q, a_j, c) \to (I, q')$ is a transition in $\mathcal{A}$,
- $d_j, \tau \models c$ and
- τ' is obtained from τ by replacing data values in registers from I by d_j.

We say that $\mathcal{A}$ accepts w if there is a sequence of configuration of $\mathcal{A}$ on w that leads $\mathcal{A}$ from the initial to a final configuration while reading w.

Remark. Given a k-register data word automaton $\mathcal{A}$ and a tuple $\tau \in \mathcal{D}_\perp^k$, we can turn $\mathcal{A}$ into an automaton $\mathcal{A}(\tau)$ defined just as $\mathcal{A}$ but starting with τ as the register configuration. Such an extension does not affect the class of accepted languages, but will be useful in inductive constructions when automata need not start with all registers unassigned.

A useful property of register automata that will be needed throughout this paper is that, intuitively, such automata can only keep track of as many data values as can be stored in their registers. Formally, we have:

Lemma 1. *Let $\mathcal{A}$ be a k-register data word automaton. If $\mathcal{A}$ recognizes some word of length n, then it recognizes a word of length n that uses at most $k+1$ different data values.*

Proof. We first set some notation. We will say that two k-register assignments τ and $\overline{\tau}$ are of the same equality type if we have $\tau(i) = \tau(j)$ if and only if $\overline{\tau}(i) = \overline{\tau}(j)$, for all $i, j \leq k$. Note that this also implies that $\tau(i) \neq \tau(j)$ if and only if $\overline{\tau}(i) \neq \overline{\tau}(j)$.

We will prove a slightly more general claim, allowing our automata to start with an nonempty assignment of the registers. Let $\mathcal{A}(\tau_0) = (Q, q_0, F, T)$ be a k-register data word automaton, starting with the initial assignment τ_0 in the registers and $w = \binom{a_1}{d_1} \dots \binom{a_n}{d_n}$ a word that it accepts. This means that there is a sequence of states $q_0, q_1, \dots, q_n$, with $q_n \in F$ and a sequence of register assignments $\tau_0, \tau_1, \dots, \tau_n$ such that $(q_{i-1}, a_i, c_i) \to (I_i, q_i) \in T$, that $\tau_{i-1}, d_i \models c_i$ and τ_i is obtained from $\tau_{i-1}$by replacing all registers from I_i with d_i, for $i = 1 \dots n$.

Now let $\overline{S} = \{\tau_0(i) : 1 \leq i \leq k\} - \{\perp\}$. That is $\overline{S}$ contains all the data values from the initial assignment, except the one denoting that the register is empty.

Let S be any set of data values such that $|S| = k+1$ and $\overline{S} \subseteq S$.

We prove by induction on $i \leq n$ that we can define a data word w_i, of length i, such that $w_i = \binom{a_1}{d_1^i} \dots \binom{a_i}{d_i^i}$, where $a_1, \dots a_i$ are from w and $d_1^i, \dots, d_i^i$ are from S. We then show that for this w_i there is a sequence of assignments $\tau_0', \tau_1', \dots \tau_i'$ such that each τ_j' is of the same equality type as τ_j, where $j \leq i$ and it holds that $\tau_{j-1}, d_j \models c_j$, for all $j \leq i$ and each τ_j' is obtained from τ_{j-1}' by replacing all the data values from I_j by d_j. Note that this actually means that $\mathcal{A}$ goes through the same sequence of states while reading w_i as it did while reading w. But then w_n is the desired word from the statement of the lemma.

To prove this we first assume that $i = 1$. We set $\tau_0' = \tau_0$ and select $d \in S$ such that $\tau_0, d \models c_1$ (note that this is possible since we have $k+1$ values at disposal and test only for equality or inequality with a fixed set of k elements) and such that τ_1 and τ_1' are of the same equality type, where τ_1' is obtained from τ_0' by replacing all data values from I_1 by d. Again, this is possible since the original d_1 (from w) could have either been different from all data values in τ_0 or equal to some of them, a choice we can simulate with elements from S. We now set $w_1 = \binom{a_1}{d}$.

Assume now that the claim holds for $i < n$. We prove the claim for $i+1$. By the induction hypothesis we know that there exists a data word $w_i = \binom{a_1}{d_1^i} \dots \binom{a_i}{d_i^i}$ with data values from S and a sequence of assignments each one obtained from the previous by the condition dictated by the original accepting run that allow $\mathcal{A}$ to go through the states $q_0, q_1, \dots, q_i$. We now pick $d \in S$ such that $\tau_i', d \models c_{i+1}$ and τ_{i+1}', obtained from τ_i' by replacing all data values from I_{i+1} by d, has the same equality type as τ_{i+1}. Note that this is possible since τ_i and τ_i' have the same equality type by the induction hypothesis and we have enough data values at our disposal (again, we have to pick d so that it is in the same relation to data values from τ_i' as d_{i+1} from w was to data values from τ_i, but this is possible since each assignment can remember at most k data values). Now we

simply define $w_{i+1} = w_i \cdot \binom{a_{i+1}}{d}$. Note that this w_{i+1} has all the desired properties and can take $\mathcal{A}$ from q_0 to q_{i+1}.

This concludes the proof of the lemma. □

We now show that we can view register automata as NFAs when restricted only to a finite set of data values.

Let $\mathcal{A} = (Q, q_0, F, T)$ be a k-register data word automaton, D a finite set of data values, and $D_\perp = D \cup \{\perp\}$. We transform $\mathcal{A}$ into an NFA $\mathcal{A}_D = (Q', q_0', F', \delta)$ over the alphabet $\Sigma \times D$ as follows:

- $Q' = Q \times D_\perp^k$;
- $q_0' = (q_0, \perp^k)$;
- $F' = F \times D_\perp^k$;
- Whenever we have a transition $(q, a, c) \to (I, q')$ in T, we add the transition

$$((q, \tau), \binom{a}{d}, (q', \tau'))$$

to T if $d, \tau \models c$ and τ' is obtained from τ by putting d in positions from the set I.

It is straightforward to check that $\mathcal{A}$ accepts a data word over $\Sigma \times D$ if and only if $\mathcal{A}_D$ does. That is we obtain the following.

Lemma 2. *Let D be a finite set of data values and $\mathcal{A}$ a register automaton over Σ. Then there exists a finite state automaton $\mathcal{A}_D$ over the alphabet $\Sigma \times D$ such that $w \in L(\mathcal{A}_D)$ iff $w \in L(\mathcal{A})$, for every w with data values from D. Moreover, $\mathcal{A}_D$ is of size exponential in the size of $\mathcal{A}$ and polynomial in the size of D.*

Since register automata closely resemble classical finite state automata, it is not surprising that some (although not all) constructions valid for NFAs can be carried over to register automata. We now recall results about closure properties of register automata [13]. Although our notion of automata is slightly different than the one used there, all constructions from [13] can be easily modified to work in the setting proposed here.

Fact 1 ([13]).

1. *The set of languages recognized by register automata is closed under union, intersection, concatenation and Kleene star.*
2. *Languages recognized by register automata are not closed under complement.*
3. *Languages recognized by register automata are closed under automorphisms: that is, if $f : \mathcal{D} \to \mathcal{D}$ is an automorphism and w is accepted by $\mathcal{A}$, then the data word $f(w)$ in which every data value d is replaced by $f(d)$ is also accepted by $\mathcal{A}$.*

Membership and nonemptiness are some of the most important decidability problems related to formal languages. We now recall the exact complexity of these problems for register automata. Since the model of register automata we use here differs slightly from the one in previous work, we sketch how these results carry over to our model.

Recall that nonemptiness problem for an automaton $\mathcal{A}$ is checking whether $L(\mathcal{A}) \neq \emptyset$.

Fact 2 ([9]). *The nonemptiness problem for register data word automata is* PSPACE*-complete.*

The lower bound will follow from Theorem 1 and Proposition 1. For the upper bound we convert our k-register automaton $\mathcal{A}$ into an NFA $\mathcal{A}_D$ over the alphabet $\Sigma \times D$ (as in the Lemma 2), where $D = \{0, \ldots, k+1\}$. We know that $\mathcal{A}_D$ recognizes all data words from $L(\mathcal{A})$ using only data values from D. By Lemma 1 and invariance under automorphisms, we know that checking $\mathcal{A}$ for nonemptiness is equivalent to checking $\mathcal{A}_D$ for nonemptiness. Using on-the-fly construction we get the desired result (note that $\mathcal{A}_D$ can not be created before checking it for nonemptiness).

The membership problem asks, for an automaton $\mathcal{A}$ and a word w, whether $w \in L(\mathcal{A})$.

Fact 3 ([21]). *The membership problem for register data word automata is* NP*-complete.*

The lower bound will follow from Theorem 1 and Proposition 2. For the upper bound it simply suffices to guess an accepting run of the automaton.

3 Regular Expressions with Memory

In this section we develop regular expressions capturing register automata in the same way as the usual regular expressions capture regular languages. To do this notice that register automata could be pictured as finite state automata whose transitions between states have labels of the form $a[c]\downarrow I$, where I is a set of registers. Such an automaton can move from one state to another using an arrow $a[c]\downarrow I$ if the letter it sees is a, and the data value (together with the current register assignment) satisfies the condition c. It then proceeds to the next state and updates the registers in I with the current data value. This suggests that the basic building blocks for our expressions will be expressions of the form $a[c]\downarrow I$.

Definition 2 (Expressions with Memory). *Let Σ be a finite alphabet and $x_1, \ldots, x_k$ a finite set of variables. Regular expressions with memory over $\Sigma[x_1, \ldots, x_k]$ are defined inductively as follows:*

- *ε and $\emptyset$ are expressions;*
- *$a[c]\downarrow I$ is an expression; here $a \in \Sigma$, c is a condition in $\mathcal{C}_k$, and $I \subseteq \{x_1, \ldots, x_k\}$;*
- *If e, e_1, e_2 are expressions, then so are $e_1 + e_2$, $e_1 \cdot e_2$, and e^*.*

For convenience we will write just a if $I = \emptyset$ and the condition $c = \texttt{tt}$ and similarly when only one of them can be ignored. Also, if $I = \{x\}$, we write $a[c]\downarrow x$, or $a\downarrow x$ when $c = \texttt{tt}$, instead of $a[c]\downarrow I$.

To define the semantics, we first define what it means for an expression e over $\Sigma[x_1, \ldots x_k]$, a data word w and a tuple $\sigma \in \mathcal{D}_\perp^k$ to infer another tuple $\sigma' \in \mathcal{D}_\perp^k$, viewed as partial assignment of values to variables. We do this inductively on e.

- $(\varepsilon, w, \sigma) \vdash \sigma'$ iff $w = \varepsilon$ and $\sigma' = \sigma$.

- $(a[c]{\downarrow}I, w, \sigma) \vdash \sigma'$ iff $w = \binom{a}{d}$ and $\sigma, d \models c$ and σ' is obtained from σ by assigning d to each $x_i \in I$.
- $(e_1 \cdot e_2, w, \sigma) \vdash \sigma'$ iff $w = w_1 \cdot w_2$ and there exists a valuation σ'' such that $(e_1, w_1, \sigma) \vdash \sigma''$ and $(e_2, w_2, \sigma'') \vdash \sigma'$.
- $(e_1 + e_2, w, \sigma) \vdash \sigma'$ iff $(e_1, w, \sigma) \vdash \sigma'$ or $(e_2, w, \sigma) \vdash \sigma'$.
- $(e^*, w, \sigma) \vdash \sigma'$ iff
  1. $w = \varepsilon$ and $\sigma = \sigma'$, or
  2. $w = w_1 \cdot w_2$ and there exists a valuation σ'' such that $(e, w_1, \sigma) \vdash \sigma''$ and $(e^*, w_2, \sigma'') \vdash \sigma'$.

We say that a regular expression e *induces* a tuple $\sigma \in \mathcal{D}_\perp^k$ on a data word w if $(e, w, \perp^k) \vdash \sigma$. We then define $L(e)$, the language of e, as the set of all data words on which e induces some tuple σ. A regular expression with memory e is *well-formed* if every variable is bound before being used in a condition. From now on we will assume that all our expressions are well-formed.

Example 1. We now give a few examples of data word languages definable by regular expressions with memory.

1. The expression $(a{\downarrow}x) \cdot (b[x^{\neq}])^*$ defines the language of data words where word part reads ab^* and such that the first data value is different from all others. It binds while reading the first a, and then it proceeds checking that the letter is b and condition $x^{\neq}$ is satisfied, which is expressed by $b[x^{\neq}]$; the expression is then put in the scope of * to indicate that the number of such values is arbitrary.
2. The language of data words in which two data values are the same is given by the expression $\Sigma^* \cdot (\Sigma{\downarrow}x) \cdot \Sigma^* \cdot (\Sigma[x^{=}]) \cdot \Sigma^*$, where Σ is the shorthand for $a_1 + \ldots + a_l$, whenever $\Sigma = \{a_1, \ldots, a_l\}$ and $\Sigma{\downarrow}x$ is a shorthand for $a_1{\downarrow}x + \ldots + a_l{\downarrow}x$. It says: at some point, bind x, and then check that after one or more letters, we have the same data value.
3. The language of data words in which the last two data values occur elsewhere in the word with label a is defined by $\Sigma^* \cdot (a{\downarrow}x) \cdot \Sigma^* \cdot (a{\downarrow}y) \cdot \Sigma^* \cdot (\Sigma[x^{=}] + \Sigma[y^{=}]) \cdot (\Sigma[x^{=}] + \Sigma[y^{=}])$.

3.1 Equivalence with Register Automata

In this section we prove that every language recognized by register automata can also be described by a regular expression with memory and vice versa. In fact, we show a tighter connection, from which the equivalence will follow. Let $L(e, \sigma, \sigma')$ be the set of all data words w such that $(e, w, \sigma) \vdash \sigma'$, and let $L(\mathcal{A}, \sigma, \sigma')$ be the set of all data words w such that w is accepted by $\mathcal{A}(\sigma)$, and there exists an accepting run that ends with a register configuration σ'.

Theorem 1. *1. For every regular expression with memory e over $\Sigma[x_1, \ldots, x_k]$ there exists (and can be constructed in logarithmic space) a k-register data word automaton $\mathcal{A}_e$ such that $L(e, \sigma, \sigma') = L(\mathcal{A}_e, \sigma, \sigma')$ for every $\sigma, \sigma' \in \mathcal{D}_\perp^k$.*

2. For every k-register data word automaton $\mathcal{A}$ there exists (and can be constructed in exponential time) a regular expression with memory $e_\mathcal{A}$ over $x_1, \ldots, x_k$ such that $L(e_\mathcal{A}, \sigma, \sigma') = L(\mathcal{A}, \sigma, \sigma')$ for every $\sigma, \sigma' \in \mathcal{D}_\perp^k$.

The structure of the proof follows of course the standard NFA-regular expressions equivalence, cf. [24], with all the necessary adjustments to handle transitions induced by $a[c] \downarrow I$. Details can be found in the complete version of the paper. Since $L(e) = \bigcup_\sigma L(e, \bot^k, \sigma)$ and $L(\mathcal{A}) = \bigcup_\sigma L(\mathcal{A}, \bot^k, \sigma)$, we obtain:

Corollary 1. *The classes of languages of data words definable by k-register data word automata, and by regular expressions with memory over $\Sigma[x_1, \ldots, x_k]$ are the same.*

3.2 Properties of Regular Expressions with Memory

Corollary 1 and closure properties of register automata immediately imply that languages defined by regular expressions with memory are closed under union, intersection, concatenation, Kleene star, but are *not* closed under complement.

We now turn to the nonemptiness problem, i.e., checking whether $L(\mathcal{A}) \neq \emptyset$. Since going from expressions to automata is polynomial, we get a PSPACE upper bound (see Fact 2). One can also prove a matching lower bound, by adapting techniques used in a different but related setting [16] for combined complexity bounds on query evaluation over graph databases and obtain:

Proposition 1. *The nonemptiness problem for regular expressions with memory is* PSPACE*-complete.*

Next we move to the membership problem, i.e., checking whether $w \in L(e)$. Again, since e can be translated efficiently into an equivalent automaton $\mathcal{A}_e$, Fact 3 gives an NP upper bound. We can prove a matching lower bound as well:

Proposition 2. *The membership problem for regular expressions with memory is* NP*-complete.*

Proof. For the lower bound we do a reduction from 3-SAT.

Let $\varphi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \ldots \wedge (a_k \vee b_k \vee c_k)$, be an arbitrary 3-CNF formula. We will construct a data word w and a regular expression with memory e, both of length linear in the length of φ, such that φ is satisfiable if and only if $w \in L(e)$.

Let $x_1, x_2, \ldots, x_n$ be all the variables occurring in φ. We define w as the following data word:

$$w = \left(\binom{a}{0}\binom{b}{1}\right)^n \left(\binom{a_1}{d_{a_1}}\binom{b_1}{d_{b_1}}\binom{c_1}{d_{c_1}}\right) \cdots \left(\binom{a_k}{d_{a_k}}\binom{b_k}{d_{b_k}}\binom{c_k}{d_{c_k}}\right),$$

where $d_{a_i} = 1$, if $a_i = x_j$, for some $j \in \{1, \ldots n\}$ and 0, if $a_i = \overline{x_j}$ and similarly for d_{b_i}, d_{c_i} (note that every a_i, b_i, c_i is of the for x_j, or $\overline{x_j}$, so this is well defined).

Also note that we are using a_i, b_i, c_i both for literals in φ and for letters of our finite alphabet, but this should not arise any confusion. The idea behind this data word is that with the first part that corresponds to the variables, i.e. with $(\binom{a}{0}\binom{b}{1})^n$, we guess a satisfying assignment and the next part corresponds to each conjunct in φ and its data value is set such that if we stop at any point for comparison we get a true literal in this conjunct.

We now define e as the following regular expression with memory:

$$e = (a{\downarrow}x_1 + ab{\downarrow}x_1) \cdot b^* \cdot (a{\downarrow}x_2 + ab{\downarrow}x_2) \cdot b^* \cdot (a{\downarrow}x_3 + ab{\downarrow}x_3) \cdots \\ b^* \cdot (a{\downarrow}x_n + ab{\downarrow}x_n) \cdot b^* \cdot \text{clause}_1 \cdot \text{clause}_2 \ldots \text{clause}_k,$$

where each clause_i corresponds to the i-th conjunct of φ in the following manner.

If ith conjunct uses variables $x_{j_1}, x_{j_2}, x_{j_3}$(possibly with repetitions), then

$$\text{clause}_i = a_i[x_{j_1}^{=}] \cdot b_i \cdot c_i + a_i \cdot b_i[x_{j_2}^{=}] \cdot c_i + a_i \cdot b_i \cdot c_i[x_{j_3}^{=}].$$

We now prove that φ is satisfiable if and only if $w \in L(e)$.

Assume first that φ is satisfiable. Then there's a way to assign a value to each x_i such that for every conjunct in φ at least one literal is true. This means that we can traverse the first part of w to chose the corresponding values for variables bounded in e. Now with this choice we can make one of the literals in each conjunct true, so we can traverse every clause_i using one of the tree possibilities.

Assume now that $w \in L(e)$. This means that after choosing the data values for variables (and thus a valuation for φ, since all data values are either 0 or 1), we are able to traverse the second part of w using these values. This means that for every clause_i there is a letter after which the data value is the same as the one bounded to the corresponding variable. Since data values in the second part of w correspond to literal in the corresponding conjunct of φ to evaluate to 1, we know that this valuation satisfies our formula φ. □

4 Regular Expressions with Equality

In this section we define yet another kind of expressions, regular expressions with equality, that will have significantly better algorithmic properties that regular expressions with memory and register automata, while still retaining much of their expressive power. The idea is to allow checking for (in)equality of data values at the beginning and at the end of subwords conforming to subexpressions.

Originally motivation for such expressions came from graph databases, where they were used to lower combined complexity of queries that mixed data and topology. Such queries, with conditions specified by register automata, had PSPACE-complete combined complexity; with the restrictions similar to those described here, it dropped to PTIME, or to NP-complete when such queries were closed under conjunction and existential quantification [16]. These bounds are the best possible, in light of the results on regular path queries. We also argue that, although limited in expressive power, they still allow specification of interesting properties in graph or XML databases.

Definition 3 (Expressions with equality). *Let Σ be a finite alphabet. Then* regular expressions with equality *are defined by the grammar:*

$$e \;:=\; \emptyset \mid \varepsilon \mid a \mid e + e \mid e \cdot e \mid e^+ \mid e_= \mid e_{\neq} \tag{1}$$

where a ranges over alphabet letters. The language $L(e)$ of data words denoted by a regular expression with equality e is defined as follows.

- $L(\emptyset) = \emptyset$.
- $L(\varepsilon) = \{\varepsilon\}$.
- $L(a) = \{\binom{a}{d} \mid d \in \mathcal{D}\}$.
- $L(e \cdot e') = L(e) \cdot L(e')$.
- $L(e + e') = L(e) \cup L(e')$.
- $L(e^+) = \{w_1 \cdots w_k \mid k \geq 1$ *and each* $w_i \in L(e)\}$.
- $L(e_=) = \{\binom{a_1}{d_1} \ldots \binom{a_n}{d_n} \in L(e) \mid d_1 = d_n\}$.
- $L(e_{\neq}) = \{\binom{a_1}{d_1} \ldots \binom{a_n}{d_n} \in L(e) \mid d_1 \neq d_n\}$.

Without any syntactic restrictions, there may be "pathological" expressions that, while formally defining the empty language, should nonetheless be excluded as really not making sense. For example, $\varepsilon_=$ is formally an expression, and so is $a_{\neq}$, although it is clear they cannot denote any data word. We exclude them by defining well-formed expressions as follows. We say that the usual regular expression e reduces to ε (respectively, to singletons) if $L(e)$ is ε or $\emptyset$ (or $|w| \leq 1$ for all $w \in L(e)$). Then we say that regular expression with equality is *well-formed* if it contains no subexpressions of the form $e_=$ or $e_{\neq}$, where e reduces to ε, or to singletons. From now on we will assume that all our expressions are well formed.

Note that we use $^+$ instead of $*$ for iteration. This is done for technical purposes (the ease of translation) and does not reduce expressiveness, since we can always use e^* as shorthand for $e^+ + \varepsilon$.

We now provide two examples. The expression $\Sigma^* \cdot (a \cdot \Sigma^* \cdot a)_= \cdot \Sigma^*$ denotes the language of data words that contain two a-labelled positions with the same data value. In XML this simply specifies that a is not a key. The language of data words in which the first and the last data value are different is given by $(\Sigma \cdot \Sigma^+)_{\neq}$.

4.1 Properties of Regular Expressions with Equality

As expected regular expressions with equality will be subsumed by register automata, but unlike expressions with memory, they will be less expressive, as illustrated by the following result.

Proposition 3. *Regular expressions with equality are strictly weaker than regular expressions with memory.*

When proving this, we simply show that regular expressions with equality can be translated into register automata using an easy inductive construction. Moreover, this translation can be carried in PTIME (in fact in NLOGSPACE). To show they are strictly weaker than expressions with memory or register automata, we show that they cannot define the language of $(a{\downarrow}x) \cdot (a[x^{\neq}])^*$. To do so, we introduce another kind of automata, called weak register automata, and show that they cannot recognize that language and that they can define any language described by expressions with equality.

As immediately follows from their definition, languages denoted by regular expressions with equality are closed under union, concatenation, and Kleene star. Also, it is straightforward to see that they are closed under automorphisms. However:

Proposition 4. *Languages recognized by regular expressions with equality are not closed under intersection and complement.*

Proof sketch. Observe first that the expression $\Sigma^* \cdot (\Sigma \cdot \Sigma^+)_= \cdot \Sigma^*$ defines a language of data words containing two positions with the same data value. The complement of this language is the set of all data words where all data values are different, which is not recognizable by register automata [13]. By Proposition 3 this implies that regular expressions with memory are not closed under complement.

To see that they are not closed under intersection we first show that the language

$$L = \left\{ \binom{a}{d_1}\binom{a}{d_2}\binom{a}{d_3} \;\middle|\; d_1 \neq d_2, d_1 \neq d_3 \text{ and } d_2 \neq d_3 \right\}$$

is not recognizable by any regular expression with equality. To prove this we simply try out all possible combinations of expressions that use at most three concatenated occurrences of a. Note that we can eliminate any expression with more that three as, or one that uses * (since this results in arbitrary long words), or union (since every member of the union would have to define words from this language and since we do not use constants we cannot just split the language into two or more parts). Also, no $=$ can occur in our expression (for subexpressions of length at least 2). This reduces the number of potential expressions to denote the language to finitely many possibilities, and we simply try them all.

Now observe that the expression $e_1 = ((a \cdot a)_\neq \cdot a)_\neq$ defines the language

$$L_1 = \left\{ \binom{a}{d_1}\binom{a}{d_2}\binom{a}{d_3} \;\middle|\; d_1 \neq d_2 \text{ and } d_1 \neq d_3 \right\}.$$

Similarly $e_2 = a \cdot (a \cdot a)_\neq$ defines

$$L_2 = \left\{ \binom{a}{d_1}\binom{a}{d_2}\binom{a}{d_3} \;\middle|\; d_2 \neq d_3 \right\}.$$

Note that $L = L_1 \cap L_2$, so if regular expressions with equality were closed under intersection they would also have been able to define the language L. □

To obtain fast membership and nonemptiness testing algorithms for expressions with equality, we first show how to reduce them to pushdown automata when only finite alphabets are involved.

Assume that we have a finite set D of data values. We now inductively construct PDAs $P_{e,D}$ for all regular expressions with equality e. The words recognized by these automata will be precisely the words from $L(e)$ whose data values come from D.

We construct these PDAs so that they accept by final state and furthermore have the property that only transitions of the kind $(q_0, \binom{a}{d}, X, \alpha, q)$ leave the initial state (that is any transition leaving the initial state will consume a letter) and every transition entering a final state will consume a letter. We will maintain these properties throughout the inductive construction.

It is quite clear how to construct the automata for $e = \varepsilon, e = \emptyset$ and $e = a$. For $e_1 + e_2, e_1 \cdot e_2$ and e_1^+ we use standard constructions, while for $e = (e_1)_=$, or $e = (e_1)_\neq$ we push the first data value on the stack, mark it by a new stack symbol and then proceed with the run of the automaton for e_1 which exists by the induction hypothesis. Every time we enter a final state of that automaton we simply empty the stack until we reach

the first data value (here we use the new stack symbol) and compare it for equality or inequality with the last data value of the input word. The additional assumptions are here to assure that the construction works correctly. Details of the proof can be found in the full version.

Lemma 3. *The language of words accepted by each PDA $P_{e,D}$ is equal to the set of data words in $L(e)$ whose data values come from D. Moreover, the PDA $P_{e,D}$ has at most $O(|e|)$ states and $O(|e| \times (|D|^2 + |e|))$ transitions, and can be constructed in polynomial time.*

From this and Lemma 1 it is easy to obtain the following.

Theorem 2. *The nonemptiness problem for regular expressions with equality is in* PTIME.

To see this, take an arbitrary expression with equality e and convert it to a n-register data word automaton $\mathcal{A}$ that recognizes the same language. From the translation, we know that n will be at most the number of times $=$ and $\neq$ appear in e. Now do the construction from Lemma 3 for e and $D = \{0, 1, \ldots, n+1\}$ to obtain a PDA $P_{e,D}$. Proposition 3 and Lemma 1 now imply that checking if $L(e) \neq \emptyset$ is equivalent to checking $P_{e,D}$ for nonemptiness. Since this automaton is of polynomial size, we can check it for nonemptiness in PTIME thus obtaining the desired result.

Proposition 5. *The membership problem for regular expressions with equality is in* PTIME.

As in the proof of Theorem 2, we construct a PDA $P_{e,D}$ for e and $D = \{0, 1, \ldots, n\}$, where n is the length of the input word w. By invariance under automorphisms we can assume that data values in w come from the set D. Next we simply check that the word is accepted by $P_{e,D}$ and since this can be done in PTIME we get the desired result. The correctness of this algorithm follows from Lemma 3.

It is natural to ask whether NFAs could not have been used instead of pushdown automata. The answer is that they can be used to capture languages of data words described by regular expressions with equality over a finite set of data values, but the cost is necessarily exponential, and hence we cannot possible use them to derive Theorem 2. That is, we can first show:

Proposition 6. *For every regular expression with equality e over the alphabet Σ and a finite set D of data values there exists an NFA $\mathcal{A}_{e,D}$, of the size exponential in $|e|$, recognizing precisely those data words from $L(e)$ that use data values from D.*

Proof sketch. We prove this by structural induction on regular expressions with equality. All of the standard cases are carried out as usual. Thus we only have to describe the construction for subexpressions of the form $e_=$ and $e_{\neq}$. In both cases by the induction hypothesis we know that there is an NFA $\mathcal{A}_{e,D}$ recognizing words in $L(e)$ with data values from D. The automaton for $\mathcal{A}_{e_{\neq},D}$ (and likewise for $\mathcal{A}_{e_=,D}$) will consist of $|D|$ disjoint copies of $\mathcal{A}_{e,D}$, each designated to remember the first data value read when processing the input. According to this, whenever our automaton would enter a final

state we test that the current data value is different (or the same) to the one corresponding to this copy of the original automaton. This is done in a manner analogous to the one used in the proof of Proposition 3. □

However, the exponential lower bound is the best we can do in the general case. To see this, we define a sequence of regular expressions with memory $\{e_n\}_{n\in\mathbb{N}}$, over the alphabet $\Sigma = \{a\}$, and each of length linear in n. We then show that for $D = \{0, 1\}$ every regular expression over the alphabet $\Sigma \times D$ recognizing precisely those data words from $L(e_n)$ with data values in D has length exponential in $|e_n|$.

To prove this we will use the following theorem for proving lower bounds of NFAs [11]. Let $L \subseteq \Sigma^*$ be a regular language and suppose there exists a set $P = \{(x_i, y_i) : 1 \leq i \leq n\}$ of pairs such that:

1. $x_i \cdot y_i \in L$, for every $i = 1, \ldots n$, and
2. $x_i \cdot y_j \notin L$, for $1 \leq i, j \leq n$ and $i \neq j$.

Then any NFA accepting L has at least n states.

Thus to prove our claim it suffices to find such a set of size exponential in the length of e_n.

Next we define the expressions e_n inductively as follows:

- $e_1 = (a \cdot a)_=$,
- $e_{n+1} = (a \cdot e_n \cdot a)_=$.

It is easy to check that $L(e_n) = \{w \cdot w^{-1} : w \in (\Sigma \times \{0, 1\})^n\}$, where w^{-1} denotes the reverse of w.

Now let $w_1, \ldots w_{2^n}$ be a list of all the elements in $(\Sigma \times \{0, 1\})^n$ in arbitrary order. We define the pairs in P as follows:

- $x_i = w_i$,
- $y_i = (w_i)^{-1}$.

Since these pairs satisfy the above assumptions 1) and 2), we conclude, using the result of [11], that any NFA recognizing $L(e_n)$ has at least $O(2^{|e_n|})$ states, so no regular expression describing it can be of length polynomial in $|e_n|$.

5 Conclusions and Future Work

Here we addressed the problem of finding analogs of regular expressions for register automata, and explored their language-theoretic properties. We also defined an expressive subclass with good algorithmic properties. In the future we would like to try and find an intermediate class of expressions that could be used to recognize a larger class of languages than regular expressions with equality, but still retain low complexity of nonemptiness and membership checking. We would also like to explore how these new classes of expressions behave as query languages in graph database models. Since language nonemptiness is closely related to query evaluation in that context we are hopeful to obtain fast and expressive query languages based on these new classes of expressions.

Acknowledgment. We would like to thank Juan Reutter and Tony Tan for helpful comments during the preparation of this paper. Work partially supported by EPSRC grant G049165 and FET-Open Project FoX, grant agreement 233599.

References

1. Angles, R., Gutiérrez, C.: Survey of graph database models. ACM Comput. Surv. 40(1) (2008)
2. Barceló, P., Hurtado, C., Libkin, L., Wood, P.: Expressive languages for path queries over graph-structured data. In: PODS 2010, pp. 3–14 (2010)
3. Benedikt, M., Ley, C., Puppis, G.: Automata vs. Logics on Data Words. In: Dawar, A., Veith, H. (eds.) CSL 2010. LNCS, vol. 6247, pp. 110–124. Springer, Heidelberg (2010)
4. Bojanczyk, M., Parys, P.: XPath evaluation in linear time. In: PODS 2008, pp. 241–250 (2008)
5. Bojanczyk, M., David, C., Muscholl, A., Schwentick, T., Segoufin, L.: Two-variable logic on words with data. ACM TOCL 12(4) (2011)
6. Bojanczyk, M., Lasota, S.: An extension of data automata that captures XPath. In: LICS 2010, pp. 243–252 (2010)
7. Calvanese, D., de Giacomo, G., Lenzerini, M., Vardi, M.Y.: Rewriting of regular expressions and regular path queries. JCSS 64(3), 443–465 (2002)
8. Colcombet, T., Ley, C., Puppis, G.: On the Use of Guards for Logics with Data. In: Murlak, F., Sankowski, P. (eds.) MFCS 2011. LNCS, vol. 6907, pp. 243–255. Springer, Heidelberg (2011)
9. Demri, S., Lazic, R.: LTL with the freeze quantifier and register automata. ACM TOCL 10(3) (2009)
10. Figueira, D.: Satisfiability of downward XPath with data equality tests. In: PODS 2009, pp. 197–206 (2009)
11. Glaister, I., Shallit, J.: A lower bound technique for the size of nondeterministic finite automata. IPL 59, 75–77 (1996)
12. Grumberg, O., Kupferman, O., Sheinvald, S.: Variable Automata over Infinite Alphabets. In: Dediu, A.-H., Fernau, H., Martín-Vide, C. (eds.) LATA 2010. LNCS, vol. 6031, pp. 561–572. Springer, Heidelberg (2010)
13. Kaminski, M., Francez, N.: Finite memory automata. Theoretical Computer Science 134(2), 329–363 (1994)
14. Kaminski, M., Tan, T.: Regular expressions for languages over infinite alphabets. Fundam. Inform. 69(3), 301–318 (2006)
15. Libkin, L.: Logics for unranked trees: an overview. Logical Methods in Computer Science 2(3) (2006)
16. Libkin, L., Vrgoč, D.: Regular path queries on graphs with data. In: ICDT 2012 (to appear, 2012)
17. Marx, M.: Conditional XPath. ACM TODS 30, 929–959 (2005)
18. Mendelzon, A.O., Wood, P.T.: Finding regular simple paths in graph databases. SIAM J. Comput. 24(6), 1235–1258 (1995)
19. Neven, F.: Automata theory for XML researchers. SIGMOD Record 31(3), 39–46 (2002)
20. Neven, F., Schwentick, T., Vianu, V.: Finite state machines for strings over infinite alphabets. ACM TOCL 5(3), 403–435 (2004)
21. Sakamoto, H., Ikeda, D.: Intractability of decision problems for finite-memory automata. Theor. Comput. Sci. 231(2), 297–308 (2000)
22. Schwentick, T.: Automata for XML – A survey. JCSS 73(3), 289–315 (2007)
23. Segoufin, L.: Automata and Logics for Words and Trees over an Infinite Alphabet. In: Ésik, Z. (ed.) CSL 2006. LNCS, vol. 4207, pp. 41–57. Springer, Heidelberg (2006)
24. Sipser, M.: Introduction to the Theory of Computation. PWS Publishing (1997)
25. Tan, T.: Graph reachability and pebble automata over infinite alphabets. In: LICS 2009, pp. 157–166 (2009)

Automatic Verification of TLA$^+$ Proof Obligations with SMT Solvers

Stephan Merz[1] and Hernán Vanzetto[1,2]

[1] INRIA Nancy Grand-Est & LORIA, Nancy, France
[2] Microsoft Research-INRIA Joint Centre, Saclay, France

Abstract. TLA$^+$ is a formal specification language that is based on ZF set theory and the Temporal Logic of Actions TLA. The TLA$^+$ proof system TLAPS assists users in deductively verifying safety properties of TLA$^+$ specifications. TLAPS is built around a *proof manager*, which interprets the TLA$^+$ proof language, generates corresponding proof obligations, and passes them to backend verifiers. In this paper we present a new backend for use with SMT solvers that supports elementary set theory, functions, arithmetic, tuples, and records. Type information required by the solvers is provided by a typing discipline for TLA$^+$ proof obligations, which helps us disambiguate the translation of expressions of (untyped) TLA$^+$, while ensuring its soundness. Preliminary results show that the backend can help to significantly increase the degree of automation of certain interactive proofs.

1 Introduction

TLA$^+$ [10] is a language for specifying and verifying systems, in particular concurrent and distributed algorithms. It is based on a variant of Zermelo-Fraenkel (ZF) set theory for specifying the data structures, and on the Temporal Logic of Actions (TLA) for describing the dynamic system behavior. Recently, a first version of the TLA$^+$ proof system TLAPS [5] has been developed, in which users can deductively verify safety properties of TLA$^+$ specifications. TLA$^+$ contains a declarative language for writing hierarchical proofs, and TLAPS is built around a *proof manager*, which interprets this proof language, expands the necessary module and operator definitions, generates corresponding proof obligations (POs), and passes them to backend verifiers, as illustrated in Figure 1. While TLAPS is an interactive proof environment that relies on users guiding the proof effort, it integrates automatic backends to discharge proof obligations that users consider trivial.

The two main backends of the current version of TLAPS are Zenon [4], a tableau prover for first-order logic and set theory, and Isabelle/TLA$^+$, a faithful encoding of TLA$^+$ in the Isabelle [13] proof assistant, which provides automated proof methods based on first-order reasoning and rewriting. The backends available prior to the work presented here also included a generic translation to the input language of SMT solvers that focused on quantifier-free formulas of linear arithmetic (not shown in Fig. 1). This SMT backend was occasionally useful because

N. Bjørner and A. Voronkov (Eds.): LPAR-18, LNCS 7180, pp. 289–303, 2012.

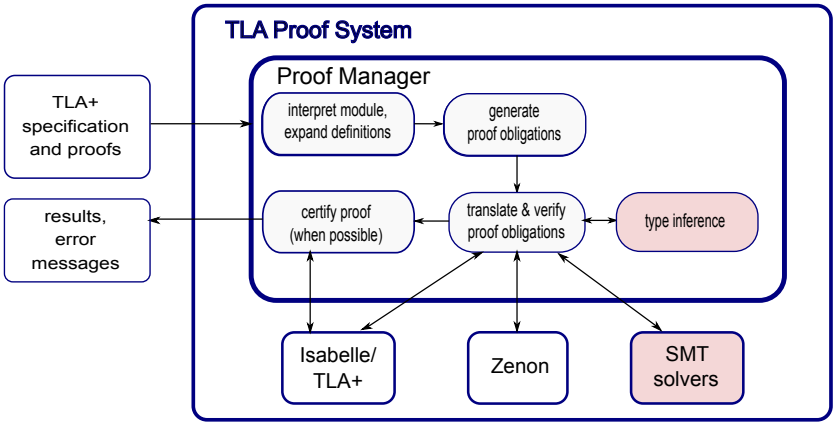


Fig. 1. General architecture of TLAPS

the other backends perform quite poorly on obligations involving arithmetic reasoning. However, it covered a rather limited fragment of TLA$^+$, which heavily relies on modeling data using sets and functions. Assertions mixing arithmetic, sets and functions arise frequently in TLA$^+$ proofs.

In the work reported here we present a new SMT-based backend for (non-temporal) TLA$^+$ formulas that encompasses set-theoretic expressions, functions, arithmetic, records, and tuples. By evaluating the performance of the backend over several existing TLA$^+$ proofs we show that it achieves good coverage for "trivial" proof obligations. The new modules comprising our backend appear shaded in Figure 1.

State of the Art and Context. Over the last years there have been several efforts to integrate interactive and automatic theorem provers (ATPs). ATP systems are satisfiability solvers for first-order logic, while the Satisfiability Modulo Theories (SMT) approach combines first-order reasoning with decision procedures for theories such as equality, integer and real arithmetic, arrays and bit-vectors. For example, Sledgehammer [3] integrates Isabelle/HOL, the encoding of polymorphic higher-order logic, with ATPs and SMT solvers. The translation to SMT is allowed to be unsound, since proof scripts produced by the solvers are reconstructed and verified in the trusted kernel of Isabelle/HOL.

TLA$^+$ is an untyped language, which makes it very expressive and flexible, but also makes automated reasoning quite challenging [11]. Since TLA$^+$ variables can assume *any* value, it is customary to start any verification project by proving a so-called *type invariant* that associates every variable of the specification with the set of values that variable may assume. Most subsequent correctness proofs rely on the type invariant. It should be noted that TLA$^+$ type invariants frequently express more sophisticated properties than what could be ensured by a decidable type system.

The input languages of state-of-the-art SMT solvers are based on many-sorted first-order logic. This allows us to design a unique translation from TLA$^+$ expressions to an intermediate language, from which the translation to the actual target languages of particular SMT solvers is straightforward. The considered languages are: (i) SMT-LIB [1], the *de facto* standard input format for SMT solvers (in our experiments we use the CVC3 solver [2] as a "baseline"), (ii) an extension of SMT-LIB for the solver Z3 [6], and (iii) the native input language of the solver Yices [8]. Using SMT-LIB as the target of our translation, TLAPS can be independent of any particular solver. Z3 adds support for datatypes, that allows us to easily encode tuples and records. On the other hand, the Yices language provides useful concepts such as sub-typing or a direct representation of tuples, records and λ-terms. The considered TLA$^+$ formulas are translated to quantified first-order formulas over the theory of linear integer arithmetic, extended with free sort and function symbols. In particular, we make heavy use of uninterpreted functions, and we do not restrict ourselves to quantifier-free formulas.

The first challenge is therefore to design a typing discipline that is compatible with the logics of SMT solvers but accommodates typical TLA$^+$ specifications. In a first step of the translation, a suitable type is assigned to every expression that appears in the proof obligation. We make use of this type assignment during the translation of expressions. For example, equality between integer expressions will be translated differently from equality between sets or functions.

Type inference may fail because not every set-theoretic expression is typable according to our typing discipline, and in this case the backend aborts. Otherwise, the proof obligation is translated to SMT formulas. Observe that type inference is relevant for the soundness of the SMT backend: a proof obligation that is unprovable according to the semantics of untyped TLA$^+$ must not become provable due to incorrect type annotation. As a trivial example, consider the formula $x + 0 = x$, which should be provable only if x is known to be of arithmetic sort. Type inference essentially relies on assumptions that are present in the proof obligation and that constrain the values of symbols (variables or operators).

Paper outline. A brief introduction to TLA$^+$ and the input languages of SMT solvers appear in the next section. A type system for TLA$^+$ together with its inference algorithm is described in Section 3, and the translation rules in Section 4. Results for some case studies and conclusions are given in Sections 5 and 6.

2 TLA$^+$ and the SMT Languages

2.1 The Non-temporal Fragment of TLA$^+$

Our backend handles non-temporal TLA$^+$ expressions, which make up the vast majority of proof obligations that arise in TLA$^+$ developments. For the purposes of this paper we fix a core subset of the language to illustrate just the main challenges, where we only include main primitive operators and constructs. This

fragment of the language, named ξ, is defined and described below by the following simplified grammar, which defines TLA$^+$ expressions ϕ, where *Id* is an identifier name for a constant, variable, record field or operator with possible arguments. Other symbols include strings and integer numbers.

$$
\begin{array}{lll}
\phi ::= & \mathit{Id} \mid \mathit{Id}(\phi, \ldots, \phi) \mid \mathit{String} \mid \mathit{Number} \mid (\phi) & \\
& \mid \textsc{true} \mid \textsc{false} \mid \textsc{boolean} \mid \mathrm{Nat} \mid \mathrm{Int} & \text{(atomic expressions)} \\
& \mid \textsc{if}\ \phi\ \textsc{then}\ \phi\ \textsc{else}\ \phi \mid \neg\phi & \text{(conditional, negation)} \\
& \mid \phi\ [\ \wedge\ \mid \in \mid \cup \mid \subseteq \mid = \mid + \mid <\ ]\ \phi & \text{(infix operators)} \\
& \mid \forall\ \mathit{Id} : \phi \mid \exists\ \mathit{Id} : \phi & \text{(quantifiers)} \\
& \mid \{\} \mid \{\phi, \ldots, \phi\} & \text{(enumerated sets)} \\
& \mid [\mathit{Id} \in \phi \mapsto \phi] \mid [\phi \to \phi] \mid \phi[\phi] \mid \textsc{domain}\ \phi & \text{(function expressions)} \\
& \mid [\mathit{Id} \mapsto \phi, \ldots, \mathit{Id} \mapsto \phi] \mid \phi.\mathit{Id} \mid \langle \phi, \ldots, \phi \rangle & \text{(records and tuples)}
\end{array}
$$

- The basic set operators consist of $\in$, $\cup$, and $\subseteq$. In TLA$^+$, equality is also an operator of set theory, since it formally means equality of sets.
- Operators on functions include function application $f[e]$, $\textsc{domain}\ f$ (domain of function f), $[x \in S \mapsto e]$ (the function f such that $f[x] = e$ for $x \in S$), and $[S \to T]$ (the set of functions f with domain S and $f[x] \in T$ for $x \in S$).
- TLA$^+$ provides the usual operators of propositional logic; our restricted fragment includes $\wedge$ and $\neg$. BOOLEAN denotes the set {TRUE, FALSE}. Quantified formulas are of the form $Qx : e$, where $Q \in \{\forall, \exists\}$.
- A record $[h_1 \mapsto e_1, \ldots, h_n \mapsto e_n]$ is a function whose domain is the finite set of strings $\{\text{“}h_1\text{”}, \ldots, \text{“}h_n\text{”}\}$. Access to record fields is written $r.h$, abbreviating $r[\text{“}h\text{”}]$, thus $[h_1 \mapsto e_1, \ldots, h_n \mapsto e_n].h_i = e_i$. Similarly, an n-tuple $\langle e_1, \ldots, e_n \rangle$ is a function whose domain is $\{1, \ldots, n\}$ and $\langle e_1, \ldots, e_n \rangle[i] = e_i$, for $1 \leq i \leq n$.
- A TLA$^+$ operator is a symbol of arity 0 or higher. Operators are associated with definitions whose expansion is controlled by the user. Expansion of defined operators is handled by the proof manager: definitions for operators that occur in proof obligations passed to backends are hidden. Operators by themselves are not expressions (they correspond to class functions in set theory), and they cannot be quantified over.
- Finally, the arithmetic operators include $+$ and $<$. Nat and Int are the sets of natural and integer numbers, respectively. We also include the construct IF/THEN/ELSE for conditional expressions.

We omit several features from this simplified description that can be introduced as syntactic sugar and that are handled by our backend, such as EXCEPT constructs for functions, and set comprehension. The extension to a larger subset of the language is straightforward following the TLA$^+$ semantics. A notable TLA$^+$ construct that is not handled by our backend is the CHOOSE primitive, known as Hilbert's ε operator. A detailed description of the full TLA$^+$ syntax and semantics can be found in [10, Chap. 16].

2.2 Input Languages of SMT Solvers

The input languages of SMT solvers are based on a many-sorted first-order logic. Accordingly, each well-formed expression has a unique sort. The languages provide syntax and commands for declaring new sort and function symbols, and for asserting formulas over the resulting signature. With each function symbol are associated the sorts of its arguments and its result sort. Terms and formulas are written in a Lisp-like language.

In particular, the SMT-LIB [1] initiative provides a common input format, as well as a repository of benchmarks, for SMT solvers. In general, an SMT input file is a sequence of declarations of sorts, functions, assumptions, and a goal. It is related with a logic, identified by a pre-established name, to which are associated sort and function declarations, and possibly syntactic and semantic restrictions. Our backend produces formulas for the SMT-LIB logic AUFLIA, which supports quantified formulas over the theory of linear integer arithmetic extended with free sort and function symbols. In this logic the predefined sorts are Bool and Int. Set theory is not currently supported natively by any pre-defined logic in SMT-LIB.

Simplified grammars for SMT-LIB sorts and terms can be given as follows:

$$\sigma ::= s \mid (s\ \sigma^+)$$
$$t\ ::= x \mid \mathit{Number} \mid (f\ t^+) \mid ([\mathsf{forall}|\mathsf{exists}]\ (((x\ \sigma))^+)\ t)$$

where σ is a sort, s is a sort identifier, t is a term, x is a variable symbol, and f is a function symbol. A sort constructor is defined by its name and the argument sorts. A function declaration is composed of the function symbol, a list of argument sorts, and the sort of the result. Constants are simply functions with no arguments. SMT-LIB provides by default a Boolean sort for terms and the standard functions and, or, not, true and false. The logic AUFLIA provides a sort for integer numbers and the arithmetic functions $+$, $-$, $*$, $/$, $<$, $<=$, $>=$ and $>$, to which we add an extra sort of arity 0 to represent the universe of TLA$^+$ constants of unspecified sort[1].

The Z3 input format is an extension of the one defined above, in particular adding algebraic datatypes that we use for representing tuples and records. The structure and syntax of the Yices input format are similar to SMT-LIB, and supports lambda expressions, tuples, and records. Boolean and integer sorts are also pre-defined, as well as a sort for natural numbers.

3 Type Inference for TLA$^+$

We define a type system for TLA$^+$ expressions that underlies our SMT translation. We consider types τ according to the following grammar:

$$\tau ::= \bot \mid \mathsf{Bool} \mid \mathsf{Str} \mid \mathsf{Nat} \mid \mathsf{Int} \mid \mathbf{P}\,\tau \mid \tau \to \tau \mid \mathsf{Rec}\,\{h_i \mapsto \tau_i\} \mid \mathsf{Tup}\,[\tau_i].$$

[1] The logic AUFLIA also provides a theory of arrays, that we do not make use of.

The atomic types are $\bot$ (terms of unspecified type), Bool (propositions), strings, and natural and integer numbers. Complex types are sets (of base type τ), functions, records (defined by a mapping from field names h_i to types) and tuples (as a fixed-size list of types). A partial order $\leq$ on types, with $\bot$ as the smallest element, is defined as the least reflexive and transitive relation that satisfies

$$\begin{array}{ll}
\bot \leq \tau & \text{for any type } \tau \\
\mathbf{P}\,\tau_1 \leq \mathbf{P}\,\tau_2 & \text{if } \tau_1 \leq \tau_2 \\
\tau_1 \to \tau_2 \leq \tau_1' \to \tau_2' & \text{if } \tau_1 \leq \tau_1' \text{ and } \tau_2 \leq \tau_2' \\
\mathsf{Rec}\,\{h_i \mapsto \tau_i\}_{i \in 1..n} \leq \mathsf{Rec}\,\{h_i \mapsto \tau_i'\}_{i \in 1..n'} & \text{if } n \leq n' \text{ and } \tau_i \leq \tau_i' \text{ for } 1 \leq i \leq n \\
\mathsf{Tup}\,[\tau_i]_{i \in 1..n} \leq \mathsf{Tup}\,[\tau_i']_{i \in 1..n} & \text{if } \tau_i \leq \tau_i' \text{ for } 1 \leq i \leq n \\
\mathsf{Nat} \leq \mathsf{Int} &
\end{array}$$

We define an inference algorithm for this type system that is based on an operator $[\![e, \varepsilon]\!]_I$ whose arguments are a TLA$^+$ expression e and an expected lower bound ε for the type of e according to the partial order. The computation either returns the inferred type or fails. The operator recurses over the structure of TLA$^+$ expressions, gathering information in a typing environment *type*, that maps each TLA$^+$ symbol to its type. Therefore, $type(x)$ is the type of symbol x.

Initially, we consider that every symbol has the unspecified type $\bot$. Recursive calls to the operator $[\![\cdot]\!]_I$ may update the type of symbols as recorded in *type* by new types that are larger than the previous ones. A type assignment is definitive only when types for all expressions in the proof obligation have been successfully inferred. For example, consider a proof obligation including two hypotheses $S = \{\}$ and $S \subseteq \mathsf{Int}$. After evaluating the first one, S will have type $\mathbf{P}\,\bot$, but it will be updated to $\mathbf{P}\,\mathsf{Int}$ when the second hypothesis is processed.

The rules of $[\![\cdot]\!]_I$ are defined in Figures 2 and 3. Before describing them, we introduce some preliminary definitions and notations. The rules are defined operationally: for example, we write $[\![\ldots]\!]_I \equiv f; g$ to indicate that f is evaluated first and the result of the overall rule is the result computed by g. We also use *if* and *case* with their usual meanings, *let* that performs pattern matching, and $:=$ for variable assignment. The base function $\mathbf{b}\,\tau$ is the dual of $\mathbf{P}\,\tau$, and is defined by $\mathbf{b}\,\mathbf{P}\,\tau = \tau$, whereas $\mathbf{b}\,\tau$ fails if τ is not a set type. The function $\mathsf{ch}(c)$ fails when condition c is false. The function $\mathsf{ret}(c(\tau))$ returns the type τ if $c(\tau)$ is satisfied, otherwise fails. For example, rule (3.6) first checks that the minimum type ε is $\bot$ or Bool, it tries to "equalize" the type of both subexpressions, and then the resulting type τ is checked to be Bool before returning it. The function max returns the greater of two comparable types as defined by

$$\max(\tau_1, \tau_2) \equiv \text{if } \tau_1 \leq \tau_2 \text{ then } \tau_2 \text{ else (if } \tau_2 \leq \tau_1 \text{ then } \tau_1 \text{ else fail)}.$$

The typing environment is updated only when evaluating symbols (rule 3.1), where $type \oplus s$ denotes the typing environment *type*, updated with the mapping s. The rules (3.8) and (3.21) introducing bound variables silently rename the variables in order to avoid any clashes with symbols already introduced. Just as any other symbol, these fresh variables are initially assigned type $\bot$. Upon recursive calls to $[\![\cdot]\!]_I$, appropriate return types are passed on to subexpressions,

Symbols and other constructs

$$[\![x, \varepsilon]\!]_I \equiv \alpha := \max(\mathit{type}(x), \varepsilon); \mathit{type} := \mathit{type} \oplus \{x \mapsto \alpha\}; \alpha \quad (3.1)$$

$$[\![\text{TRUE}, \varepsilon]\!]_I \equiv \mathsf{ch}(\varepsilon \leq \mathsf{Bool}); \mathsf{Bool} \qquad [\![\text{FALSE}, \varepsilon]\!]_I \equiv \mathsf{ch}(\varepsilon \leq \mathsf{Bool}); \mathsf{Bool} \quad (3.2)$$

$$[\![\text{BOOLEAN}, \varepsilon]\!]_I \equiv \mathsf{ch}(\varepsilon \leq \mathbf{P}\,\mathsf{Bool}); \mathbf{P}\,\mathsf{Bool} \qquad [\![\text{"..."}, \varepsilon]\!]_I \equiv \mathsf{ch}(\varepsilon \leq \mathsf{Str}); \mathsf{Str} \quad (3.3)$$

$$[\![e(e_1, \ldots, e_n), \varepsilon]\!]_I \equiv \text{let } \alpha_1 = [\![e_1, \bot]\!]_I, \ldots, \alpha_n = [\![e_n, \bot]\!]_I \text{ in}$$
$$\text{let } (\_ \to \ldots \to \_ \to \alpha_{n+1}) = [\![e, \alpha_1 \to \ldots \to \alpha_n \to \varepsilon]\!]_I \text{ in } \alpha_{n+1} \quad (3.4)$$

$$[\![\text{IF } p \text{ THEN } e_1 \text{ ELSE } e_2, \varepsilon]\!]_I \equiv \mathsf{ch}([\![p, \mathsf{Bool}]\!]_I = \mathsf{Bool}); \mathsf{eq}([e_1, e_2], \varepsilon) \quad (3.5)$$

Logic

$$[\![e_1 \wedge e_2, \varepsilon]\!]_I \equiv \mathsf{ch}(\varepsilon \leq \mathsf{Bool}); \mathsf{eq}([e_1, e_2], \mathsf{Bool}); \mathsf{Bool} \quad (3.6)$$

$$[\![\neg e, \varepsilon]\!]_I \equiv \mathsf{ch}(\varepsilon \leq \mathsf{Bool}); [\![e, \mathsf{Bool}]\!]_I; \mathsf{Bool} \quad (3.7)$$

$$[\![Q\ x : e, \varepsilon]\!]_I \equiv \mathsf{ch}(\varepsilon \leq \mathsf{Bool}); [\![e, \mathsf{Bool}]\!]_I; \mathsf{Bool} \quad \text{for } Q \in \{\forall, \exists\} \quad (3.8)$$

Arithmetic

$$[\![e_1 + e_2, \varepsilon]\!]_I \equiv \mathsf{ch}(\varepsilon \leq \mathsf{Int}); \alpha := \mathsf{eq}([e_1, e_2], \varepsilon); \mathsf{ret}(\alpha \in \{\mathsf{Nat}, \mathsf{Int}\}) \quad (3.9)$$

$$[\![e_1 < e_2, \varepsilon]\!]_I \equiv \mathsf{ch}(\varepsilon \leq \mathsf{Bool}); \mathsf{ch}(\mathsf{eq}([e_1, e_2], \mathsf{Nat}) \leq \mathsf{Int}); \mathsf{Bool} \quad (3.10)$$

$$[\![n, \varepsilon]\!]_I \equiv \mathsf{ch}(\varepsilon \leq \mathsf{Int}); \mathsf{ret}(\mathsf{Nat} \leq \varepsilon) \quad \text{(where } n \text{ is a number)} \quad (3.11)$$

$$[\![\mathsf{Nat}, \varepsilon]\!]_I \equiv \mathsf{ch}(\varepsilon \leq \mathbf{P}\,\mathsf{Int}); \mathsf{ret}(\mathbf{P}\,\mathsf{Nat} \leq \varepsilon) \qquad [\![\mathsf{Int}, \varepsilon]\!]_I \equiv \mathsf{ch}(\varepsilon \leq \mathbf{P}\,\mathsf{Int}); \mathbf{P}\,\mathsf{Int} \quad (3.12)$$

Sets

$$[\![e_1 = e_2, \varepsilon]\!]_I \equiv \mathsf{ch}(\varepsilon \leq \mathsf{Bool}); \mathsf{eq}([e_1, e_2], \bot); \mathsf{Bool} \quad (3.13)$$

$$[\![S \subseteq T, \varepsilon]\!]_I \equiv \mathsf{ch}(\varepsilon \leq \mathsf{Bool}); \mathsf{eq}([S, T], \mathbf{P}\,\bot); \mathsf{Bool} \quad (3.14)$$

$$[\![e_1 \in e_2, \varepsilon]\!]_I \equiv [\![\{e_1\} \subseteq e_2, \varepsilon]\!]_I \quad (3.15)$$

$$[\![S \cup T, \varepsilon]\!]_I \equiv \max(\mathbf{P}\,\bot, \mathsf{eq}([S, T], \varepsilon)) \quad (3.16)$$

$$[\![\{\}, \varepsilon]\!]_I \equiv \max(\mathbf{P}\,\bot, \varepsilon) \quad (3.17)$$

$$[\![\{e_1, \ldots, e_n\}, \varepsilon]\!]_I \equiv \mathbf{P}\,\mathsf{eq}([e_1, \ldots, e_n], \mathbf{b}\varepsilon) \quad (3.18)$$

Functions

$$[\![f[e], \varepsilon]\!]_I \equiv \alpha := [\![e, \bot]\!]_I; \text{let } (\alpha' \to \beta) = [\![f, \alpha \to \varepsilon]\!]_I \text{ in } (\mathsf{ch}(\alpha = \alpha'); \beta) \quad (3.19)$$

$$[\![\text{DOMAIN } f, \varepsilon]\!]_I \equiv \text{let } (\alpha \to \_) = [\![f, \mathbf{b}\varepsilon \to \bot]\!]_I \text{ in } \mathbf{P}\,\alpha \quad (3.20)$$

$$[\![[x \in S \mapsto e], \varepsilon]\!]_I \equiv \text{case } \varepsilon \text{ of } | \ \alpha \to \beta : \ \mathbf{b}[\![S, \mathbf{P}\,\alpha]\!]_I \to [\![e, \beta]\!]_I$$
$$| \ \bot : \ \mathbf{b}[\![S, \mathbf{P}\,\bot]\!]_I \to [\![e, \bot]\!]_I \quad (3.21)$$

$$[\![[S \to T], \varepsilon]\!]_I \equiv \text{case } \varepsilon \text{ of } | \ \mathbf{P}\,(\alpha \to \beta) : \mathbf{P}\,(\mathbf{b}[\![S, \mathbf{P}\,\alpha]\!]_I \to \mathbf{b}[\![T, \mathbf{P}\,\beta]\!]_I)$$
$$| \ \bot : \ \mathbf{P}\,(\mathbf{b}[\![S, \mathbf{P}\,\bot]\!]_I \to \mathbf{b}[\![T, \mathbf{P}\,\bot]\!]_I) \quad (3.22)$$

Fig. 2. Rules for the type inference operator $[\![\cdot]\!]_I$

Records and Tuples

$$[\![r.h, \varepsilon]\!]_I \equiv \text{let } \mathsf{Rec}\,\{\ldots, h \mapsto \alpha, \ldots\} = [\![r, \mathsf{Rec}\,\{h \mapsto \varepsilon\}]\!]_I \text{ in } \alpha \tag{3.23}$$

$$[\![t[i], \varepsilon]\!]_I \equiv \text{let } \mathsf{Tup}\,[\ldots, \alpha_i, \ldots] = [\![t, \bot]\!]_I \text{ in } \max(\alpha_i, \varepsilon) \tag{3.24}$$

$$\begin{aligned} [\![[h_1 \mapsto e_1, \ldots, h_n \mapsto e_n], \varepsilon]\!]_I \equiv \text{case } \varepsilon \text{ of } &\mid \mathsf{Rec}\,\{h_i \mapsto \varepsilon_i\} : \mathsf{Rec}\,\{h_i \mapsto [\![e_i, \varepsilon_i]\!]_I\} \\ &\mid \bot : \quad\; \mathsf{Rec}\,\{h_i \mapsto [\![e_i, \bot]\!]_I\} \end{aligned} \tag{3.25}$$

$$\begin{aligned} [\![\langle e_1, \ldots, e_n\rangle, \varepsilon]\!]_I \equiv \text{case } \varepsilon \text{ of } &\mid \mathsf{Tup}\,[\varepsilon_i] : \quad\; \mathsf{Tup}\,[[\![e_i, \varepsilon_i]\!]_I] \\ &\mid \bot : \quad\; \mathsf{Tup}\,[[\![e_i, \bot]\!]_I] \end{aligned} \tag{3.26}$$

Fig. 3. Rules for the type inference operator $[\![\cdot]\!]_I$ (continued)

propagating the type information through the formula. The type information associated with a symbol x is updated to a larger type when so required by the expected minimum type ε.

The operator $[\![\cdot]\!]_I$ assigns types to complex expressions based on the types of their constituents. Although expressions such as $\langle a\rangle \cup 0$ or $3 + \text{TRUE}$ appear silly, they are allowed in TLA$^+$, yet their meaning is unknown. Our fragment rules out such expressions by enforcing a typing discipline that requires subexpressions to have types compatible with the larger expression.

When type inference succeeds on a proof obligation, the typing environment *type* will contain the resulting final type assignments. There are two reasons why the inference algorithm may fail: (1) The expected type ε or the type obtained from subexpressions can be incompatible with the type associated with primitive operators, as in the examples given above. The inference rules check for this kind of mismatch using the operators ch and ret. (2) Type inference can fail to solve a constraint stating that two or more expressions need to be of the same type, as we discuss next.

The sorting discipline of SMT solvers requires in several cases that subexpressions of a TLA$^+$ expressions be assigned the same type. This is in particular true for the expressions e_1 and e_2 in $e_1 = e_2$, $e_1 \subseteq e_2$, IF p THEN e_1 ELSE e_2 (rules 3.13, 3.14, 3.5); the second of these expressions moreover requires e_1 and e_2 to be of set type. Arithmetic expressions (rules 3.9-3.10) and set operators (3.16 and 3.18) pose similar constraints. The expression $e_1 \in e_2$ requires e_2 to be of type $\mathbf{P}\,\alpha$ and e_1 of type α (rule 3.15), and $f[e]$ requires f to be of type $\alpha \to \alpha'$, and e of type α (rule 3.19), for some types α, α'. Similarly, we do not allow different applications of the same function or operator symbol to return values of different types. For those cases, the type inference rules make use of the function $\mathsf{eq}([e_1, \ldots, e_n], \varepsilon) : \tau$, that given a list of expressions $e_1, \ldots, e_n$ and an expected type ε, returns the common type of all expressions e_i (bounded below by ε), or fails if no such type can be assigned.

Type inference proceeds in three steps, that all rely on (variants of) the operator $[\![\cdot]\!]_I$. We will explain the algorithm using a proof obligation whose hypotheses are $x \in S$ and $S \subseteq \mathsf{Nat}$ and whose conclusions are $x+0 = x$ and $y \cup \{\} = y$. In the

first step, the algorithm computes an approximate type assignment for the proof obligation by applying the operator $[\![\cdot]\!]_I^{safe}$, which differs from $[\![\cdot]\!]_I$ by restricting types to *safe* types defined by the grammar $\tau_s ::= \bot \mid \mathbf{P}\,\tau_s$ (the rules of Figs. 2 and 3 are adapted accordingly). In other words, this step only distinguishes between elementary values and sets, and ensures that all symbols that appear in the proof obligation are used consistently according to these categories. In our running example, it infers types $\bot$ for the symbol x, and $\mathbf{P}\,\bot$ for y and S.

The second step refines this type assignment by running the operator $[\![\cdot]\!]_I$ on "typing hypotheses", i.e. available facts of the forms

$$x \otimes e \qquad \text{and} \qquad \forall a_1 \in S_1, \ldots, a_n \in S_n : x(a_1, \ldots, a_n) \otimes e,$$

for $\otimes \in \{=, \in, \subseteq\}$, starting from the typing environment computed during the first step. In these expressions, x is a constant, variable or operator, and e is an expression whose type can already be inferred.[2] These typing hypotheses are obtained by decomposing the assumptions present in a proof obligation by elementary heuristics. In our example, we have the typing hypotheses $x \in S$ and $S \subseteq \mathsf{Nat}$, and the previously inferred types for x and S will be refined to Nat and $\mathbf{P}\,\mathsf{Nat}$. The reason to perform this step on a restricted set of facts is to avoid the evaluation of $[\![\cdot]\!]_I$ on hypotheses such as $z \notin Nat$, which would incorrectly assign type Nat to z.

The third step ensures that the entire proof obligation can be typed using the typing environment computed in the first two steps. It does so by applying the operator $[\![o]\!]_I^{check}$, which differs from $[\![\cdot]\!]_I$ in that the rule 3.1 for symbols is defined as $[\![x, \varepsilon]\!]_I \equiv \mathsf{ch}(\varepsilon \leq type(x))$. In particular, the typing environment is not updated. This step will succeed for our running example, given the previously assumed typing hypotheses for x. (No typing hypothesis is needed for y.) However, it would fail without an appropriate typing hypothesis because the subexpression $x + 0$ requires x to be of arithmetic type, not $\bot$.

Similarly, consider the proof obligation $(\neg\neg P) = P$. While we may infer that $\neg\neg P$ is Boolean, the typing rule for equality requires that the expressions on both sides must be of equal type. However, the type of P inferred during the first step is just $\bot$. If we allowed the algorithm to assign Bool as the type of P the above proof obligation could be proved without any hypotheses – but its instance $(\neg\neg 42) = 42$ should not be provable in TLA$^+$. The soundness of the type inference algorithm is asserted by the following proposition.

Proposition 1. *Assume given a* TLA$^+$ *proof obligation o for which type inference succeeds. Then for any expression e occurring in the obligation, to which the algorithm assigns a non-safe type τ, we have $\Gamma \vdash e \in [\![\tau]\!]_{\mathrm{TLA}^+}$ where Γ denotes the set of typing hypotheses of o and $[\![\tau]\!]_{\mathrm{TLA}^+}$ denotes the* TLA$^+$ *expression representing the set of values of type τ.*

Proof (idea). The first step of the algorithm assigns only safe types, so there is nothing to prove. Note that semantically, safe types correspond to the universe

[2] For variables, facts of this kind usually come from the type invariant. Our backend requires similar type-correctness lemmas for operators.

of all TLA$^+$ values, so that step cannot introduce any unsoundness. However, if it fails, the proof obligation cannot be represented in the multi-sorted logic of SMT solvers. The assertion for the second and third steps is proved by induction on the expression e, using the rules of Figs. 2 and 3. QED

4 From TLA$^+$ to SMT

Once a type assignment is determined for the symbols in a TLA$^+$ proof obligation, it can be translated to the input languages of SMT solvers. This is done in two steps. In a first phase, the proof obligation is pre-processed to eliminate expressions that are not directly available in SMT, such as set operators or function expressions. The resulting formula will contain only TLA$^+$ expressions that have a direct representation in the first-order logic of SMTs, namely, the logical and arithmetic operators and the IF/THEN/ELSE construct. These are called *basic* expressions in the language ξ_b. The translation to our target languages – SMT-LIB, Yices and Z3 – is then just a syntactic rewriting.

The operator $[\![\cdot]\!]_B : \xi \to \xi_b$ transforms TLA$^+$ expressions to basic expressions, using the type information gathered previously. During this transformation, we temporarily introduce λ-terms to represent non-basic expressions such as set or function operators. We will prove that all λ-terms introduced during the translation of a well-typed TLA$^+$ expression can be β-reduced to an expression in ξ_b, which no longer contains λ-expressions.

Sets are encoded by their characteristic predicate, allowing for the direct translation of the set membership relation. Set of sets are not considered for this translation, in order to stay within the realm of first-order logic. Any hypotheses of a proof obligation that fall outside this class are discarded. For example, hypotheses of the form $S \in T$ where T is of type $\mathbf{PP}\,\tau$ are useful during type inference in order to determine the type of S but are then dropped during the translation.

A similar translation for sets in Event-B was given by Déharbe [7], who also considers alternative representations of sets, such as via arrays or using a finite axiomatization of ZF set theory. Hence, if S is an expression of type $\mathbf{P}\,\tau$ then $[\![S]\!]_B$ is a λ-abstraction, and $[\![e \in S]\!]_B \equiv [\![S]\!]_B([\![e]\!]_B)$. Elementary sets are represented as uninterpreted functions. The following rules indicate the translation of more complex set expressions; we simplify the presentation of the rules by omitting the type annotations.

$$[\![S \cup T]\!]_B \equiv \lambda x.\ [\![x \in S \vee x \in T]\!]_B \tag{4.1}$$

$$[\![S \subseteq T]\!]_B \equiv [\![\forall x : x \in S \Rightarrow x \in T]\!]_B \tag{4.2}$$

$$[\![\{\}]\!]_B \equiv \lambda x.\ \text{FALSE} \tag{4.3}$$

$$[\![\{e_1, \ldots, e_n\}]\!]_B \equiv \lambda x.\ [\![x = e_1 \vee \ldots \vee x = e_n]\!]_B \tag{4.4}$$

$$[\![\mathsf{Nat}]\!]_B \equiv \lambda x.\ x \geq 0 \tag{4.5}$$

$$[\![\mathsf{Int}]\!]_B \equiv \lambda x.\ \text{TRUE} \tag{4.6}$$

Translation of equality depends on the type of the two sub-expressions, which must be equal because of typing rule 3.13.

$$\begin{aligned}
[\![e_1 = e_2]\!]_B \equiv\ & \text{case } [\![e_1, \bot]\!]_I \text{ of} \\
& \mid \mathbf{P}\ _ : \quad [\![\forall x : x \in e_1 \Leftrightarrow x \in e_2]\!]_B \\
& \mid _ \to _ : [\![\text{DOMAIN } e_1 = \text{DOMAIN } e_2 \\
& \qquad\qquad \wedge\ \forall x : x \in \text{DOMAIN } e_1 \Rightarrow e_1[x] = e_2[x]]\!]_B \\
& \mid _ : \quad [\![e_1]\!]_B = [\![e_2]\!]_B
\end{aligned} \tag{4.7}$$

Similarly to set membership, function application reduces to λ-aplication (rule 4.8). A function $[x \in S \mapsto e]$ is translated to $\lambda y.\ [\![e(x \leftarrow y)]\!]_B$ (rule 4.9), where x is replaced by y in the expression e (the domain S is represented separately, as explained later).

$$[\![f[e]]\!]_B \equiv [\![f]\!]_B([\![e]\!]_B) \qquad (\lambda\text{-application}) \tag{4.8}$$

$$[\![[x \in S \mapsto e]]\!]_B \equiv \lambda y.\ [\![e(x \leftarrow y)]\!]_B \tag{4.9}$$

$$[\![[S \to T]]\!]_B \equiv \lambda f.\ [\![S = \text{DOMAIN } f \wedge \forall x \in S : f[x] \in T]\!]_B \tag{4.10}$$

$$[\![\text{DOMAIN } f]\!]_B \equiv [\![dom(f)]\!]_B \qquad (\text{when } f \text{ is a symbol})$$

$$[\![\text{DOMAIN } [x \in S \mapsto e]]\!]_B \equiv [\![S]\!]_B \tag{4.11}$$

The translation of function or operator symbols is guided by their types. In case of atomic type, they are simply represented by symbols of appropriate type declared in the SMT output. An n-ary operator or function that returns a set of individuals is represented as an $(n+1)$-ary characteristic predicate. For example, a function symbol $f : \mathsf{Int} \to \mathbf{P}\,\mathsf{Int}$ will be encoded by a binary predicate f over integers.

Because SMT functions have no notion of function domain other than their argument type(s), we associate with each function or operator symbol f a set DOMAIN f. We maintain a mapping $dom : Id \mapsto \xi$ that associates symbols with their domains. Domains of operators are extracted from the corresponding typing hypotheses. For every function or operator application that occurs in the proof obligation, we check that the argument values are in the domain: otherwise the value of the application would be unspecified. To this end, we define an auxiliary operator $[\![\cdot]\!]_F : \xi \to \xi$ that computes corresponding proof obligations. It maintains the structure of the original proof obligation, preserving the quantified variables and conditionals, and collects all function or operator applications that occur in the formula. In particular, we define the following rules.

$$\begin{aligned}
[\![f[e]]\!]_F &\equiv [\![f]\!]_F \wedge [\![e]\!]_F \wedge e \in \text{DOMAIN } f \\
[\![f(e)]\!]_F &\equiv [\![f]\!]_F \wedge [\![e]\!]_F \wedge e \in dom(f) \\
[\![\forall x : e]\!]_F &\equiv \forall x : [\![e]\!]_F \qquad\qquad [\![e_1 \wedge e_2]\!]_F \equiv [\![e_1]\!]_F \wedge [\![e_2]\!]_F \quad \text{etc.} \\
[\![[x \in S \mapsto e]]\!]_F &\equiv \forall x : [\![x \in S \Rightarrow e]\!]_F
\end{aligned}$$

For compound expressions other than logical formulas, the operator $[\![\cdot]\!]_F$ recurses on all subexpressions. For example, $[\![[e_1 \subseteq e_2]]\!]_F \equiv [\![e_1]\!]_F \wedge [\![e_2]\!]_F$. For atomic expressions x, we define $[\![x]\!]_F \equiv \text{TRUE}$.

It can be shown that given a well-typed TLA$^+$ expression e from the fragment ξ, all λ-terms that occur in its translation $[\![e]\!]_B$ can be β-reduced, as stated by the following proposition.

Proposition 2. *Given a well-typed* TLA$^+$ *expression from the fragment* ξ *that contains only sets of individuals and functions whose arguments are atomic types, then the* β*-normal form of* $[\![e]\!]_B$ *does not contain* λ*-terms.*

Proof (idea). The translation of expressions $e \in S$ and $f[e]$ introduces function applications that, due to the assumption on the types of TLA$^+$ expressions that appear in the input, remove any λ's introduced during the translation. The only atomic formulas that directly involve set or function types are $S \subseteq T$, $S = T$, and $f = g$, for sets S and T and functions f and g, and in these cases the translation introduces explicit quantifiers that provide the required function arguments for β-reduction. QED

After transforming a proof obligation o in ξ to a basic expression, $[\![o]\!]_B$ is ready to be translated to the input format of SMT solvers. Purely arithmetic and first-order expressions are translated to the corresponding built-in operators of the target languages. For example, the basic expression $e_1 + e_2$ (where e_1 and e_2 must be of arithmetic type because of type checking) is translated to SMT-LIB as $(+\ e_1\ e_2)$ and $\forall x : e$ as $(\mathsf{forall}\ ((\mathsf{x}\ [\![\tau_x]\!]_S))\ e)$, where x is a fresh identifier, τ_x is the type of x as determined by type inference, and where $[\![\cdot]\!]_S$ translates a type to an SMT sort.

The SMT-LIB, Yices, and Z3 backends mainly differ in the encoding of tuples and records. SMT-LIB currently does not have a pre-defined theory for these types, whereas Yices supports them natively, and the Z3 extension of SMT-LIB provides algebraic data types. These kinds of expressions are therefore translated differently for each particular solver format. Currently, only constituents of tuples and records of atomic types are allowed.

In the Yices format, the encoding of records and tuples is almost verbatim. For example, the TLA$^+$ record $[h_1 \mapsto e_1, h_2 \mapsto e_2]$ is translated to $(\mathsf{mk\text{-}record}\ h_1{::}e_1\ h_2{::}e_2)$ and the expression $r.h$ corresponds to $(\mathsf{select}\ r\ h)$. The type $\mathsf{Rec}\,\{h_1 \mapsto \tau_1, h_2 \mapsto \tau_2\}$ is represented as the sort $(\mathsf{record}\ h_1 :: \tau_1\ h_2 :: \tau_2)$. The translation of tuples is analogous, with indexes taking the place of record field names.

For every record type $r = \mathsf{Rec}\,\{h_1 \mapsto \tau_1, \ldots, h_n \mapsto \tau_n\}$, the Z3 backend declares the data type

$$(\mathsf{record\text{-}sort}_r\ (\mathsf{mk\text{-}record}_r\ (\mathsf{h}_1\ \tau_1)\ \ldots\ (\mathsf{h}_n\ \tau_n))$$

that introduces the new sort identifier $\mathsf{record\text{-}sort}_r$, the datatype constructor $\mathsf{mk\text{-}record}_r$, and the selector function h_i with their corresponding types. Record construction and selection are then translated according to the following rules (the operator $[\![\cdot]\!]_T : \xi \to \text{SMT}$ represents the translation function to SMT format.)

$$[\![[h_1 \mapsto e_1, h_2 \mapsto e_2]]\!]_T \equiv \lambda r.[\![r = \mathsf{mk\text{-}record}_r(e_1, e_2)]\!]_T \tag{4.12}$$

$$[\![r.h]\!]_T \equiv [\![h(r)]\!]_T \tag{4.13}$$

In the SMT-LIB backend, records are axiomatized as follows. For each record sort r that occurs in the proof obligation, we declare a new sort $\mathsf{record\text{-}sort}_r$ of arity 0. The record constructor and the selector functions are declared separately as uninterpreted functions with the appropriate sorts. The translation rules are the same as above (4.12 and 4.13). The logical connection between the constituents with their function selectors and the constructor are asserted for each new declared datatype by the axioms

$$\forall x_1 : \tau_1, \ldots, x_n : \tau_n.\ x_i = \mathsf{h}_i(\mathsf{mk\text{-}record}_r\ x_1\ \ldots\ x_n) \qquad \text{for } 1 \leq i \leq n.$$

5 Experimental Results

We have used our new backend with good success on several examples that had previously been proved interactively using TLAPS. In particular, we show the results for two cases in the following table. For each benchmark, we indicate the size (number of lines) of the interactive proof, the time (in seconds) required to verify that proof on a standard laptop, as well as the corresponding figures when parts of the proof are performed using the SMT backend, in its three flavors.

	Original		SMT-LIB/CVC3		Yices		Z3	
	size	time	size	time	size	time	size	time
Bakery	398	24	7	33	76	11	7	5
Memoir	2381	53	208	7	208	5	208	7

The first example concerns the invariant proof for (an atomic version of) the well-known N-process Bakery algorithm [9], which mainly uses set theory, functions and arithmetic over the natural numbers. It could be reduced from almost 400 lines of interactive proof to a completely automatic proof. The resulting obligation generates SMT formulas containing 105 quantifiers (many of them nested), which could be proved by the CVC3 SMT solver in around 33 seconds and by Z3 in 5 seconds. On the other hand, Yices could not handle the entire proof obligation at once, and it was necessary to split the theorem into separate cases per subaction; it then takes about 11 seconds to prove the resulting obligations.

More interestingly, the backend could handle significant parts of the type and safety invariant proofs of the Memoir security architecture [12], a generic framework for executing modules of code in a protected environment. The proofs were almost fully automated, except for three sub-proofs that required manual Skolemization of second-order quantifiers. In terms of lines of proof, they were reduced to around 10% of the original size. In particular, the original 2381 lines of proof for the complete type invariant theorems were reduced to 208 lines. Our three solvers took between 5 and 7 seconds to prove them.

These encouraging results show that significant automation can be gained by using SMT solver for the verification of standard TLA$^+$ models, without adapting these models to the SMT backend. There are, however, certain proof

obligations that cannot be translated and on which the backend fails. Such examples typically involve the use of advanced set-theoretic constructs, or even just sets of sets, which cannot be encoded in first-order logic using characteristic predicates. For example, our backend cannot prove

$$\forall S \in T : S \neq \{\} \wedge (\forall x \in S : P(x)) \Rightarrow \exists x \in S : P(x).$$

In simple cases such as this one, it suffices to Skolemize the outermost quantifier: the backend will then discard the irrelevant hypothesis $S \in T$ that cannot be translated.

Another source of failures is the use of TLA$^+$ operators that accept arguments of different SMT sorts and that cannot be type checked according to our typing discipline. Fortunately, such cases appear rarely in actual specifications.

6 Conclusions

We defined a translation of certain TLA$^+$ proof obligations to the input language of state-of-the-art SMT solvers. The translation relies on imposing a typing discipline on the untyped specification language TLA$^+$, and is based on a corresponding type inference algorithm. This discipline restricts the class of TLA$^+$ expressions that can be translated. Nevertheless, a significant fragment of the source language can be handled. In particular, we support first-order logic, elementary set theory, functions, integer and real arithmetic, records and tuples. Sets and functions are represented as lambda-abstractions, which works quite efficiently but excludes handling second-order expressions involving, for example, sets of sets. The translation of records and tuples relies on an axiomatization for SMT-LIB, and on appropriate native constructs of Yices and Z3. Our type inference and translation algorithms provide the formal basis for the implementation of an SMT-based backend prover for TLAPS. Universal set quantifiers that occur at the outermost level can easily be removed by the user of TLAPS, by introducing Skolem constants. An automatic pre-processing of such terms would further improve the backend.

In future work, we intend to study the question of interpreting proofs that many SMT solvers can produce for reconstructing them (as well as the type assignment) in the trusted object logic of Isabelle/TLA$^+$. This would allow us to check the results of these solvers, as well as of the translation from TLA$^+$ into SMT input, and would raise the confidence in the SMT backend, just as currently TLAPS can check proofs produced by Zenon.

We also envisage extending our translation to support λ-abstractions (for functions as basic terms) using, for example, combinators, and to support some more advanced set-theoretic constructions, perhaps using a different representation of sets.

Acknowledgements. Denis Cousineau, Damien Doligez, and Leslie Lamport provided constructive feedback on the design and implementation of the SMT backend. Helpful comments from the anonymous referees are gratefully acknowledged.

References

1. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB standard: Version 2.0. In: Gupta, A., Kroening, D. (eds.) Satisfiability Modulo Theories (SMT 2010), Edinburgh, UK (2010), http://www.SMT-LIB.org
2. Barrett, C.W., Tinelli, C.: CVC3. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 298–302. Springer, Heidelberg (2007)
3. Blanchette, J.C., Böhme, S., Paulson, L.C.: Extending Sledgehammer with SMT Solvers. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS, vol. 6803, pp. 116–130. Springer, Heidelberg (2011)
4. Bonichon, R., Delahaye, D., Doligez, D.: Zenon: An Extensible Automated Theorem Prover Producing Checkable Proofs. In: Dershowitz, N., Voronkov, A. (eds.) LPAR 2007. LNCS (LNAI), vol. 4790, pp. 151–165. Springer, Heidelberg (2007)
5. Chaudhuri, K., Doligez, D., Lamport, L., Merz, S.: Verifying Safety Properties with the TLA$^+$ Proof System. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS, vol. 6173, pp. 142–148. Springer, Heidelberg (2010)
6. de Moura, L., Bjørner, N.S.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
7. Déharbe, D.: Automatic Verification for a Class of Proof Obligations with SMT-Solvers. In: Frappier, M., Glässer, U., Khurshid, S., Laleau, R., Reeves, S. (eds.) ABZ 2010. LNCS, vol. 5977, pp. 217–230. Springer, Heidelberg (2010)
8. Dutertre, B., de Moura, L.: The Yices SMT solver. Tool Paper (2006), http://yices.csl.sri.com/tool-paper.pdf
9. Lamport, L.: A new solution of Dijkstra's concurrent programming problem. Communications of the ACM 17(8), 453–454 (1974)
10. Lamport, L.: Specifying Systems: The TLA$^+$ Language and Tools for Hardware and Software Engineers. Addison-Wesley, Boston (2002)
11. Lamport, L., Paulson, L.C.: Should your specification language be typed? ACM Trans. Prog. Lang. Syst. 21(3), 502–526 (1999)
12. Parno, B., Lorch, J.R., Douceur, J.R., Mickens, J., McCune, J.M.: Memoir: Practical state continuity for protected modules. In: IEEE Symp. Security and Privacy, Berkeley, California, U.S.A., 2011. IEEE Computer Society. Formal Specifications and Correctness Proofs: Tech. Report, Microsoft Research (February 2011)
13. Wenzel, M., Paulson, L.C., Nipkow, T.: The Isabelle Framework. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 33–38. Springer, Heidelberg (2008)

An Asymptotically Correct Finite Path Semantics for LTL

Andreas Morgenstern, Manuel Gesell, and Klaus Schneider

Embedded Systems Group, Department of Computer Science,
University of Kaiserslautern
P.O. Box 3049
67653 Kaiserslautern, Germany
{morgenstern,gesell,schneider}@cs.uni-kl.de
http://es.cs.uni-kl.de

Abstract. Runtime verification of temporal logic properties requires a definition of the truth value of these properties on the finite paths that are observed at runtime. However, while the semantics of temporal logic on infinite paths has been precisely defined, there is not yet an agreement on the definition of the semantics on finite paths. Recently, it has been observed that the accuracy of runtime verification can be improved by a 4-valued semantics of temporal logic on finite paths. However, as we argue in this paper, even a 4-valued semantics is not sufficient to achieve a semantics on finite paths that converges to the semantics on infinite paths. To overcome this deficiency, we consider in this paper Manna and Pnueli's temporal logic hierarchy consisting of safety, liveness (guarantee), co-Büchi (persistence), and Büchi (recurrence) properties. We propose the use of specialized semantics for each of these subclasses to improve the accuracy of runtime verification. In particular, we prove that our new semantics converges to the infinite path semantics which is an important property that has not been achieved by previous approaches.

1 Introduction

Runtime verification aims at detecting faults of a system by monitoring its input/output behavior during runtime. For the specification of the desired behavior, temporal logics in general, and linear temporal logic (LTL) in particular, proved to be convenient formalisms to precisely and conveniently determine complex temporal properties. During the last two decades, many model-checking procedures for temporal logics have been developed that improved the efficiency to become interesting for practical use. As a consequence, the PSL logic (extending LTL) became now an industry standard that is used by many tools and programming languages. Since temporal logics are therefore well-established, it is natural to use them also for runtime verification.

However, while the tools used to solve the model-checking problem refer to the original LTL semantics that is given for infinite behaviors, runtime verification can only reason about the finite behavior that has been observed up to a considered point of time. Whether a fault occurred at runtime can therefore not be decided by the existing LTL semantics, and instead one has to consider the meaning of LTL formulas on finite paths.

N. Bjørner and A. Voronkov (Eds.): LPAR-18, LNCS 7180, pp. 304–319, 2012.

While the semantics of LTL on infinite paths has been precisely defined in the literature (without producing alternatives), there is not yet such a consensus on the meaning of LTL properties on finite paths. Several two-valued semantics for LTL on finite paths have been proposed [6] that are well-suited for safety properties. In [1], special reset and abort operators have been added to LTL to cope with finite path semantics, but these do not solve the problem for the other operators. Recently, it has been observed that at least a three-valued semantics is necessary to give informative results [14,2,13]. Using three-valued semantics, a property evaluates to true or false whenever the truth value defined by the LTL semantics on infinite paths is already determined by its finite prefix. If the finite prefix does not determine the truth value, an inconclusive result is obtained by a third truth value. In the first case, the considered prefix is called a *good* prefix, otherwise it is called a *bad* prefix. This scheme is well-suited for pure *safety* properties like $\mathsf{G}p$ and simple liveness (guarantee) properties like $\mathsf{F}p$. Indeed, it has been observed in [8] that the only properties for which a three-valued semantics gives satisfactory results consists of boolean combinations of safety and guarantee properties which form the obligation properties in the temporal logic hierarchy of Manna and Pnueli [5]. For this reason, the formulas in this class have been already called prefix properties in [15,16].

However, there are many properties which can not be dealt with such a three-valued semantics: consider e. g. the request/acknowledge property $\mathsf{G}(r \rightarrow \mathsf{F}a)$ taken from [2] that states that every request is finally acknowledged. Finite paths cannot decide the truth of this property since it belongs to the Büchi (recurrence) class, but not to the prefix (obligation) class. Hence a three-valued logic will always evaluate to an inconclusive result. In [3], previously proposed semantics for LTL are compared with each other and a new four-valued semantics for LTL on finite paths was proposed that is argued to overcome these problems. For the request/acknowledge property, the proposed RV-LTL semantics yields value $\top_P$ (meaning *'presumable good'*) whenever the so-far read finite input path ends at a point of time where a holds. The value $\bot_P$ (meaning *'presumably bad'*) is used whenever the so-far read finite input path ends at a point of time where r holds, indicating that is is likely that the specification remains unsatisfied.

While the proposed solution gives a reasonable result for the above mentioned request/acknowledge property, we argue that for other interesting properties, the proposed RV-LTL semantics gives misleading results: For example, consider the property $\mathsf{FG}p_1 \vee \mathsf{FG}p_2$. This property states that from a certain point on, either always p_1 or always p_2 holds (note its equivalence to $\mathsf{F}(\mathsf{G}p_1 \vee \mathsf{G}p_2)$). For the behavior $p_1, p_2, p_1, p_2, \ldots$, RV-LTL determines the value $\top_P$ (presumably good) for every finite prefix, indicating the misleading result that the property is 'presumably true' while the property is not satisfied on the infinite behavior.

In this paper, we therefore define a new semantics of LTL on finite paths to improve the previously proposed semantics so that the definition on finite paths converges to the definition of infinite paths. To this end, we consider the temporal logic hierarchy of Manna and Pnueli [5,16]. Instead of distinguishing between *presumably good* and *presumably bad* in case no definitive answer is possible, we use truth values that are more specialized to the unknown infinite suffix. For example, a persistence (co-Büchi) property like $\mathsf{FG}p_1 \vee \mathsf{FG}p_2$ is evaluated

over a four-valued semantics with the truth values $\{\mathsf{true}, \mathsf{false}, \top_{\mathsf{FG}}, \bot_{\mathsf{FG}}\}$ with the intuition that whenever an infinite path satisfies the property from a certain point of time on, we assign $\top_{\mathsf{FG}}$ for the corresponding prefixes of that path from that point on. On the other hand, output $\bot_{\mathsf{FG}}$ is used whenever the system has not yet stabilized; and outputs true and false are used whenever a definite answer is possible.

While this modification of the many-valued semantics seems to be only a notational change, it already improves the evaluation of the above example: $\mathsf{FG}p_1 \vee \mathsf{FG}p_2$ is evaluated on every finite prefix of even length of p_1, p_2, p_1, p_2 to $\bot_{\mathsf{FG}}$ so that a verification engineer considering the results during simulation or runtime verification will see that either the system has not yet stabilized or that something is wrong in the system. Indeed, we prove that our new semantics is asymptotically correct for persistence properties in the sense that only finitely many prefixes of a satisfying infinite path of a persistence property yield the wrong result $\bot_{\mathsf{FG}}$.

Recurrence properties like the request/response property are evaluated over a *different four-valued set of truth values* $\{\mathsf{true}, \mathsf{false}, \top_{\mathsf{GF}}, \bot_{\mathsf{GF}}\}$. For the prefixes of a satisfying infinite path of a recurrence property, infinitely often the right result $\top_{\mathsf{GF}}$ is obtained so that we again obtain an asymptotically correct semantics for recurrence properties. For the simpler classes of safety, guarantee and prefix (boolean combination of safety and guarantee) properties that can be already evaluated on a three-valued semantics, we obtain the same semantics as in RV-LTL (and LTL_3). Our improvements are based on a new definition of the disjunction operator which also considers the prefix of a path, and a context-dependent interpretation of the next operator.

While we ultimately fail to give a finite path semantics for full LTL, we are able to provide a solution for all classes of the temporal logic hierarchy. In practice, this is no restriction: nearly all formulas belong (syntactically) to the most powerful class of the hierarchy and for others, it is typically not difficult to find an equivalent formula in that class [11]. This is due to the fact that this class contains an (semantically) equivalent formula for every LTL formula [16].

The outline of this paper is as follows: In Section 2, the syntax and semantics of LTL over infinite words and Manna and Pnueli's temporal logic hierarchy are reviewed. We reconsider the definition of two previously published definitions of LTL on finite words in Section 3, namely LTL_3 [4] which is essentially the same as [13] and the four-valued semantics of RV-LTL [3] which is essentially the logic-based variant of [8]. Since both logics produce misleading results on certain properties, we present a new semantics of LTL on finite paths in Section 4. We prove that our new semantics is asymptotically correct in Section 4.3 and add concluding remarks in Section 5.

2 Syntax and Semantics of LTL

Linear Temporal Logic (LTL) [12,7] is a popular formalism for the specification of temporal properties. For a given set of boolean variables (propositions) $\mathcal{V}$, we define the set of LTL formulas by the following grammar: $\varphi := \mathcal{V} \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathsf{X}\varphi \mid [\varphi \,\underline{\mathsf{U}}\, \varphi]$. Additionally, we define $\varphi \wedge \psi$, $\mathsf{F}\varphi$, $\mathsf{G}\varphi$, and $[\varphi \,\mathsf{U}\, \psi]$ as abbreviations

for $\neg(\neg\varphi \vee \neg\psi)$, $[1 \underline{\mathsf{U}} \varphi]$, $\neg\mathsf{F}\neg\varphi$, and $[\varphi \underline{\mathsf{U}} \psi] \vee \mathsf{G}\varphi$, respectively. The semantics of LTL is usually given with respect to an infinite path through a transition system. These infinite paths are nothing else than infinite sequences of boolean assignments to the variables $\mathcal{V}$:

Definition 1 (Infinite Words). *Given a set of atomic propositions $\mathcal{V}$, an* infinite word *is a function $\mathfrak{v} : \mathbb{N} \to \wp(\mathcal{V})$. For reasons of simplicity, $\mathfrak{v}(i)$ is often denoted by $\mathfrak{v}^{(i)}$ for $i \in \mathbb{N}$. Using this notation, words are often given in the form $\mathfrak{v}^{(0)}\mathfrak{v}^{(1)}\ldots$. The suffix starting at t is written as : $\mathfrak{v}^{(t\ldots)} := \mathfrak{v}^{(t)}\mathfrak{v}^{(t+1)}\ldots$. For $a \in \mathcal{V}$, we define $\mathfrak{v} = a^\omega$ as $\mathfrak{v} = a^{(0)}a^{(1)}a^{(2)}\ldots$. Given an infinite word $\mathfrak{v} = a^{(0)}a^{(1)}\ldots$, we define $\mathfrak{v}^{(s\ldots t)}$ as the finite word $\mathfrak{u} = \mathfrak{v}^{(s)}\mathfrak{v}^{(s+1)}\ldots\mathfrak{v}^{(t)}$.*

The semantics of LTL is typically defined as follows [7,16]:

Definition 2 (Semantics of LTL). *Given an infinite word $\mathfrak{v}$, the following rules define the semantics of LTL:*

- $[\mathfrak{v} \models_\omega p]$ *iff* $p \in \mathfrak{v}^{(0)}$ *for* $p \in \mathcal{V}$
- $[\mathfrak{v} \models_\omega \neg\varphi]$ *iff* $[\mathfrak{v} \not\models_\omega \varphi]$
- $[\mathfrak{v} \models_\omega \varphi \wedge \psi]$ *iff* $[\mathfrak{v} \models_\omega \varphi]$ *and* $[\mathfrak{v} \models_\omega \psi]$
- $[\mathfrak{v} \models_\omega \varphi \vee \psi]$ *iff* $[\mathfrak{v} \models_\omega \varphi]$ *or* $[\mathfrak{v} \models_\omega \psi]$
- $[\mathfrak{v} \models_\omega \mathsf{X}\varphi]$ *iff* $[\mathfrak{v}^{(1\ldots)} \models_\omega \varphi]$
- $[\mathfrak{v} \models_\omega [\varphi \underline{\mathsf{U}} \psi]]$ *iff there is a* δ *such that* $[\mathfrak{v}^{(\delta\ldots)} \models_\omega \psi]$ *and for all* t *with* $t < \delta$, *we have* $[\mathfrak{v}^{(t\ldots)} \models_\omega \varphi]$

In [5,15,16], a temporal logic hierarchy has been defined in analogy to the hierarchy of ω-automata. Following [15], we define the hierarchy of temporal formulas by the grammar rules of Figure 1:

$P_\mathsf{G} ::= \mathcal{V} \mid \neg P_\mathsf{F} \mid P_\mathsf{G} \wedge P_\mathsf{G} \mid P_\mathsf{G} \vee P_\mathsf{G}$ $\mid \mathsf{X}P_\mathsf{G} \mid [P_\mathsf{G} \,\mathsf{U}\, P_\mathsf{G}]$	$P_\mathsf{F} ::= \mathcal{V} \mid \neg P_\mathsf{G} \mid P_\mathsf{F} \wedge P_\mathsf{F} \mid P_\mathsf{F} \vee P_\mathsf{F}$ $\mid \mathsf{X}P_\mathsf{F} \mid [P_\mathsf{F} \,\underline{\mathsf{U}}\, P_\mathsf{F}]$
$P_\mathsf{Prefix} ::= P_\mathsf{G} \mid P_\mathsf{F} \mid \neg P_\mathsf{Prefix} \mid P_\mathsf{Prefix} \wedge P_\mathsf{Prefix} \mid P_\mathsf{Prefix} \vee P_\mathsf{Prefix}$	
$P_\mathsf{GF} ::= P_\mathsf{Prefix}$ $\mid \neg P_\mathsf{FG} \mid P_\mathsf{GF} \wedge P_\mathsf{GF} \mid P_\mathsf{GF} \vee P_\mathsf{GF}$ $\mid \mathsf{X}P_\mathsf{GF} \mid [P_\mathsf{GF} \,\mathsf{U}\, P_\mathsf{GF}] \mid [P_\mathsf{GF} \,\underline{\mathsf{U}}\, P_\mathsf{F}]$	$P_\mathsf{FG} ::= P_\mathsf{Prefix}$ $\mid \neg P_\mathsf{GF} \mid P_\mathsf{FG} \wedge P_\mathsf{FG} \mid P_\mathsf{FG} \vee P_\mathsf{FG}$ $\mid \mathsf{X}P_\mathsf{FG} \mid [P_\mathsf{FG} \,\underline{\mathsf{U}}\, P_\mathsf{FG}] \mid [P_\mathsf{G} \,\mathsf{U}\, P_\mathsf{FG}]$
$P_\mathsf{Streett} ::= P_\mathsf{GF} \mid P_\mathsf{FG} \mid \neg P_\mathsf{Streett} \mid P_\mathsf{Streett} \wedge P_\mathsf{Streett} \mid P_\mathsf{Streett} \vee P_\mathsf{Streett}$	

Fig. 1. Classes of the Temporal Logic Hierarchy

Definition 3 (Temporal Logic Classes). *We define the logics TL_κ for $\kappa \in \{\mathsf{G}, \mathsf{F}, \mathsf{Prefix}, \mathsf{FG}, \mathsf{GF}, \mathsf{Streett}\}$ by the grammar rules given in Figure 1, where TL_κ is the set of formulas that can be derived from the non-terminal P_κ ($\mathcal{V}$ represents any variable $v \in \mathcal{V}$).*

TL_G is the set of formulas where each occurrence of a weak/strong temporal operator is positive/negative, and similarly, each occurrence of a weak/strong temporal operator in TL_F is negative/positive. Hence, both logics are dual to each other, which means that one contains the negations of the other one. $\mathsf{TL}_\mathsf{Prefix}$

is the boolean closure of TL_G and TL_F. The logics TL_GF and TL_FG are constructed in the same way as TL_G and TL_F; however, there are two differences: (1) these logics allow occurrences of $\mathsf{TL}_{\mathsf{Prefix}}$ where otherwise variables would have been required in TL_G and TL_F, and (2) there are additional 'asymmetric' grammar rules. It can be easily proved that TL_GF and TL_FG are also dual to each other, and their intersection strictly contains $\mathsf{TL}_{\mathsf{Prefix}}$. Finally, $\mathsf{TL}_{\mathsf{Streett}}$ is the boolean closure of TL_GF and TL_FG. While there are syntactic restrictions on $\mathsf{TL}_{\mathsf{Streett}}$, i. e. not every LTL formula is a $\mathsf{TL}_{\mathsf{Streett}}$ formula, $\mathsf{TL}_{\mathsf{Streett}}$ contains for each LTL formula an equivalent formula, and nearly all formulas used in practice belong to $\mathsf{TL}_{\mathsf{Streett}}$ [11]. Moreover, for those formulas not in $\mathsf{TL}_{\mathsf{Streett}}$, it is typically not difficult to find an equivalent one in $\mathsf{TL}_{\mathsf{Streett}}$.

3 Previous Definitions of LTL on Finite Paths

In the following, we consider the recently proposed semantics for LTL on finite paths as given in [3]. We also show that this definition has certain deficiencies.

3.1 LTL_3

In [2], LTL_3 was introduced as an extension of LTL to finite paths which follows the idea that a finite path is a prefix of a so-far unknown infinite path. LTL_3 uses three-valued truth values $\mathbb{B}_3 = \{1, 0, ?\}$. While the syntax of LTL_3 coincides with that of LTL, its semantics is defined on finite words:

Definition 4 (Semantics of LTL_3). *Let $\mathfrak{u} = \mathfrak{u}^{(0)}\mathfrak{u}^{(1)}\ldots\mathfrak{u}^{(n)} \in \Sigma^*$ denote a finite path of length $n+1$. The truth value of a LTL_3 formula φ w.r.t. $\mathfrak{u}$, denoted by $[\mathfrak{u} \models_3 \varphi]$ is defined as follows:*

$$[\mathfrak{u} \models_3 \varphi] = \begin{cases} 1 & \text{if } \forall \mathfrak{w} \in \Sigma^\omega : \mathfrak{u}\mathfrak{w} \models_\omega \varphi \\ 0 & \text{if } \forall \mathfrak{w} \in \Sigma^\omega : \mathfrak{u}\mathfrak{w} \not\models_\omega \varphi \\ ? & \text{else} \end{cases}$$

The intuition behind LTL_3 is clear: whenever all infinite words obtained by concatenating the finite word with an infinite suffix agree on the truth value of φ, this truth value is used also for the prefix. Otherwise, the value ? is used. As argued in [3], LTL_3 can never give a result other than ? for request-response properties like $\mathsf{G}(r \rightarrow \mathsf{F}a)$ since every prefix of an infinite accepted word can be both a good or a bad prefix. Hence the authors propose to combine LTL_3 with another logic called FLTL.

3.2 FLTL

In [9,3] it is argued that there is a need to distinguish between a strong ($\underline{\mathsf{X}}$) and a weak ($\overline{\mathsf{X}}$) next operator when interpreting LTL over finite paths. While a weak next operator should be satisfied whenever no next position exists, a strong next operator should be evaluated to false in that case. This leads to the following definition of FLTL:

Definition 5 (FLTL). *Let* $\mathfrak{u} = \mathfrak{u}^{(0)}\mathfrak{u}^{(1)}\dots\mathfrak{u}^{(n)} \in \Sigma^*$ *denote a finite path of length* $n+1$ *with* $\mathfrak{u} \neq \varepsilon$*. The truth value of a FLTL formula* φ *wrt.* $\mathfrak{u}$*, denoted as* $[\mathfrak{u} \models_{FLTL} \varphi]$*, is an element of* $\mathbb{B}_2 = \{\bot, \top\}$ *and is inductively defined as follows: While atomic propositions and boolean operators are defined as for LTL, the temporal operators are defined as follows:*

$$[\mathfrak{u} \models_{FLTL} \overline{\mathsf{X}}\varphi] = \begin{cases} [\mathfrak{u}^1 \models_{FLTL} \varphi] & \text{if } \mathfrak{u}^1 \neq \varepsilon \\ \top & \text{else} \end{cases}$$

$$[\mathfrak{u} \models_{FLTL} \underline{\mathsf{X}}\varphi] = \begin{cases} [\mathfrak{u}^1 \models_{FLTL} \varphi] & \text{if } \mathfrak{u}^1 \neq \varepsilon \\ \bot & \text{else} \end{cases}$$

$$[\mathfrak{u} \models_{FLTL} [\varphi\ \underline{\mathsf{U}}\ \psi]] = \begin{cases} \top & \exists k \in \{1,\dots n\} : [\mathfrak{u}^k \models_{FLTL} \psi] = \top \wedge \\ & \forall 1 \leq l \leq k : [\mathfrak{u}^l \models_{FLTL} \phi] = \top \\ \bot & \text{else} \end{cases}$$

$$[\mathfrak{u} \models_{FLTL} [\varphi\ \mathsf{U}\ \psi]] = \begin{cases} \top & \forall 1 \leq l \leq n : [\mathfrak{u}^l \models_{FLTL} \varphi] = \top \vee \\ & \exists k \in \{1,\dots n\} : [\mathfrak{u}^k \models_{FLTL} \psi] = \top \wedge \\ & \forall 1 \leq l \leq k : [\mathfrak{u}^l \models_{FLTL} \phi] = \top \\ \bot & \text{else} \end{cases}$$

In [3], the two definitions of LTL_3 and FLTL are combined in a logic called RV-LTL. This logic is evaluated over a four-valued de Morgan lattice $0 \sqsubset \bot_P \sqsubset \top_P \sqsubset 1$ to express false, presumably false, presumably true and true. To obtain a de Morgan lattice and thus a truth domain, the operators $\sqcap$ and $\sqcup$ are defined as expected and $1/0$ and $\top_P/\bot_P$, respectively, are defined to be complementary to each other. Note that the thereby obtained truth domain $\mathbb{B}_4$ is not a boolean lattice.

RV-LTL is now defined such that the truth value of LTL_3 is used whenever it is conclusive, i.e. gives 1 or 0. If LTL_3 provides the inconclusive result (?), the definition of FLTL is used instead:

Definition 6 (RV-LTL). *Let* $\mathfrak{u} = \mathfrak{u}^{(0)}\mathfrak{u}^{(1)}\dots\mathfrak{u}^{(n)} \in \Sigma^*$ *denote a finite path of length* $n+1$ *with* $\mathfrak{u} \neq \varepsilon$*. The truth value of an RV-LTL formula* φ *wrt.* $\mathfrak{u}$*, denoted as* $[\mathfrak{u} \models_{FLTL} \varphi]$*, is an element of* $\mathbb{B}_4$ *and is defined as follows:*

$$[\mathfrak{u} \models_3 \varphi] = \begin{cases} 1 & \text{if } [\mathfrak{u}\mathfrak{w} \models_3 \varphi] = 1 \\ 0 & \text{if } [\mathfrak{u}\mathfrak{w} \models_3 \varphi] = 0 \\ \top_P & \text{if } [\mathfrak{u}\mathfrak{w} \models_3 \varphi] = ? \wedge [\mathfrak{u}\mathfrak{w} \models_{FLTL} \varphi] = \top \\ \bot_P & \text{if } [\mathfrak{u}\mathfrak{w} \models_3 \varphi] = ? \wedge [\mathfrak{u}\mathfrak{w} \models_{FLTL} \varphi] = \bot \end{cases}$$

3.3 Problems with RV-LTL

In the following, we consider some examples to show unsatisfactory results of the RV-LTL semantics.

Request/Acknowledge Properties: In [3], it has been shown that $\mathsf{F}\varphi \equiv_{RV} \varphi \vee \underline{\mathsf{X}}\mathsf{F}\varphi$ holds, satisfying the intuitive meaning that $\mathsf{F}\varphi$ holds, iff either φ holds immediately or there must be a future state satisfying φ. If no such future state exists, the formula evaluates to $\bot_P$, unless the formula evaluates to one of $\{1, 0\}$, in which case the future is not important. Similarly, we have $\mathsf{G}\varphi \equiv_{RV} \varphi \wedge \overline{\mathsf{X}}\mathsf{G}\varphi$ which shows that φ must be satisfied in the current state and in all observable future states. Hence, if there is no future state, the formula evaluates to $\top_P$, unless the formula evaluates to one of $\{1, 0\}$. Hence, the request/acknowledge property is evaluated as follows:

$$\begin{aligned}\mathsf{G}\,(r \rightarrow \mathsf{F}a) &\equiv_{RV} (r \rightarrow \mathsf{F}a) \wedge \overline{\mathsf{X}}\mathsf{G}\,(r \rightarrow \mathsf{F}a)\\ &\equiv_{RV} (\neg r \vee a \vee \underline{\mathsf{X}}\mathsf{F}a) \wedge \overline{\mathsf{X}}\mathsf{G}\,(r \rightarrow \mathsf{F}a)\end{aligned}$$

This formula evaluates to $\bot_P$ under RV-LTL if the path contains an r but ends before a occurs and evaluates to $\top_P$ in all other cases. Thus, its semantics seems to be reasonable. However, consider the following generalized request/acknowledge property:

$$\begin{aligned}&\mathsf{G}\,(r_1 \rightarrow \mathsf{F}a_1) \wedge \mathsf{G}\,(r_2 \rightarrow \mathsf{F}a_2) \equiv_{RV}\\ &\quad(\neg r_1 \vee a_1 \vee \underline{\mathsf{X}}\mathsf{F}a_1) \wedge \overline{\mathsf{X}}\mathsf{G}\,(r_1 \rightarrow \mathsf{F}a_1) \wedge (\neg r_2 \vee a_2 \vee \underline{\mathsf{X}}\mathsf{F}a_2) \wedge \overline{\mathsf{X}}\mathsf{G}\,(r_2 \rightarrow \mathsf{F}a_2)\end{aligned}$$

According to the previous discussion, a finite word that satisfies $r_1 \wedge a_2$ on odd positions and $r_2 \wedge a_1$ on even positions (the others being false) will be evaluated to $\bot_P$ in all states. This is unfortunate because the infinite word that is obtained by an infinite concatenation of those odd/even positions clearly satisfies the specification under the infinite semantics

Stabilization Properties: While having a semantics that evaluates to a 'bad' value even if we read a 'good' word may be acceptable, the following example demonstrates that RV-LTL even has the undesirable property that for a non-accepted word of a LTL property, each finite prefix may be evaluated to $\top_P$. To this end, consider the following RV-LTL equivalence:

$$\mathsf{FG}a \vee \mathsf{FG}\neg a \equiv_{RV} \mathsf{F}\left(a \wedge \overline{\mathsf{X}}\mathsf{G}a\right) \vee \mathsf{F}\left(\neg a \wedge \overline{\mathsf{X}}\mathsf{G}\neg a\right)$$

Since $\overline{\mathsf{X}}\mathsf{G}a$ is evaluated weakly and $\mathsf{FG}a$ may start the evaluation of $\overline{\mathsf{X}}\mathsf{G}a$ at an arbitrary position (for example, the last position of the finite word read so far), every finite word that ends with a evaluates to true. However, with the same argument, every finite word that ends with $\neg a$ is evaluated to true, too. Hence, the word with a on even positions and $\neg a$ on odd positions will be evaluated to $\top_P$ on each position. Nevertheless, the thereby constructed infinite word is not accepted by the infinite semantics of LTL. The problem is that the evaluation of $\varphi \vee \psi$ in RV-LTL does not consider which property has been responsible for the satisfaction in previous steps and hence such an infinite shift between good and bad prefixes for φ and ψ is possible. We will later see that this problem can be fixed by an improved semantics for the disjunction.

The Problem with Two Next-Operators: While having weak/strong next operators may seem plausible at first sight, we argue that it leads to problems when one is interested in an asymptotically correct semantics for LTL. One might expect that both $\underline{\mathsf{X}}$ and $\overline{\mathsf{X}}$ will behave asymptotically like the original X operator. However, this may not hold: Consider e.g. the property $\mathsf{G}\underline{\mathsf{X}}a$. This property is evaluated to $\bot$ on every input at every step. However, since the formula $\mathsf{GX}a$ holds on the word a^ω with the infinite semantics, one might expect that at least at some point, $\mathsf{G}\underline{\mathsf{X}}a$ yields $\top$, which is however not the case. Moreover, the intuitive meaning of G should be that it is evaluated to $\top$ as long as we have not detected that the property is violated. This intuitive interpretation does no longer hold if we allow a $\underline{\mathsf{X}}$ inside a G. A similar problem occurs with $\mathsf{F}\overline{\mathsf{X}}a$. One might expect that in the limit, this formula behaves like $\mathsf{FX}a$. However, since $\overline{\mathsf{X}}a$ is evaluated weak, it is not hard to see that this formula evaluates to $\top$, no matter which input is read.

To circumvent those problems, we refrain therefore from two different next operators and evaluate the next operator depending on the context of a formula. The intuitive idea behind our construction is that if $\mathsf{X}\varphi$ is in the scope of a weak temporal operator, it is evaluated weakly, otherwise it is evaluated strongly. Hence, for $\mathsf{TL_G}$ formulas, we evaluate the formula always weakly in accordance to the intuitive meaning that a safety formula should be evaluated to $\top$ as long as nothing bad happened. Analogously, we evaluate a X operator in the scope of a strong until operator strongly, as e.g. in $\mathsf{F}(a \wedge \mathsf{X}b)$. This supports the intuitive meaning that a guarantee property should be evaluated to $\bot$ as long as it has not definitely been satisfied.

4 Asymptotic Finite Linear Temporal Logic (RV^∞–LTL)

In this section, we define for each $\kappa \in \{\mathsf{G}, \mathsf{F}, \mathsf{Prefix}, \mathsf{FG}, \mathsf{GF}, \mathsf{Streett}\}$ specialized semantics that are intended to replace the FLTL semantics in the definition of RV-LTL. We call the resulting logics RV^∞–TL_κ. For better readability of the following definitions, we assume that the case conditions are evaluated in a top-down manner, i.e, if the first satisfied case is used (ignoring all remaining ones, including also possibly satisfied cases).

4.1 The Temporal Logic Classes RV^∞–$\mathsf{TL_G}$ and RV^∞–$\mathsf{TL_F}$

We start by defining the base class RV^∞–$\mathsf{TL_G}$:

Definition 7 (Semantics of Linear Temporal Logic RV^∞–$\mathsf{TL_G}$). *Let* $\mathfrak{u} = \mathfrak{u}^{(0)}\mathfrak{u}^{(1)}\dots\mathfrak{u}^{(n)} \in \Sigma^*$ *denote a finite path of length* $n+1$*. The truth value of an* $\mathsf{TL_G}$ *formula* φ *wrt.* $\mathfrak{u}$*, denoted with* $[\mathfrak{u} \models_\mathsf{G} \varphi]$*, is an element of* $\mathbb{B}_3$ *and is inductively defined as follows:*

- $[\varepsilon \models_\mathsf{G} \varphi] = \top_\mathsf{G}$
- $[\mathfrak{u} \models_\mathsf{G} a] = \begin{cases} 1 & \text{if } a \in \mathfrak{u}^{(0)} \\ 0 & \text{else} \end{cases}$, *for every* $a \in \mathcal{V}$

$$- \ [\mathfrak{u} \models_{\mathsf{G}} \varphi \wedge \psi] = \begin{cases} 1 & \text{if } \forall \mathfrak{w} \in \Sigma^{\omega} : \mathfrak{u}\mathfrak{w} \models_{\omega} \varphi \wedge \psi \\ \top_{\mathsf{G}}, & \text{if } [\mathfrak{u} \models_{\mathsf{G}} \varphi] = \top_{\mathsf{G}} \text{ and } [\mathfrak{u} \models_{\mathsf{G}} \psi] = \top_{\mathsf{G}} \\ 0, & \text{otherwise} \end{cases}$$

$$- \ [\mathfrak{u} \models_{\mathsf{G}} \varphi \vee \psi] = \begin{cases} 1 & \text{if } \forall \mathfrak{w} \in \Sigma^{\omega} : \mathfrak{u}\mathfrak{w} \models_{\omega} \varphi \vee \psi \\ \top_{\mathsf{G}}, & \text{if } [\mathfrak{u} \models_{\mathsf{G}} \varphi] = \top_{\mathsf{G}} \text{ or } [\mathfrak{u} \models_{\mathsf{G}} \psi] = \top_{\mathsf{G}} \\ 0, & \text{otherwise} \end{cases}$$

$$- \ [\mathfrak{u} \models_{\mathsf{G}} \mathsf{X}\varphi] = [\mathfrak{u}^{(1\ldots n)} \models_{\mathsf{G}} \varphi]$$

$$- \ [\mathfrak{u} \models_{\mathsf{G}} [\varphi \ \mathsf{U} \ \psi]] = [\mathfrak{u} \models_{\mathsf{G}} (\psi \vee (\varphi \wedge \mathsf{X}[\varphi \ \mathsf{U} \ \psi]))]$$

Taking into account that the X operator is evaluated weakly in a TL_{G} formula, the definition of $[\varphi \ \mathsf{U} \ \psi]$ is exactly the fixpoint evaluation of $[\varphi \ \mathsf{U} \ \psi]$. Hence, it is not hard to see that FLTL and RV^{∞}–LTL are evaluated in the same manner:

Proposition 1. *Let φ be a TL_{G} formula and $\mathfrak{u} \neq \varepsilon$ be a finite word. Let φ' be obtained from φ by replacing each X operator by a $\overline{\mathsf{X}}$ operator. Then, the following holds: $[\mathfrak{u} \models_{\mathsf{G}} \varphi] = \top_{\mathsf{G}}$ iff $[\mathfrak{u} \models_{FLTL} \varphi'] = \top$.*

Since the negations of safety properties are guarantee properties, we define:

Definition 8 (Semantics of Linear Temporal Logic RV^{∞}–TL_{F}). *Given a finite prefix $\mathfrak{u} = \mathfrak{u}^{(0)}\mathfrak{u}^{(1)}\ldots\mathfrak{u}^{(n)}$ of an infinite word $\mathfrak{u}_{\infty}$, the the semantics of RV^{∞}–TL_{F} is defined by* $[\mathfrak{u} \models_{\mathsf{F}} \varphi] = \begin{cases} 1, & \text{if } [\mathfrak{u} \models_{\mathsf{G}} \neg\varphi] = 0 \\ \bot_{\mathsf{F}}, & \text{if } [\mathfrak{u} \models_{\mathsf{G}} \neg\varphi] = \top_{\mathsf{G}} \\ 0, & \text{otherwise} \end{cases}$

Hence, the following is also obvious:

Proposition 2. *Let φ be a TL_{F} formula and $\mathfrak{u} \neq \varepsilon$ be a finite word. Let φ' be obtained from φ by replacing each X operator by a $\underline{\mathsf{X}}$ operator. Then, the following holds: $[\mathfrak{u} \models_{\mathsf{F}} \varphi] = 1$ iff $[\mathfrak{u} \models_{FLTL} \varphi'] = \top$.*

4.2 The Temporal Logic RV^{∞}–$\mathsf{TL}_{\mathsf{FG}}$

In the following, we will use $\mathfrak{u} \models_{\mathsf{FG}} \varphi$ as shorthand for $[\mathfrak{u} \models_{\mathsf{FG}} \varphi] \in \{1, \top_{\mathsf{FG}}\}$ and $\mathfrak{u} \not\models_{\mathsf{FG}} \varphi$ as a shorthand for $[\mathfrak{u} \models_{\mathsf{FG}} \varphi] \in \{0, \bot_{\mathsf{FG}}\}$

Definition 9 (Semantics of Linear Temporal Logic RV^{∞}–$\mathsf{TL}_{\mathsf{FG}}$). *Let $\mathfrak{u} = \mathfrak{u}^{(0)}\mathfrak{u}^{(1)}\ldots\mathfrak{u}^{(n)} \in \Sigma^{*}$ denote a finite path of length $n+1$. The truth value of a $\mathsf{TL}_{\mathsf{FG}}$ formula φ wrt. $\mathfrak{u}$, denoted with $[\mathfrak{u} \models_{\mathsf{FG}} \varphi]$, is an element of $\mathbb{B}_4$ and is recursively defined as follows:*

$$[\mathfrak{u} \models_{\mathsf{FG}} \varphi] = \begin{cases} 1 & \text{if } \forall \mathfrak{w} \in \Sigma^{\omega} : \mathfrak{u}\mathfrak{w} \models_{\omega} \varphi \\ 0 & \text{if } \forall \mathfrak{w} \in \Sigma^{\omega} : \mathfrak{u}\mathfrak{w} \not\models_{\omega} \varphi \\ \top_{\mathsf{FG}}{}^{1} & \text{if } \varphi \in \mathsf{TL}_{\mathsf{G}}{}^{2} \text{ and } \mathfrak{u} \models_{\mathsf{G}} \varphi \\ [\mathfrak{u} \models_{\mathsf{FG}'} \varphi] & \text{otherwise} \end{cases}$$

where we define $[\mathfrak{u} \models_{\mathsf{FG}'} \varphi]$ *by:*[3]

- $[\varepsilon \models_{\mathsf{FG}'} \varphi] = \bot_{\mathsf{FG}}$
- $[\mathfrak{u} \models_{\mathsf{FG}'} \varphi \wedge \psi] = \begin{cases} \top_{\mathsf{FG}}, & \text{if } \mathfrak{u} \models_{\mathsf{FG}} \varphi \text{ and } \mathfrak{u} \models_{\mathsf{FG}} \psi \\ \bot_{\mathsf{FG}}, & \text{otherwise} \end{cases}$
- $[\mathfrak{u} \models_{\mathsf{FG}'} \varphi \vee \psi] = \begin{cases} \top_{\mathsf{FG}}, & \text{if } \exists t \left(\mathfrak{u}^{(0\ldots t)} \not\models_{\mathsf{FG}} \varphi \vee \psi\right) \text{ and} \\ & \quad \left(\left(\forall_{k=t+1}^{n} \mathfrak{u}^{(0\ldots k)} \models_{\mathsf{FG}} \varphi\right) \text{ or } \left(\forall_{k=t+1}^{n} \mathfrak{u}^{(0\ldots k)} \models_{\mathsf{FG}} \psi\right)\right) \\ \bot_{\mathsf{FG}}, & \text{otherwise} \end{cases}$
- $[\mathfrak{u} \models_{\mathsf{FG}'} [\varphi \,\underline{\mathsf{U}}\, \psi]] = \begin{cases} \top_{\mathsf{FG}}, & \text{if } \exists t \left(\mathfrak{u}^{(0\ldots t)} \not\models_{\mathsf{FG}} [\varphi \,\underline{\mathsf{U}}\, \psi]\right) \text{ and} \\ & \quad \exists j \leq t . \mathfrak{u}^{(j\ldots n)} \models_{\mathsf{FG}} \psi \wedge \forall k < j . \mathfrak{u}^{(k\ldots n)} \models_{\mathsf{FG}} \varphi \\ \bot_{\mathsf{FG}}, & \text{otherwise} \end{cases}$
- $[\mathfrak{u} \models_{\mathsf{FG}'} [\varphi \,\mathsf{U}\, \psi]] = \begin{cases} \top_{\mathsf{FG}}, & \text{if } \left(\forall k \leq n . \mathfrak{u}^{(k\ldots n)} \models_{\mathsf{FG}} \varphi\right) \qquad (*) \\ & \text{or} \\ & \quad \exists t \left(\mathfrak{u}^{(0\ldots t)} \not\models_{\mathsf{FG}} [\varphi \,\mathsf{U}\, \psi]\right) \text{ and} \\ & \quad \exists j \leq t . \mathfrak{u}^{(j\ldots n)} \models_{\mathsf{FG}} \psi \wedge \forall k < j . \mathfrak{u}^{(k\ldots n)} \models_{\mathsf{FG}} \varphi \\ \bot_{\mathsf{FG}}, & \text{otherwise} \end{cases}$

Before presenting the proof of asymptotic correctness, we would like to emphasize the strength of our definition which is the consideration of *breakpoints*[4] in the definition of the $\vee$ and the two until operators. This *breakpoint* is a point of time where the currently evaluated formula has evaluated to $\bot_{\mathsf{FG}}$ for the last time. In case of a disjunction, the evaluation of a finite word $\mathfrak{u}$ of length $n+1$ evaluates to $\top_{\mathsf{FG}}$ if and only if after a breakpoint (which can be also at position -1 where we evaluate the empty word) one of the two formulas invariantly evaluates to $\top_{\mathsf{FG}}$. This ensures that we can not jump freely from evaluating once φ and once ψ, but must instead stick to one particular subformula.

A similar trick is used in the definition of the strong until operator. Here, we demand that the starting point j from where on ψ holds does not cross the last breakpoint. This ensures that we can not freely jump to an arbitrary position and restart the evaluation of ψ in each step in an RV-context. Consider for example the formula $[a \,\underline{\mathsf{U}}\, (\mathsf{FG}b \vee \mathsf{FG}c)]$ and the following path for runtime verification: a holds in every step while in an even step b holds and in an odd step c holds. If we remove the t-breakpoint from the definition, we would have the unpleasant behavior that this formula evaluates to $\top_{\mathsf{FG}}$ in every step which is however not true. Having the breakpoint ensures that this can not happen.

[1] The value $\top_{\mathsf{G}}$ is also reasonable here.

[2] Notice that the case $\varphi \in \mathsf{TL}_{\mathsf{F}}$ is already contained in the first case, because $\mathfrak{u} \models_{\mathsf{F}} \varphi$ is defined as $[\mathfrak{u} \models_{\mathsf{F}} \varphi] = 1$, which means that once we found that a TL_{F} is satisfied, it is satisfied for all suffixes.

[3] Notice that the case of propositional variables is handled by the $\mathsf{RV}^{\infty}\text{–}\mathsf{TL}_{\mathsf{G}}$ evaluation.

[4] Readers familiar with Miyano and Hayashi's breakpoint construction [10] for the non-determinization of alternating Büchi automata or the closely related determinization procedure for co-Büchi automata [16] might notice the similarity: in their construction a set is filled with a new set of states whenever it is discovered that the co-Büchi condition is falsified.

4.3 Asymptotic Correctness

We will now turn to the proof of asymptotic correctness. To this end, we show that an (infinite) word $\mathfrak{u}$ is accepted by a $\mathsf{TL}_{\mathsf{FG}}$ formula if and only if there is a definitive last breakpoint, called the *rv-threshold*, so that after this point, the RV^{∞}–$\mathsf{TL}_{\mathsf{FG}}$-definition invariantly evaluates to $\top_{\mathsf{FG}}$.

Lemma 1. *Let $\mathfrak{u}$ be an infinite word, and Φ be an $\mathsf{TL}_{\mathsf{FG}}$ formula. Then, the following holds: If $\mathfrak{u} \models_{\omega} \Phi$, there exists a rv-threshold $t \in \mathbb{N}$ such that for every $k > t$ we have $\mathfrak{u}^{(0\dots k)} \models_{\mathsf{FG}} \Phi$.*

Proof. We neglect the case that at some point the whole formula evaluates to 1 since in that case the claim trivially holds. We prove this lemma by induction on the formula length. Clearly, if the length is 1, we have a constant value and our rv-threshold is 1 so that the proof is obtained. Assume now that the claim holds for every formula of length l. We show that it also holds for formula of length $l + 1$. To this end, we split the proof into different cases, depending on the top-level operator Φ:

$\varphi \vee \psi$: According to the definition of LTL, we must have that $\mathfrak{u} \models_{\omega} \varphi$ or $\mathfrak{u} \models_{\omega} \psi$ holds. W.l.o.g. assume that $\mathfrak{u} \models_{\omega} \varphi$ holds. Thus, we must have a rv-threshold t for φ according to our induction hypothesis. Now assume that we have infinitely often that $\mathfrak{u}^{(0\dots k)} \not\models_{\mathsf{FG}} \Phi$ holds. Thus, we must have a position $t' > t$ such that $\mathfrak{u}^{(0\dots k)} \not\models_{\mathsf{FG}} \varphi \vee \psi$ holds. However, according to the rv-threshold, we have that $\mathfrak{u}^{(0\dots k)} \models_{\mathsf{FG}} \varphi$ holds for every $k > t$. It is not hard to see that this ensures that $\varphi \vee \psi$ is evaluated to $\top_{\mathsf{FG}}$ from that point on.

$[\varphi \,\underline{\mathsf{U}}\, \psi]$: According to the definition of LTL, there must exist a position j such that $\mathfrak{u}^{(j\dots)} \models_{\omega} \psi$ and for all $k < j$ we have $\mathfrak{u}^{(k\dots)} \models_{\omega} \varphi$. According to the induction hypothesis, there must exists a rv-threshold t_{ψ} and for each $k < j$ a rv-threshold t_k such that $\mathfrak{u}^{(0\dots t')} \models_{\mathsf{FG}} \psi$ for every $t' > t_{\psi}$ and $\mathfrak{u}^{(k\dots t')} \models_{\mathsf{FG}} \varphi$ for every $t' > t_k$. Thus, the maximum of $t_{\psi}, t_0 \dots t_{j-1}$ is our desired rv-threshold.

$[\varphi \,\mathsf{U}\, \psi]$: We can distinguish two cases: if $\mathfrak{u}$ also satisfies the strong until operator, we can use the same proof as above. For the second case, notice that φ is a TL_{G} formula (see Figure 1). Since φ is satisfied by $\mathfrak{u}$, the evaluation function for RV^{∞}–TL_{G} will always be evaluated to $\top_{\mathsf{G}}$. Thus, the claim holds.

$\mathsf{X}\psi$: : According to the definition of LTL, we have $\mathfrak{u}^{(1\dots)} \models_{\omega} \psi$ and we can apply the induction hypothesis on $\mathfrak{u}^{(1\dots)}$ to proof the claim.

$\varphi \wedge \psi$: According to the definition of LTL, $\mathfrak{u} \models_{\omega} \varphi$ and $\mathfrak{u} \models_{\omega} \psi$ holds. Thus, according to the induction hypothesis, there must exist t_{φ} and t_{ψ} as rv-thresholds. The maximum of them is the rv-threshold for $\varphi \wedge \psi$.

■

The opposite direction is shown in a similar manner:

Lemma 2. *Let $\mathfrak{u}$ be an infinite word, and Φ be an $\mathsf{TL}_{\mathsf{FG}}$ formula. Then, the following holds: If there exists a rv-threshold $t \in \mathbb{N}$ such that for every $k > t$ we have $\mathfrak{u}^{(0\dots k)} \models_{\mathsf{FG}} \Phi$. Then, $\mathfrak{u} \models_{\omega} \Phi$.*

Proof. Again, we neglect the case that at some point $\mathfrak{u}^{(0\ldots k)}$ evaluates to 1 in a rv-context. We prove this lemma by induction on the formula length. Clearly, if the length is 1, we have a constant value and our rv-threshold is 1 and the proof is obtained. Assume now that the claim holds for every formula of length l. We show that it also holds for formula of length $l+1$. To this end, we split the proof into different cases, depending on the top-level operator:

$\varphi \vee \psi$**:** According to our assumption, we have a minimal rv-threshold t such that for every $t' \geq t$ $\mathfrak{u}^{(0\ldots t')} \models_{\mathsf{FG}} \varphi \vee \psi$ holds. Since t is minimal, we have $\mathfrak{u}^{(0\ldots t-1)} \not\models_{\mathsf{FG}} \varphi \vee \psi$. According to the definition of $[\mathfrak{u} \models_{\mathsf{FG}'} \varphi \vee \psi]$, this means that either $\left(\forall_{k=t}^{t'} \mathfrak{u}^{(0\ldots k)} \models_{\mathsf{FG}} \varphi\right)$ or $\left(\forall_{k=t}^{t'} \mathfrak{u}^{(0\ldots k)} \models_{\mathsf{FG}} \psi\right)$ holds. In other words, we can not freely switch between evaluating either φ or ψ, but one of the two formulas must be evaluated to $\top_{\mathsf{FG}}$ in all places after t. This means that we can apply the induction hypothesis and can conclude that either $\mathfrak{u} \models_\omega \varphi$ or $\mathfrak{u} \models_\omega \psi$ holds. Hence, $\mathfrak{u} \models_\omega \varphi \vee \psi$ holds trivially.

$[\varphi \underline{\mathsf{U}} \psi]$**:** According to our assumption, a rv-threshold t exists such that for every $t' \geq t$ we have $\mathfrak{u}^{(0\ldots t')} \models_{\mathsf{FG}} [\varphi \underline{\mathsf{U}} \psi]$. This means that for every t' there must exist a $j_{t'} \leq t$ such that $\mathfrak{u}^{(j_{t'}\ldots t')} \models_{\mathsf{FG}} \psi$ and $\forall k < j_{t'}.\mathfrak{u}^{(k\ldots t')} \models_{\mathsf{FG}} \varphi$ holds. Now, notice that although we might have different $j_{t'}$ for each t', there can be only finitely many of them (namely those less or equal t). Hence, we must have a minimal j such that for every $t' > t$ the following holds: $\mathfrak{u}^{(j\ldots t')} \models_{\mathsf{FG}} \psi$ and $\forall k < j.\mathfrak{u}^{(k\ldots t')} \models_{\mathsf{FG}} \varphi$. Hence, according to our induction hypothesis, we must have $\mathfrak{u}^{(j\ldots)} \models_\omega \psi$ and $\forall k < j.\ \mathfrak{u}^{(k\ldots)} \models_\omega \varphi$.

$[\varphi \mathsf{U} \psi]$**:** The first case is that for every $n \in \mathbb{N}$ and every $n' > n.\mathfrak{u}^{(n\ldots n')} \models_{\mathsf{FG}} \varphi$. This means that for every $n \in \mathbb{N}$ the rv-threshold for $\mathfrak{u}^{(n\ldots)}$ is one. But this implies that we can use our induction hypothesis to show that for every $n \in \mathbb{N}$, we have $\mathfrak{u}^{(n\ldots)} \models_\omega \varphi$. Thus $\mathfrak{u} \models_\omega [\varphi \mathsf{U} \psi]$ holds. Assume now that this property does not hold, i. e. for some $n \in \mathbb{N}$ and some $n < n' \in \mathbb{N}$, we have that $\mathfrak{u}^{(n\ldots n')} \not\models_{\mathsf{FG}} \varphi$. According to the grammar of $\mathsf{TL}_{\mathsf{FG}}$, φ is a TL_{G} formula, thus $\mathfrak{u}^{(n\ldots n')} \not\models_{\mathsf{G}} \varphi$ holds also. However, the safety formula of TL_{G} are evaluated in a way such that if they are evaluated to $\mathsf{0}$ for a finite prefix $\mathfrak{w}$, they are evaluated to $\mathsf{0}$ for every suffix of $\mathfrak{w}$. Hence, after position n', the first case (*) in the $\mathsf{RV}^\infty\text{–}\mathsf{TL}_{\mathsf{FG}}$ definition of $[\varphi \mathsf{U} \psi]$ is never again satisfied. This means that the second condition must be satisfied from that point on which is exactly the same as the condition used for defining $[\varphi \underline{\mathsf{U}} \psi]$. Hence we can use the same proof as for $[\varphi \underline{\mathsf{U}} \psi]$.

$\mathsf{X}\varphi$**,** $\varphi \wedge \psi$**:** are trivial and omitted here. ∎

Remark 1. The proof for the weak until operator $[\varphi \mathsf{U} \psi]$ shows why we restricted our attention to $\mathsf{TL}_{\mathsf{FG}}$ formula: we can guarantee that φ is evaluated to $\bot_{\mathsf{FG}}$ whenever a prefix is evaluated to $\bot_{\mathsf{FG}}$ only due the special syntactic requirement that φ is a safety formula, something that is missing in arbitrary LTL formulas.

Remark 2. An alternative definition for the $[\varphi \underline{\mathsf{U}} \psi]$ operator would be based on the fixpoint iteration scheme known from translating LTL to Büchi automata:

$[\mathfrak{u} \models_{\mathsf{FG}} [\varphi \underline{\mathsf{U}}\, \psi]] := [\mathfrak{u} \models_{\mathsf{FG}} (\psi \vee (\varphi \wedge \mathsf{X}[\varphi \underline{\mathsf{U}}\, \psi]))]$. Here, the $\vee$-operator is evaluated according to our breakpoint-definition. The two definitions are indeed equivalent as one can check by an induction on n. Nevertheless we preferred the one given above since it simplifies the correctness proof.

The following theorem is therefore our main result:

Theorem 1. *Given a finite prefix* $\mathfrak{u} = \mathfrak{u}^{(0)}\mathfrak{u}^{(1)}\dots\mathfrak{u}^{(n)}$ *of an infinite word* $\mathfrak{u}_\infty$*, we have* $[\mathfrak{u}_\infty \models_\omega \varphi]$ *iff* $\nexists^\infty k.\mathfrak{u}^{(0\dots k)} \not\models_{\mathsf{FG}} \varphi$ *for every* RV^∞–$\mathsf{TL}_{\mathsf{FG}}$ *formula* φ*.*

Hence, $[\mathfrak{u}_\infty \models_\omega \varphi]$ iff $\lim\limits_{n\to\infty} [\mathfrak{u}^{(0\dots n)} \models_{\mathsf{FG}} \varphi] = \top_{\mathsf{FG}}$.

4.4 The Temporal Logic RV^∞–$\mathsf{TL}_{\mathsf{GF}}$

Since $\mathsf{TL}_{\mathsf{GF}}$ is the dual class of $\mathsf{TL}_{\mathsf{FG}}$, the following definition together with the corresponding theorem is rather straightforward:

Definition 10 (Semantics of RV^∞–$\mathsf{TL}_{\mathsf{GF}}$).
Given a finite prefix $\mathfrak{u} = \mathfrak{u}^{(0)}\mathfrak{u}^{(1)}\dots\mathfrak{u}^{(n)}$ *of an infinite word* $\mathfrak{u}_\infty$*, the semantics of* RV^∞–$\mathsf{TL}_{\mathsf{GF}}$ *is defined by*

$$[\mathfrak{u} \models_{\mathsf{GF}} \varphi] = \begin{cases} 1, & \text{if } [\mathfrak{u} \models_{\mathsf{FG}} \neg\varphi] = 0 \\ \top_{\mathsf{GF}} & \text{if } [\mathfrak{u} \models_{\mathsf{FG}} \neg\varphi] = \bot_{\mathsf{FG}} \\ \bot_{\mathsf{GF}}, & \text{if } [\mathfrak{u} \models_{\mathsf{FG}} \neg\varphi] = \top_{\mathsf{FG}} \\ 0, & \text{if } [\mathfrak{u} \models_{\mathsf{FG}} \neg\varphi] = 1 \end{cases}$$

Theorem 2. *Given a finite prefix* $\mathfrak{u} = \mathfrak{u}^{(0)}\mathfrak{u}^{(1)}\dots\mathfrak{u}^{(n)}$ *of an infinite word* $\mathfrak{u}_\infty$*, we have* $[\mathfrak{u}_\infty \models_\omega \varphi]$ *iff* $\exists^\infty k.\mathfrak{u}^{(0\dots k)} \models_{\mathsf{GF}} \varphi$ *for every* RV^∞–$\mathsf{TL}_{\mathsf{GF}}$ *formula* φ*.*

Hence, $[\mathfrak{u}_\infty \models_\omega \varphi]$ iff $\lim\limits_{n\to\infty} [\mathfrak{u}^{(1\dots n)} \models_{\mathsf{GF}} \varphi] \notin \{\bot_{\mathsf{GF}}, 0\}$. This means, that either (1) no limit exists or (2) the limit exists and is neither $\bot_{\mathsf{GF}}$ nor 0. In case (1) holds, the result of the evaluation must oscillate between the two possible truth values, hence $\top_{\mathsf{GF}}$ holds infinitely often (note that 1 is a limit of the evaluation). If (2) holds, the limit exists and is neither $\bot_{\mathsf{GF}}$ nor 0, hence either $\top_{\mathsf{GF}}$ must hold infinitely often or 1 holds from a certain point on.

4.5 The Temporal Logic RV^∞–$\mathsf{TL}_{\mathsf{Streett}}$

We now consider the most expressive logic RV^∞–$\mathsf{TL}_{\mathsf{Streett}}$ that is obtained from $\mathsf{TL}_{\mathsf{Streett}}$. Looking at the grammar of $\mathsf{TL}_{\mathsf{Streett}}$, one sees that this logic is a positive boolean combination of $\mathsf{TL}_{\mathsf{FG}}$ and $\mathsf{TL}_{\mathsf{GF}}$ formulas. Hence, in the following we assume that our formula is given in conjunctive normal form, meaning that we have a formula of the following form:

$$\bigwedge_{i=0}^{k} \left(\bigvee_{j=0}^{m} \varphi_{i,j} \vee \bigvee_{j=0}^{n} \psi_{i,j} \right)$$

where every $\varphi_j \in \mathsf{TL_{FG}}$ and every $\psi_j \in \mathsf{TL_{GF}}$. This means that for every i we have $\left(\bigvee_{j=0}^{m} \varphi_{i,j}\right) \in \mathsf{TL_{FG}}$ and $\left(\bigvee_{j=0}^{n} \psi_{i,j}\right) \in \mathsf{TL_{GF}}$. Thus, we may even assume that our formula has the form: $\bigwedge_{i=0}^{k} \varphi_i \vee \psi_i$ where $\varphi_i \in \mathsf{TL_{FG}}$ and $\psi_i \in \mathsf{TL_{GF}}$. Hence, we can restrict ourself to formulae of that type, since every formula from $\mathsf{TL_{Streett}}$ can be brought into the desired form. To formally define a semantics for these formulae, we introduce first the Streett-k class:

Definition 11. *A* $\mathsf{TL_{Streett\text{-}k}}$ *formula is a formula of the form* $\bigwedge_{i=0}^{k} \varphi_i \vee \psi_i$*, where each* $\varphi_i \in \mathsf{TL_{FG}}$ *and each* $\psi_i \in \mathsf{TL_{GF}}$*.*

Restricting our attention first to $\mathsf{TL_{Streett\text{-}1}}$-formulas, a straightforward definition for their runtime semantics is given as follows:

Definition 12 (Semantics of $\mathsf{RV^{\infty}}$–$\mathsf{TL_{Streett\text{-}1}}$). *Let* $\mathfrak{u} = \mathfrak{u}^{(0)}\mathfrak{u}^{(1)}\ldots\mathfrak{u}^{(n)} \in \Sigma^*$ *denote a finite path of length* $n+1$*. The truth value of a* $\mathsf{TL_{Streett\text{-}1}}$ *formula* $\varphi \vee \psi$ *wrt.* $\mathfrak{u}$*, denoted with* $[\mathfrak{u} \models_{\mathsf{Street\text{-}1}} \varphi]$*, is defined as follows:*

$$[\mathfrak{u} \models_{\mathsf{Street\text{-}1}} \varphi \vee \psi] = \begin{cases} 1 & \text{if } \forall \mathfrak{w} \in \Sigma^{\omega} : \mathfrak{u}\mathfrak{w} \models_{\omega} \varphi \vee \psi \\ 0 & \text{if } \forall \mathfrak{w} \in \Sigma^{\omega} : \mathfrak{u}\mathfrak{w} \not\models_{\omega} \varphi \vee \psi \\ \top_{\mathsf{FG}} & \text{if } [\mathfrak{u} \models_{\mathsf{FG}} \varphi] \\ \top_{\mathsf{GF}} & \text{if } [\mathfrak{u} \not\models_{\mathsf{FG}} \varphi] \text{ and} \\ & \quad \exists t \leq n. \begin{pmatrix} \mathfrak{u}^{(0\ldots t)} \models_{\mathsf{GF}} \psi \text{ and} \\ \forall t \leq t' < n. [\mathfrak{u} \models_{\mathsf{Street\text{-}1}} \varphi \vee \psi] \neq \top_{\mathsf{GF}} \end{pmatrix} \\ \bot & \text{else} \end{cases}$$

Hence, if from a certain point on the so-far read prefix invariantly evaluates to $\top_{\mathsf{FG}}$, we can be sure that the corresponding φ-formula from $\mathsf{TL_{FG}}$ is invariantly satisfied. If, on the other hand, this does not hold, and we have detected that at some point $t \leq n$ the following holds: $\mathfrak{u}^{(0\ldots t)} \models_{\mathsf{GF}} \psi$ and this 'good' event has not been registered, i. e. for all values between t and n we have $[\mathfrak{u} \models_{\mathsf{Street\text{-}1}} \varphi \vee \psi] \neq \top_{\mathsf{GF}}$, then this 'good' event must be reported in the current step. Accordingly, if $\top_{\mathsf{GF}}$ holds infinitely often, φ need not hold, but we know from Theorem 2 that in that case ψ holds in the limit. Hence, the following theorem immediately follows:

Theorem 3. *Given a finite prefix* $\mathfrak{u} = \mathfrak{u}^{(0)}\mathfrak{u}^{(1)}\ldots\mathfrak{u}^{(n)}$ *of an infinite word* $\mathfrak{u}_{\infty}$*, the following holds for the semantics of* $\mathsf{RV^{\infty}}$–$\mathsf{TL_{Streett\text{-}1}}$*:*

$$[\mathfrak{u}_{\infty} \models_{\omega} \varphi] \text{ iff } \begin{pmatrix} \exists^{\infty} k. [\mathfrak{u}^{(0\ldots k)} \models_{\mathsf{Street\text{-}1}} \varphi] = \top_{\mathsf{GF}} \text{ or} \\ \nexists^{\infty} k. [\mathfrak{u}^{(0\ldots k)} \models_{\mathsf{Street\text{-}1}} \varphi] \notin \{\top_{\mathsf{FG}}, 1\} \end{pmatrix}$$

Hence, $[\mathfrak{u}_{\infty} \models_{\omega} \varphi]$ holds iff $\lim_{n\to\infty} [\mathfrak{u}^{(0\ldots n)} \models_{\mathsf{Street\text{-}1}} \varphi] \notin \{\bot, 0\}$ holds which means that either no limit exists (i.e., $\top_{\mathsf{GF}}$ holds infinitely often), or the limit is in $\{1, \top_{\mathsf{FG}}, \top_{\mathsf{GF}}\}$.

Finally, we can easily generalize this result to $\mathsf{RV^{\infty}}$–$\mathsf{TL_{Streett\text{-}}k}$:

Definition 13 (Semantics of RV^{∞}–$\mathsf{TL}_{\mathsf{Streett\text{-}k}}$). *Let* $\mathfrak{u} = \mathfrak{u}^{(0)}\mathfrak{u}^{(1)}\ldots\mathfrak{u}^{(n)} \in \Sigma^*$ *denote a finite path of length* $n+1$*. The truth value of a* $\mathsf{TL}_{\mathsf{Streett\text{-}k}}$ *formula* $\bigwedge_{i=0}^{k-1} \varphi_i \vee \psi_i$ *wrt.* $\mathfrak{u}$*, denoted with* $[\mathfrak{u} \models_{\mathsf{Street\text{-}k}} \varphi]$*, is a truth value from the domain* $(\mathbb{B}_5)^k$ *given by:*

$$[\mathfrak{u} \models_{\mathsf{Street\text{-}k}} \varphi \vee \psi] = [\mathfrak{u} \models_{\mathsf{Street\text{-}1}} \varphi_0 \vee \psi_0] \times \cdots \times [\mathfrak{u} \models_{\mathsf{Street\text{-}1}} \varphi_{k-1} \vee \psi_{k-1}]$$

5 Conclusion

In this paper, we show that the semantics for LTL on finite paths used in runtime verification so-far have certain deficiencies, in particular, they do not always converge to the truth values of infinite paths. Therefore, we defined a new semantics for LTL on finite paths that is asymptotically correct in this sense. To this end, we considered the temporal logic hierarchy of Manna and Pnueli [5,16] and developed specialized semantics for each temporal logic of this hierarchy. All classes are evaluated over a different set of truth values which leads to the surprising result that for the most expressive logic $\mathsf{TL}_{\mathsf{Streett}}$ of the hierarchy, we need a n-tuple of five-valued truth values where n is the number of clauses in the conjunctive normal form of the formula. It would be interesting to investigate whether this is unavoidable. More precisely: are there formulas in $\mathsf{TL}_{\mathsf{Streett}}$ such that an asymptotically correct semantics will need at least 5^n different truth values? We speculate that this is the case and that this question is related to the Rabin/Streett index of the formula.

References

1. Armoni, R., Bustan, D., Kupferman, O., Vardi, M.: Resets vs. Aborts in Linear Temporal Logic. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 65–80. Springer, Heidelberg (2003)
2. Bauer, A., Leucker, M., Schallhart, C.: The Good, the Bad, and the Ugly, But How Ugly Is Ugly? In: Sokolsky, O., Taşıran, S. (eds.) RV 2007. LNCS, vol. 4839, pp. 126–138. Springer, Heidelberg (2007)
3. Bauer, A., Leucker, M., Schallhart, C.: Comparing LTL semantics for runtime verification. Journal of Logic and Computation 20(3), 651–674 (2010)
4. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for LTL and TLTL. ACM Transactions on Software Engineering and Methodology (2011)
5. Chang, E., Manna, Z., Pnueli, A.: Characterization of Temporal Property Classes. In: Kuich, W. (ed.) ICALP 1992. LNCS, vol. 623, pp. 474–486. Springer, Heidelberg (1992)
6. Eisner, C., Fisman, D., Havlicek, J., Lustig, Y., McIsaac, A., van Campenhout, D.: Reasoning with Temporal Logic on Truncated Paths. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 27–39. Springer, Heidelberg (2003)
7. Emerson, E.: Temporal and modal logic. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science: Formal Models and Semantics, vol. B, ch.16, pp. 995–1072. Elsevier (1990)
8. Falcone, Y., Fernandez, J.-C., Mounier, L.: What can you verify and enforce at runtime? Research Report TR-2010-5, Verimag (January 2010)

9. Maler, O., Pnueli, A.: Timing Analysis of Asynchronous Circuits Using Timed Automata. In: Camurati, P.E., Eveking, H. (eds.) CHARME 1995. LNCS, vol. 987, pp. 189–205. Springer, Heidelberg (1995)
10. Miyano, S., Hayashi, T.: Alternating automata on ω-words. Theoretical Computer Science (TCS) 32, 321–330 (1984)
11. Morgenstern, A., Schneider, K., Lamberti, S.: Generating deterministic ω-automata for most LTL formulas by the breakpoint construction. In: Scholl, C., Disch, S. (eds.) Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV), Freiburg, Germany, pp. 119–128. Shaker (2008)
12. Pnueli, A.: The temporal logic of programs. In: Foundations of Computer Science (FOCS), pp. 46–57. IEEE Computer Society, Providence (1977)
13. Pnueli, A., Zaks, A.: PSL Model Checking and Run-Time Verification Via Testers. In: Misra, J., Nipkow, T., Karakostas, G. (eds.) FM 2006. LNCS, vol. 4085, pp. 573–586. Springer, Heidelberg (2006)
14. Ruf, J., Hoffmann, D., Kropf, T., Rosenstiel, W.: Simulation-guided property checking based on a multi-valued AR-automata. In: Design, Automation and Test in Europe (DATE), Munich, Germany, pp. 742–748. ACM (2001)
15. Schneider, K.: Improving Automata Generation for Linear Temporal Logic by Considering the Automaton Hierarchy. In: Nieuwenhuis, R., Voronkov, A. (eds.) LPAR 2001. LNCS (LNAI), vol. 2250, pp. 39–54. Springer, Heidelberg (2001)
16. Schneider, K.: Verification of Reactive Systems – Formal Methods and Algorithms. Texts in Theoretical Computer Science (EATCS Series). Springer, Heidelberg (2003)

On the Domain and Dimension Hierarchy of Matrix Interpretations

Friedrich Neurauter* and Aart Middeldorp

Institute of Computer Science, University of Innsbruck, Austria

Abstract. Matrix interpretations are a powerful technique for proving termination of term rewrite systems. Depending on the underlying domain of interpretation, one distinguishes between matrix interpretations over the real, rational and natural numbers. In this paper we clarify the relationship between all three variants, showing that matrix interpretations over the reals are more powerful than matrix interpretations over the rationals, which are in turn more powerful than matrix interpretations over the natural numbers. We also clarify the ramifications of matrix dimension on termination proving power. To this end, we establish a hierarchy of matrix interpretations with respect to matrix dimension and show it to be infinite, with each level properly subsuming its predecessor.

Keywords: term rewriting, termination, matrix interpretations.

1 Introduction

Since their inception in 2006, matrix interpretations have evolved into one of the most important (that is, powerful) methods for termination analysis and complexity analysis of term rewrite systems. While originally introduced by Hofbauer and Waldmann as a stand-alone method for termination proofs in the context of string rewriting [13, 14], allowing them to solve challenging termination problems like $\{\mathsf{aa} \to \mathsf{bc}, \mathsf{bb} \to \mathsf{ac}, \mathsf{cc} \to \mathsf{ab}\}$, problem #104 on the RTA list of open problems,[1] it was not long until Endrullis *et al.* [6] generalized (one particular instance of) the matrix method to term rewriting and also incorporated it into the dependency pair (DP) framework [3, 9–11, 23], the state-of-the-art framework for establishing termination of term rewrite systems.

The matrix method is based on the well-known paradigm of interpreting terms into a domain equipped with a suitable well-founded order. In the original approach of [6], the authors consider the set of vectors of natural numbers as underlying domain, together with a well-founded order that relates two vectors if and only if there is a strict decrease in the respective first components and a weak decrease in all other components. Function symbols are interpreted by suitable linear mappings represented by square matrices of natural numbers. Recently, another generalization appeared in [5] that employs matrices of natural

* Friedrich Neurauter is supported by a grant of the University of Innsbruck.

[1] http://rtaloop.mancoosi.univ-paris-diderot.fr

N. Bjørner and A. Voronkov (Eds.): LPAR-18, LNCS 7180, pp. 320–334, 2012.

numbers as underlying domain and interprets each function symbol by a linear matrix polynomial. In principle, this approach also allows for non-linear matrix polynomials. In [1, 7, 24] the method of Endrullis *et al.* was lifted to the non-negative rational and real (algebraic) numbers using the same technique that was already used to lift polynomial interpretations from the natural numbers to the rationals and reals (cf. [12]). Thus, one distinguishes three variants of matrix interpretations, matrix interpretations over the real, rational and natural numbers. So the obvious question is: what is their relationship with regard to termination proving power?

As a starting point, it is instructive to restrict to one-dimensional matrix interpretations, that is, *linear* polynomial interpretations, for which the termination hierarchy is known (cf. [16, 18]) and can be pictured as in Figure 1. That

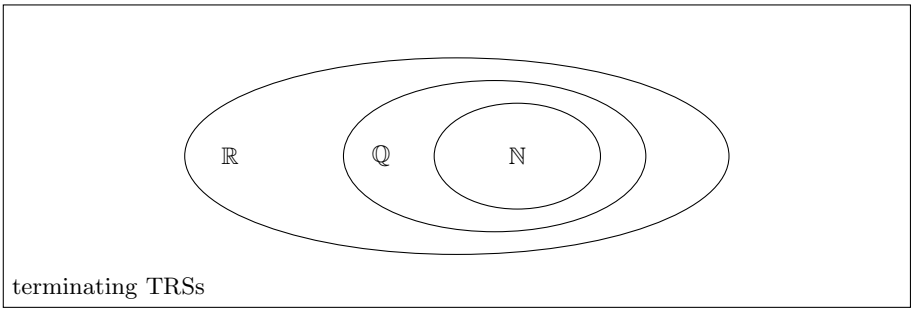


Fig. 1. Linear polynomial interpretations

is, linear polynomial interpretations over the real numbers subsume linear polynomial interpretations over the rational numbers, which in turn subsume linear polynomial interpretations over the natural numbers. Both inclusions are proper. To this end, [16] introduces the rewrite systems $\mathcal{R}_{\mathbb{Q}}$ and $\mathcal{R}_{\mathbb{R}}$, the first of which can be shown terminating by a linear polynomial interpretation over the rational numbers but not over the natural numbers. Similarly, the second system can be shown terminating by a linear polynomial interpretation over the reals but not over the rationals. Unfortunately, the usefulness of both $\mathcal{R}_{\mathbb{Q}}$ and $\mathcal{R}_{\mathbb{R}}$ is limited to dimension one (cf. [17]) because, without restricting the dimension, both systems can be handled with 2-dimensional matrix interpretations over the natural numbers. In this context, we also mention related work appearing in [8], where a relative termination problem in the form of a string rewrite system is presented that can be handled with matrix interpretations over the rationals but not with matrix interpretations over the natural numbers. However, *relative* termination is essential in this example because the relative component is the key ingredient for precluding matrix interpretations over the natural numbers. As the latter component consists of a single non-terminating rule, the entire example does not readily generalize to (real) termination problems. Besides, there is no evidence in [8] demonstrating the benefit of using irrational numbers in matrix

interpretations. Thus, we conclude that new techniques are required to clarify the relationship between the aforementioned variants of matrix interpretations.

One of the main results of this paper is to show that the termination hierarchy depicted in Figure 1 does in fact extend from one-dimensional matrix interpretations to arbitrary matrix interpretations. That is, matrix interpretations over the reals are more powerful with respect to proving termination than matrix interpretations over the rationals, which are in turn more powerful than matrix interpretations over the natural numbers. In particular, we show that this relationship does not only hold in the context of direct termination (using matrix interpretations as a stand-alone method) but also in the setting of the DP framework. Moreover, our results point out the limitations of a recent attempt [17] to simulate matrix interpretations over the rationals with matrix interpretations over the natural numbers (of higher dimension).

We also investigate the ramifications of matrix dimension on termination proving power. Clearly, by increasing the dimension, one can never lose power (in theory; in practice the increased search space may prohibit finding a termination proof). But what is the exact shape of the inherent dimension hierarchy? A partial answer to this question was given in [8], where the authors show that the hierarchy is infinite. Yet no exact information is provided as to which levels are actually inhabited. We close this gap in the second part of this paper, thus giving a complete answer to the question raised above. To this end, we establish a hierarchy of matrix interpretations with respect to matrix dimension and show it to be infinite, with each level properly subsuming its predecessor. In other words, we show that matrix interpretations of dimension $(n+1)$ are strictly more powerful for proving termination than n-dimensional matrix interpretations (for any $n \geqslant 1$). The construction we use for this purpose is entirely different from the one proposed in [8]. Apart from the fact that it allows to infer the exact shape of the dimension hierarchy, it has the additional advantage that it produces witnesses (that is, rewrite systems) that are substantially smaller than the ones of [8]. To be precise, the construction employed in [8] gives rise to a family of string rewrite systems $(\mathcal{S}_d)_{d\geqslant 2}$ having the property that any of its members $\mathcal{S}_{2d}$ (of even index) cannot be handled with matrix interpretations of dimension d or less (as a consequence of the Amitsur-Levitzki theorem [2]), but can be handled with dimension $d' = 2d+3$. Each system $\mathcal{S}_d$ consists of the following rules over the finite alphabet $\Sigma_d = \{\mathsf{s}, \mathsf{1}, \ldots, \mathsf{d}, \mathsf{f}\}$: $\mathsf{s}\, e_k\, \mathsf{f} \to \mathsf{s}\, o_k\, \mathsf{f}$ for all $1 \leqslant k \leqslant \frac{d!}{2}$. Here, $e_1, e_2, \ldots$ $(o_1, o_2, \ldots)$ is any enumeration of even (odd)[2] permutations of the symbols $\{\mathsf{1}, \ldots, \mathsf{d}\}$. Hence, the number of rewrite rules in $\mathcal{S}_d$ exhibits factorial growth in the dimension d. In contrast, the systems created by our approach have constant size and the dimension d' is *optimal*, i.e., $d' = d+1$.

The remainder of this paper is organized as follows. In the next section we recall preliminaries from linear algebra and term rewriting. In particular, we review the matrix method for establishing termination of term rewrite systems. Then, in Section 3, we show that matrix interpretations over the reals are more

[2] A permutation is called even (odd) if it can be written as a composition of an even (odd) number of transpositions.

powerful than matrix interpretations over the rationals, which are in turn more powerful than matrix interpretations over the natural numbers. Subsequently, we present our results on the dimension hierarchy related to matrix interpretations in Section 4, before concluding with suggestions for future research in Section 5.

2 Preliminaries

As usual, we denote by $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$ and $\mathbb{R}$ the sets of natural, integer, rational and real numbers. A real number is said to be *algebraic* if it is a root of a non-zero polynomial in one indeterminate with integer coefficients, otherwise it is said to be *transcendental*. The set of all real algebraic numbers is denoted by $\mathbb{R}_{\mathsf{alg}}$. Given $D \in \{\mathbb{N}, \mathbb{Z}, \mathbb{Q}, \mathbb{R}_{\mathsf{alg}}, \mathbb{R}\}$ and $m \in D$, $>_D$ (resp. $>$ if D is clear from the context) denotes the natural order of the respective domain, $\geqslant_D$ (resp. $\geqslant$) its reflexive closure, and D_m abbreviates $\{x \in D \mid x \geqslant m\}$; for example, $\mathbb{Q}_0$ ($\mathbb{R}_0$) refers to the set of all non-negative rational (real) numbers.

2.1 Linear Algebra

Let R be a commutative ring (e.g., $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}_{\mathsf{alg}}$, $\mathbb{R}$). The ring of all *n-dimensional square matrices* over R is denoted by $R^{n\times n}$ and the *polynomial ring* in n *indeterminates* $x_1, \ldots, x_n$ by $R[x_1, \ldots, x_n]$. In the special case $n = 1$, any polynomial $p \in R[x]$ can be written as $p(x) = \sum_{k=0}^{d} a_k x^k$ for some $d \in \mathbb{N}$. For the largest k such that $a_k \neq 0$, we call $a_k x^k$ the *leading term* of p, a_k its *leading coefficient* and k its *degree*. The polynomial p is said to be *monic* if its leading coefficient is one. It is said to be *linear*, *quadratic*, *cubic* if its degree is one, two, three.

In case R is equipped with a partial order $\geqslant$, the component-wise extension of this order to $R^{n\times n}$ is also denoted by $\geqslant$. The $n \times n$ *identity matrix* is denoted by I_n and the $n \times n$ *zero matrix* by 0_n. We simply write I and 0 if n is clear from the context. We say that a matrix A is *non-negative* if $A \geqslant 0$ and denote the set of all non-negative n-dimensional square matrices of $\mathbb{Z}^{n\times n}$ by $\mathbb{N}^{n\times n}$. As usual, we write A^{T} for the *transpose* of a matrix (vector) A.

For a square matrix $A \in R^{n\times n}$, the *characteristic polynomial* $\chi_A(\lambda)$ is defined as $\det(\lambda I_n - A)$, where det denotes the (matrix) determinant. It is a monic polynomial of degree n with coefficients in R. The equation $\chi_A(\lambda) = 0$ is called the *characteristic equation* of A. The solutions of this equation, that is, the *roots* of $\chi_A(\lambda)$, are precisely the *eigenvalues* of A. If R is a subset of an algebraically closed field (where each polynomial of degree n with coefficients in the field is guaranteed to have exactly n roots), then A has exactly n (not necessarily distinct) eigenvalues in this field.

We say that a polynomial $p \in R[x]$ *annihilates* a square matrix $A \in R^{n\times n}$ if $p(A) = 0$. The Cayley-Hamilton theorem [21] states that A satisfies its own characteristic equation, that is, χ_A annihilates A. Let R be a field and consider the set $\{\, p \in R[x] \mid p(A) = 0 \,\}$ of annihilating polynomials of $A \in R^{n\times n}$. This set is generated by the *minimal polynomial* $\mathrm{m}_A(x)$ of A, which is the unique monic polynomial of minimum degree that annihilates A. Any polynomial that

annihilates A is a (polynomial) multiple of $\mathrm{m}_A(x)$. In other words, if $p(A) = 0$ for $p \in R[x]$, then $\mathrm{m}_A(x)$ divides $p(x)$. In particular, $\mathrm{m}_A(x)$ divides the characteristic polynomial of A, and $\mathrm{m}_A(\lambda) = 0$ if and only if λ is an eigenvalue of A (cf. [15]).

2.2 Term Rewriting

We assume familiarity with the basics of term rewriting [4, 22]. Let $\mathcal{V}$ denote a countably infinite set of variables and $\mathcal{F}$ a *signature*, that is, a set of function symbols equipped with fixed arities. The set of *terms* over $\mathcal{F}$ and $\mathcal{V}$ is denoted by $\mathcal{T}(\mathcal{F}, \mathcal{V})$. A *rewrite rule* is a pair of terms written as $\ell \to r$ such that ℓ is not a variable and all variables of r are contained in ℓ. A *term rewrite system* (TRS for short) $\mathcal{R}$ over $\mathcal{T}(\mathcal{F}, \mathcal{V})$ is a finite set of rewrite rules. The rewrite relation induced by $\to$ is denoted by $\to_{\mathcal{R}}$. As usual, $\to_{\mathcal{R}}^*$ denotes the reflexive transitive closure of $\to_{\mathcal{R}}$.

2.3 Monotone Algebras and Matrix Interpretations

We use the following notation for monotone algebras [6]. An *$\mathcal{F}$-algebra* $\mathcal{A}$ consists of a non-empty *carrier* set A and a collection of *interpretation functions* $f_{\mathcal{A}} \colon A^k \to A$ for each k-ary function symbol $f \in \mathcal{F}$. By $[\alpha]_{\mathcal{A}}(\cdot)$ we denote the usual evaluation function of $\mathcal{A}$ with respect to a variable assignment $\alpha \colon \mathcal{V} \to A$. A *weakly monotone $\mathcal{F}$-algebra* $(\mathcal{A}, >, \geqslant)$ is an $\mathcal{F}$-algebra $\mathcal{A}$ together with two binary relations $>$ and $\geqslant$ on A such that $>$ is well-founded, $> \cdot \geqslant \subseteq >$ and for each $f \in \mathcal{F}$, $f_{\mathcal{A}}$ is monotone with respect to $\geqslant$ (in all arguments). If, in addition, each $f_{\mathcal{A}}$ is monotone with respect to $>$, then we speak of an *extended monotone algebra*. Any monotone algebra $(\mathcal{A}, >, \geqslant)$ (or just $\mathcal{A}$ if $>$ and $\geqslant$ are clear from the context) induces the following relations on $\mathcal{T}(\mathcal{F}, \mathcal{V})$:

- $s >_{\mathcal{A}} t$ if and only if $[\alpha]_{\mathcal{A}}(s) > [\alpha]_{\mathcal{A}}(t)$ for all assignments α, and
- $s \geqslant_{\mathcal{A}} t$ if and only if $[\alpha]_{\mathcal{A}}(s) \geqslant [\alpha]_{\mathcal{A}}(t)$ for all assignments α.

We say that a monotone algebra $\mathcal{A}$ is *compatible* with a rewrite rule $\ell \to r$ if $\ell >_{\mathcal{A}} r$, it is said to be *weakly compatible* if $\ell \geqslant_{\mathcal{A}} r$. In the same vein, we say that $\mathcal{A}$ is (weakly) compatible with a TRS $\mathcal{R}$ if it is (weakly) compatible with all rewrite rules of $\mathcal{R}$. We use the following abbreviations: $\mathcal{R} \subseteq >_{\mathcal{A}}$ for compatibility and $\mathcal{R} \subseteq \geqslant_{\mathcal{A}}$ for weak compatibility.

It is well-known that a TRS is terminating if and only if there is an extended monotone algebra that is compatible with it (cf. [6, Theorem 2]). Moreover, extended monotone algebras facilitate *incremental termination proofs* (cf. [6, Theorem 3]). To this end, let $\mathcal{A}$ be an extended monotone algebra and suppose $\mathcal{R}$ is a TRS such that $\mathcal{R} \subseteq \geqslant_{\mathcal{A}}$ and $\mathcal{S} \subseteq >_{\mathcal{A}}$ for some non-empty subset $\mathcal{S}$ of $\mathcal{R}$. Then, after removing all $\mathcal{S}$-rules from $\mathcal{R}$, termination of $\mathcal{R} \setminus \mathcal{S}$ implies termination of $\mathcal{R}$. Thus, one is free to choose a different extended monotone algebra for the remaining rules $\mathcal{R} \setminus \mathcal{S}$. This process is continued until eventually all rewrite rules have been removed.

Weakly monotone algebras play an important role in the context of termination analysis in the DP framework. In this modular framework, the problem

of establishing termination of a TRS is typically split into several subproblems called *DP problems*. A DP problem is a pair $(\mathcal{P}, \mathcal{S})$, where $\mathcal{P}$ and $\mathcal{S}$ are finite sets of rewrite rules such that the root symbols of the rules in $\mathcal{P}$ neither occur in $\mathcal{S}$ nor in proper subterms of the left- and right-hand sides of the rules in $\mathcal{P}$. In the sequel, we sometimes write $(\_, \mathcal{S})$ to indicate that we are only interested in the second component of a DP problem. A *DP processor* is a mapping that takes a DP problem as input and returns a set of DP problems as output. In the context of this paper, we only consider DP processors based on *reduction pairs*. Given a DP problem $(\mathcal{P}, \mathcal{S})$, the aim of such a processor is to return a simplified version of its input by removing rules from the $\mathcal{P}$ component. It is well-known that weakly monotone algebras give rise to reduction pairs. One can use them to simplify DP problems as follows. Let $\mathcal{A}$ be a weakly monotone algebra and $(\mathcal{P}, \mathcal{S})$ a DP problem. If $\mathcal{P} \cup \mathcal{S} \subseteq \geqslant_{\mathcal{A}}$ and $\mathcal{P}' \subseteq >_{\mathcal{A}}$ for some non-empty subset $\mathcal{P}'$ of $\mathcal{P}$, then one may remove all rules of $\mathcal{P}'$ from $\mathcal{P}$, thus simplifying the original DP problem to the DP problem $(\mathcal{P} \setminus \mathcal{P}', \mathcal{S})$ containing less rules. In this situation, we say that the weakly monotone algebra $\mathcal{A}$ *succeeds on the DP problem* $(\mathcal{P}, \mathcal{S})$, otherwise it *fails*.

We define matrix interpretations as follows. For *matrix interpretations over* $\mathbb{R}$, we fix a dimension $n \in \mathbb{N} \setminus \{0\}$, some positive real number δ and use the set $\mathbb{R}_0^n$ as the carrier of an algebra $\mathcal{M}$, together with the orders $>_\delta$ and $\geqslant$ on $\mathbb{R}_0^n$:

$$(x_1, \dots, x_n)^{\mathrm{T}} >_\delta (y_1, \dots, y_n)^{\mathrm{T}} \iff x_1 >_{\mathbb{R},\delta} y_1 \wedge x_i \geqslant_{\mathbb{R}} y_i \text{ for } i = 2, \dots, n$$
$$(x_1, \dots, x_n)^{\mathrm{T}} \geqslant (y_1, \dots, y_n)^{\mathrm{T}} \iff x_i \geqslant_{\mathbb{R}} y_i \text{ for } i = 1, \dots, n$$

Here, $x >_{\mathbb{R},\delta} y$ if and only if $x \geqslant_{\mathbb{R}} y + \delta$. Each k-ary function symbol f is interpreted by a linear function of the shape

$$f_{\mathcal{M}} \colon (\mathbb{R}_0^n)^k \to \mathbb{R}_0^n, (\boldsymbol{x}_1, \dots, \boldsymbol{x}_k) \mapsto F_1\boldsymbol{x}_1 + \cdots + F_k\boldsymbol{x}_k + \boldsymbol{f}$$

where $\boldsymbol{x}_1, \dots, \boldsymbol{x}_k$ are (column) vectors of variables, $F_1, \dots, F_k \in \mathbb{R}_0^{n \times n}$ and $\boldsymbol{f} \in \mathbb{R}_0^n$. In this way, $(\mathcal{M}, >_\delta, \geqslant)$ forms a weakly monotone algebra. If, in addition, the top left entry $(F_i)_{11}$ of each matrix F_i is at least one, then we call $\mathcal{M}$ a *monotone* matrix interpretation over $\mathbb{R}$, in which case $(\mathcal{M}, >_\delta, \geqslant)$ becomes an extended monotone algebra. Note that in any case we have $>_{\mathcal{M}} \subseteq \geqslant_{\mathcal{M}}$ since $>_\delta \subseteq \geqslant$ (independently of δ).

We obtain matrix interpretations over $\mathbb{R}_{\mathsf{alg}}$ by restricting the carrier to the set of vectors of non-negative real algebraic numbers. Similarly, matrix interpretations over $\mathbb{Q}$ operate on the carrier $\mathbb{Q}_0^n$. For matrix interpretations over $\mathbb{N}$, one uses the carrier $\mathbb{N}^n$ and $\delta = 1$, such that

$$(x_1, \dots, x_n)^{\mathrm{T}} >_\delta (y_1, \dots, y_n)^{\mathrm{T}} \iff x_1 >_{\mathbb{N}} y_1 \wedge x_i \geqslant_{\mathbb{N}} y_i \text{ for } i = 2, \dots, n$$

According to [20], matrix interpretations over $\mathbb{R}$ are equivalent to matrix interpretations over $\mathbb{R}_{\mathsf{alg}}$ with respect to proving termination. So transcendental numbers are not relevant for termination proofs based on matrix interpretations. Nevertheless, for the sake of brevity of notation, we will stick to the term "matrix interpretations over the real numbers" for the rest of this paper.

3 The Domain Hierarchy

In this section we show that matrix interpretations over the real numbers are more powerful with respect to proving termination than matrix interpretations over the rational numbers, which are in turn more powerful than matrix interpretations over the natural numbers. To begin with, we show that matrix interpretations over $\mathbb{R}$ subsume matrix interpretations over $\mathbb{Q}$, which in turn subsume matrix interpretations over $\mathbb{N}$. Then, in Sections 3.1 and 3.2, both inclusions are proved to be proper.

Lemma 1. *Let $\mathcal{M}$ be an n-dimensional matrix interpretation over $\mathbb{N}$ (not necessarily monotone), and let $\mathcal{S}_1$ and $\mathcal{S}_2$ be finite sets of rewrite rules such that $\mathcal{S}_1 \subseteq >_{\mathcal{M}}$ and $\mathcal{S}_2 \subseteq \geqslant_{\mathcal{M}}$. Then there exists an n-dimensional matrix interpretation $\mathcal{N}$ over $\mathbb{Q}$ such that $\mathcal{S}_1 \subseteq >_{\mathcal{N}}$ and $\mathcal{S}_2 \subseteq \geqslant_{\mathcal{N}}$. Moreover, $\mathcal{N}$ is monotone if and only if $\mathcal{M}$ is monotone.*

Proof. Let $\mathcal{F}$ denote the signature associated with $\mathcal{S}_1 \cup \mathcal{S}_2$. Then, by assumption, $\mathcal{M}$ associates each k-ary function symbol $f \in \mathcal{F}$ with a linear function $f_{\mathcal{M}}(\boldsymbol{x}_1, \dots, \boldsymbol{x}_k) = F_1\boldsymbol{x}_1 + \cdots + F_k\boldsymbol{x}_k + \boldsymbol{f}$, where $F_1, \dots, F_k \in \mathbb{N}^{n\times n}$ and $\boldsymbol{f} \in \mathbb{N}^n$, such that $\mathcal{S}_1 \subseteq >_{\mathcal{M}}$ and $\mathcal{S}_2 \subseteq \geqslant_{\mathcal{M}}$. Based on this interpretation, we define the matrix interpretation $\mathcal{N}$ by letting $\delta = 1$ and taking the same interpretation functions, i.e., $f_{\mathcal{N}}(\boldsymbol{x}_1, \dots, \boldsymbol{x}_k) = f_{\mathcal{M}}(\boldsymbol{x}_1, \dots, \boldsymbol{x}_k)$ for all $f \in \mathcal{F}$. Then $\mathcal{N}$ is well-defined, and it is monotone if and only if $\mathcal{M}$ is monotone.

As to compatibility of $\mathcal{N}$ with $\mathcal{S}_1$, let us consider an arbitrary rewrite rule $\ell \to r \in \mathcal{S}_1$ and show that $\ell >_{\mathcal{M}} r$ implies $\ell >_{\mathcal{N}} r$, i.e., $[\alpha]_{\mathcal{N}}(\ell) >_\delta [\alpha]_{\mathcal{N}}(r)$ for all variable assignments α. Because of linearity of the interpretation functions, we can write $[\alpha]_{\mathcal{N}}(\ell) = L_1\boldsymbol{x}_1 + \cdots + L_m\boldsymbol{x}_m + \boldsymbol{\ell}$ and $[\alpha]_{\mathcal{N}}(r) = R_1\boldsymbol{x}_1 + \cdots + R_m\boldsymbol{x}_m + \boldsymbol{r}$, where $x_1, \dots, x_m$ are the variables occurring in ℓ, r and $\boldsymbol{x}_i = \alpha(x_i)$ for $i = 1, \dots, m$. Thus, it remains to show that the inequality

$$L_1\boldsymbol{x}_1 + \cdots + L_m\boldsymbol{x}_m + \boldsymbol{\ell} >_\delta R_1\boldsymbol{x}_1 + \cdots + R_m\boldsymbol{x}_m + \boldsymbol{r}$$

holds for all $\boldsymbol{x}_1, \dots, \boldsymbol{x}_m \in \mathbb{Q}_0^n$. This is exactly the case if $L_i \geqslant R_i$ for $i = 1, \dots, m$ and $\boldsymbol{\ell} >_\delta \boldsymbol{r}$, i.e., $\ell_i \geqslant r_i$ for $i = 2, \dots, n$ and $\ell_1 \geqslant r_1 + \delta = r_1 + 1$. Indeed, all these conditions follow from compatibility of $\mathcal{M}$ with $\ell \to r$ because, by the same reasoning as above (and since the interpretation functions of $\mathcal{M}$ and $\mathcal{N}$ coincide), $\ell >_{\mathcal{M}} r$ holds in $(\mathcal{M}, >, \geqslant)$ if and only if

$$L_1\boldsymbol{x}_1 + \cdots + L_m\boldsymbol{x}_m + \boldsymbol{\ell} > R_1\boldsymbol{x}_1 + \cdots + R_m\boldsymbol{x}_m + \boldsymbol{r}$$

holds for all $\boldsymbol{x}_1, \dots, \boldsymbol{x}_m \in \mathbb{N}^n$, which implies $L_i \geqslant R_i$ for $i = 1, \dots, m$ and $\boldsymbol{\ell} > \boldsymbol{r}$, i.e., $\ell_i \geqslant r_i$ for $i = 2, \dots, n$ and $\ell_1 >_{\mathbb{N}} r_1$, the latter being equivalent to $\ell_1 \geqslant r_1 + 1$ as $\boldsymbol{\ell}, \boldsymbol{r} \in \mathbb{N}^n$. This shows compatibility of $\mathcal{N}$ with $\mathcal{S}_1$. Weak compatibility with $\mathcal{S}_2$ follows in the same way. □

The essence of the proof of this lemma is that any matrix interpretation over $\mathbb{N}$ can be conceived as a matrix interpretation over $\mathbb{Q}$. Likewise, any matrix interpretation over $\mathbb{Q}$ can be conceived as a matrix interpretation over $\mathbb{R}$.

Lemma 2. *Let $\mathcal{M}$ be an n-dimensional matrix interpretation over $\mathbb{Q}$ (not necessarily monotone), and let $\mathcal{S}_1$ and $\mathcal{S}_2$ be finite sets of rewrite rules such that $\mathcal{S}_1 \subseteq >_{\mathcal{M}}$ and $\mathcal{S}_2 \subseteq \geqslant_{\mathcal{M}}$. Then there exists an n-dimensional matrix interpretation $\mathcal{N}$ over $\mathbb{R}$ such that $\mathcal{S}_1 \subseteq >_{\mathcal{N}}$ and $\mathcal{S}_2 \subseteq \geqslant_{\mathcal{N}}$. Moreover, $\mathcal{N}$ is monotone if and only if $\mathcal{M}$ is monotone.*

Proof. Similar to the proof of Lemma 1, with $\mathcal{N}$ defined as follows: $\delta_{\mathcal{N}} = \delta_{\mathcal{M}} = \delta$ and $f_{\mathcal{N}}(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k) = f_{\mathcal{M}}(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k)$ for all $f \in \mathcal{F}$. □

As an immediate consequence of the previous lemmata, we obtain the following corollary stating that matrix interpretations over $\mathbb{N}$ are no more powerful than matrix interpretations over $\mathbb{Q}$, which are in turn no more powerful than matrix interpretations over $\mathbb{R}$.

Corollary 3. *Let $\mathcal{R}$ be a TRS and $(\mathcal{P}, \mathcal{S})$ a DP problem.*

1. *If there is an (incremental) termination proof for $\mathcal{R}$ using monotone matrix interpretations over $\mathbb{N}$ (resp. $\mathbb{Q}$), then there is also one using monotone matrix interpretations over $\mathbb{Q}$ (resp. $\mathbb{R}$).*
2. *If a matrix interpretation over $\mathbb{N}$ (resp. $\mathbb{Q}$) succeeds on $(\mathcal{P}, \mathcal{S})$, then there is also a matrix interpretation over $\mathbb{Q}$ (resp. $\mathbb{R}$) of the same dimension that succeeds on $(\mathcal{P}, \mathcal{S})$.* □

In the remainder of this section we show that the converse statements do not hold.

3.1 Matrix Interpretations over $\mathbb{Q}$

In order to show that matrix interpretations over $\mathbb{Q}$ are indeed more powerful than matrix interpretations over $\mathbb{N}$, let us first consider the TRS $\mathcal{S}$ consisting of the following rewrite rules:

$$x + \mathsf{a} \to x \tag{1}$$
$$x + \mathsf{a} \to (x + \mathsf{b}) + \mathsf{b} \tag{2}$$
$$\mathsf{a} + x \to x \tag{3}$$
$$\mathsf{a} + x \to \mathsf{b} + (\mathsf{b} + x) \tag{4}$$

This TRS will turn out to be very helpful for our purposes, not only in the current subsection but also in the subsequent one. This is due to the following property, which holds for matrix interpretations over $\mathbb{N}$, $\mathbb{Q}$ and $\mathbb{R}$.

Lemma 4. *Let $\mathcal{M}$ be a matrix interpretation (not necessarily monotone) with carrier set M such that $\mathcal{S} \subseteq \geqslant_{\mathcal{M}}$. Then $+_{\mathcal{M}}(\boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{x} + \boldsymbol{y} + \boldsymbol{v}$, $\boldsymbol{v} \in M$.*

Proof. Without loss of generality, let $+_{\mathcal{M}}(\boldsymbol{x}, \boldsymbol{y}) = A_1\boldsymbol{x} + A_2\boldsymbol{y} + \boldsymbol{v}$, $\boldsymbol{v} \in M$. As $\mathcal{M}$ is weakly compatible with rule (1), we obtain $A_1 \geqslant I$; hence, $A_1^2 \geqslant A_1$ due to non-negativity of A_1. Similarly, by weak compatibility with (2), we infer

$A_1 \geqslant A_1^2$, which implies $A_1^2 = A_1 \geqslant I$ together with the previous result. Yet this means that A_1 must in fact be equal to I. To this end, we observe that $A_1 \geqslant I$ implies $(A_1 - I)^2 \geqslant 0$, which simplifies to $I \geqslant 2A_1 - A_1^2 = A_1$; hence, $A_1 = I$. In the same way, we obtain $A_2 = I$ from the compatibility constraints associated with (3) and (4). □

So in any matrix interpretation that is weakly compatible with the TRS $\mathcal{S}$ the symbol + must be interpreted by a function $+_{\mathcal{M}}(\boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{x} + \boldsymbol{y} + \boldsymbol{v}$ that models addition of two elements of the underlying carrier set (modulo adding a constant). The inherent possibility to count objects can be exploited to show that matrix interpretations over $\mathbb{Q}$ are indeed more powerful than matrix interpretations over $\mathbb{N}$. To this end, we extend the TRS $\mathcal{S}$ with the rules (5) and (6), calling the resulting system $\mathcal{R}_1$:

$$((x + x) + x) + \mathsf{a} \to \mathsf{g}(x + x) \tag{5}$$

$$\mathsf{g}(x + x) \to (x + x) + x \tag{6}$$

By construction, this TRS is not compatible, not even weakly compatible, with any matrix interpretation over $\mathbb{N}$.

Lemma 5. *Let $\mathcal{M}$ be an n-dimensional matrix interpretation (not necessarily monotone) with carrier set M such that $\mathcal{R}_1 \subseteq \geqslant_{\mathcal{M}}$. Then $M \neq \mathbb{N}^n$.*

Proof. As $\mathcal{M}$ is weakly compatible with $\mathcal{R}_1$, it is also weakly compatible with the TRS $\mathcal{S}$. So, by Lemma 4, the function symbol + must be interpreted by $+_{\mathcal{M}}(\boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{x} + \boldsymbol{y} + \boldsymbol{v}$, $\boldsymbol{v} \in M$. Assuming $\mathsf{g}_{\mathcal{M}}(\boldsymbol{x}) = G\boldsymbol{x} + \boldsymbol{g}$ without loss of generality, we obtain $3I \geqslant 2G$ from weak compatibility of $\mathcal{M}$ with (5) and $2G \geqslant 3I$ from weak compatibility with (6); hence, $G = \frac{3}{2}I \notin \mathbb{N}^{n \times n}$. Therefore, $\mathcal{M}$ cannot be a matrix interpretation over $\mathbb{N}$. □

The previous lemma, together with the observation that the TRS $\mathcal{R}_1$ admits a compatible matrix interpretation over $\mathbb{Q}$, directly leads to the main result of this subsection.

Theorem 6.

1. *The TRS $\mathcal{R}_1$ is terminating. In particular, $\mathcal{R}_1$ is compatible with a monotone matrix interpretation over $\mathbb{Q}$.*
2. *There cannot be an (incremental) termination proof of $\mathcal{R}_1$ using only monotone matrix interpretations over $\mathbb{N}$.*
3. *No matrix interpretation over $\mathbb{N}$ succeeds on the DP problem $(\_, \mathcal{R}_1)$.*

Proof. The last two statements are immediate consequences of Lemma 5. As to the first claim, the following monotone one-dimensional matrix interpretation (i.e., linear polynomial interpretation) over $\mathbb{Q}$ is compatible with $\mathcal{R}_1$: $\delta = 1$, $\mathsf{a}_{\mathcal{M}} = 2$, $\mathsf{b}_{\mathcal{M}} = 0$, $\mathsf{g}_{\mathcal{M}}(x) = \frac{3}{2}x + 1$ and $+_{\mathcal{M}}(x, y) = x + y$. □

3.2 Matrix Interpretations over $\mathbb{R}$

Next we show that matrix interpretations over $\mathbb{R}$ are more powerful than matrix interpretations over $\mathbb{Q}$. To this end, we extend the TRS $\mathcal{S}$ of the previous subsection with the rules (7) – (9) and call the resulting system $\mathcal{R}_2$:

$$(x + x) + \mathsf{a} \to \mathsf{k}(\mathsf{k}(x)) \tag{7}$$

$$\mathsf{k}(\mathsf{k}(x)) \to x + x \tag{8}$$

$$\mathsf{k}(x) \to x \tag{9}$$

By construction, this TRS admits only matrix interpretations over $\mathbb{R}$.

Lemma 7. *Let $\mathcal{M}$ be an n-dimensional matrix interpretation (not necessarily monotone) with carrier set M such that $\mathcal{R}_2 \subseteq \geqslant_{\mathcal{M}}$. Then $M \neq \mathbb{N}^n$ and $M \neq \mathbb{Q}_0^n$.*

Proof. As the TRS $\mathcal{S}$ is a subsystem of $\mathcal{R}_2$, $\mathcal{R}_2 \subseteq \geqslant_{\mathcal{M}}$ implies $\mathcal{S} \subseteq \geqslant_{\mathcal{M}}$. Hence, by Lemma 4, the function symbol $+$ must be interpreted by $+_{\mathcal{M}}(\boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{x} + \boldsymbol{y} + \boldsymbol{v}$, $\boldsymbol{v} \in M$. Assuming $\mathsf{k}_{\mathcal{M}}(\boldsymbol{x}) = K\boldsymbol{x} + \boldsymbol{k}$ without loss of generality, the (weak) compatibility constraint associated with rule (7) implies $2I \geqslant K^2$. We also have $K^2 \geqslant 2I$ by weak compatibility with (8) and $K \geqslant I$ due to (9). Hence, the $n \times n$ square matrix K must satisfy the following conditions:

$$K^2 = 2I \qquad \text{and} \qquad K \geqslant I \tag{10}$$

Clearly, for dimension $n = 1$, the unique solution is $K = \sqrt{2}$; in particular, K is not a rational number. In fact, for any dimension $n \geqslant 1$, the unique solution turns out to be $K = \sqrt{2}I$. To this end, let us first show that the conditions given in (10) imply that K is a diagonal matrix. Because of $K \geqslant I$, we can write $K = I + N$ for some non-negative matrix N. Then $K^2 = 2I$ if and only if $N^2 + 2N = I$. Now non-negativity of N implies $I \geqslant N$. Hence, N is a diagonal matrix and therefore also K. So all entries of K^2 are zero except its diagonal entries: $(K^2)_{ii} = K_{ii}^2$ for $i = 1, \ldots, n$. But then K_{ii} must be $\sqrt{2}$ in order to satisfy $K^2 = 2I$ and $K \geqslant I$. In other words, $K = \sqrt{2}I \notin \mathbb{Q}_0^{n \times n}$. Therefore, $\mathcal{M}$ cannot be a matrix interpretation over $\mathbb{N}$ or $\mathbb{Q}$. □

Remark 8. Rule (9) is essential for the statement of Lemma 7. Without it, the conditions given in (10) would turn into $K^2 = 2I$ and $K \geqslant 0$, the conjunction of which is satisfiable over $\mathbb{N}^{n \times n}$; for example, by choosing

$$\mathsf{a}_{\mathcal{M}} = \begin{pmatrix} 2 \\ 1 \end{pmatrix} \quad \mathsf{b}_{\mathcal{M}} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \quad \mathsf{k}_{\mathcal{M}}(\boldsymbol{x}) = \begin{pmatrix} 0 & 2 \\ 1 & 0 \end{pmatrix} \boldsymbol{x} + \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad +_{\mathcal{M}}(\boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{x} + \boldsymbol{y}$$

we obtain a non-monotone 2-dimensional matrix interpretation over $\mathbb{N}$ that is compatible with the TRS $\mathcal{R}_2 \setminus \{(9)\}$. However, in case monotonicity of the matrix interpretation in Lemma 7 is explicitly required, rule (9) becomes superfluous because $K_{11} \geqslant 1$ and $K^2 = 2I$ imply that all entries of the first row and the first column of K are zero except K_{11} (as K must be non-negative). This means that $(K^2)_{11} = K_{11}^2$, so K_{11} must be equal to $\sqrt{2}$, hence irrational, in order to satisfy $K^2 = 2I$.

Lemma 7 shows that no matrix interpretation over $\mathbb{N}$ or $\mathbb{Q}$ is weakly compatible with the TRS $\mathcal{R}_2$. However, $\mathcal{R}_2$ can be shown terminating by a compatible matrix interpretation over $\mathbb{R}$.

Theorem 9.

1. *The TRS $\mathcal{R}_2$ is terminating. In particular, $\mathcal{R}_2$ is compatible with a monotone matrix interpretation over $\mathbb{R}$.*
2. *There cannot be an (incremental) termination proof of $\mathcal{R}_2$ using only monotone matrix interpretations over $\mathbb{N}$ or $\mathbb{Q}$.*
3. *No matrix interpretation over $\mathbb{N}$ or $\mathbb{Q}$ succeeds on the DP problem $(\_, \mathcal{R}_2)$.*

Proof. The last two claims are immediate consequences of Lemma 7. Finally, the first claim holds by the following monotone 1-dimensional matrix interpretation over $\mathbb{R}$ that is compatible with $\mathcal{R}_2$: $\delta = 1$, $\mathsf{a}_\mathcal{M} = 4$, $\mathsf{b}_\mathcal{M} = 0$, $\mathsf{k}_\mathcal{M}(x) = \sqrt{2}x + 1$ and $+_\mathcal{M}(x, y) = x + y$. □

4 The Dimension Hierarchy

Unlike the previous section, where we have established a hierarchy of matrix interpretations regarding the domain of the matrix entries, the purpose of this section is to examine matrix interpretations with respect to their dimension. That is, we fix $D \in \{\mathbb{N}, \mathbb{Q}_0, \mathbb{R}_0\}$ and consider matrix interpretations over the family of carrier sets $(D^n)_{n \geqslant 1}$. The main result is that the inherent termination hierarchy is infinite with respect to the dimension n, with each level of the hierarchy properly subsuming its predecessor. In other words, $(n+1)$-dimensional matrix interpretations are strictly more powerful for proving termination than n-dimensional matrix interpretations (for any $n \geqslant 1$). We show this by constructing a family of TRSs $(\mathcal{T}_k)_{k \geqslant 2}$ having the property that any of its members $\mathcal{T}_k$ can only be handled with matrix interpretations of dimension at least k. The construction is based on the idea of encoding (i.e., specifying) the degree of the minimal polynomial $\mathrm{m}_A(x)$ of some matrix A occurring in a matrix interpretation in terms of rewrite rules. Thus, if $\mathcal{M}$ is an n-dimensional matrix interpretation such that the degree of the minimal polynomial of some matrix is fixed to a value of k, then the degree of the characteristic polynomial of this matrix must be at least k, i.e., $n \geqslant k$ (since the minimal polynomial divides the characteristic polynomial whose degree is n). In other words, the dimension n of $\mathcal{M}$ must then be at least k. The family of TRSs $(\mathcal{T}_k)_{k \geqslant 2}$ mentioned above is made up as follows. For any natural number $k \geqslant 2$, $\mathcal{T}_k$ denotes the union of the TRS $\mathcal{S}$ of Section 3 and the following rewrite rules:

$$\mathsf{f}^k(x) + \mathsf{d} \to \mathsf{f}^{k-1}(x) + \mathsf{c} \tag{11}$$

$$\mathsf{f}^{k-1}(x) + \mathsf{c} \to \mathsf{f}^k(x) \tag{12}$$

$$\mathsf{h}(\mathsf{f}^{k-2}(\mathsf{h}(x))) \to \mathsf{h}(\mathsf{f}^{k-1}(\mathsf{h}(x))) + x \tag{13}$$

$$\mathsf{h}(\mathsf{f}^{k-1}(\mathsf{h}(x))) \to x \tag{14}$$

The intuition is that if $\mathcal{M}$ is an n-dimensional matrix interpretation that is weakly compatible with all rules of $\mathcal{T}_k$, then the minimal polynomial $\mathrm{m}_F(x)$ of the matrix F associated with the interpretation of the unary function symbol f is forced to be equal to the polynomial $p_k(x) = x^k - x^{k-1}$, a monic polynomial of degree k. This is the purpose of the rules (11) – (14). More precisely, the first two rules ensure that $p_k(x)$ annihilates F, whereas the latter two specify that $p_k(x)$ is the monic polynomial of least degree having this property.

Lemma 10. *Let $\mathcal{M}$ be an n-dimensional matrix interpretation (not necessarily monotone), and let $k \geqslant 2$ be a natural number. Then $\mathcal{T}_k \subseteq \geqslant_{\mathcal{M}}$ implies $n \geqslant k$.*

Proof. Let us assume $\mathcal{T}_k \subseteq \geqslant_{\mathcal{M}}$. Then we also have $\mathcal{S} \subseteq \geqslant_{\mathcal{M}}$ because the TRS $\mathcal{S}$ is contained in $\mathcal{T}_k$. Therefore, the function symbol $+$ must be interpreted by $+_{\mathcal{M}}(\boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{x} + \boldsymbol{y} + \boldsymbol{v}$ according to Lemma 4. Assuming $\mathsf{f}_{\mathcal{M}}(\boldsymbol{x}) = F\boldsymbol{x} + \boldsymbol{f}$ and $\mathsf{h}_{\mathcal{M}}(\boldsymbol{x}) = H\boldsymbol{x} + \boldsymbol{h}$ without loss of generality, the (weak) compatibility constraint associated with rule (11) implies $F^k \geqslant F^{k-1}$. We also have $F^{k-1} \geqslant F^k$ due to rule (12); hence, $F^k = F^{k-1}$. Next we consider the compatibility constraints associated with rule (13) and rule (14). From the former we infer $HF^{k-2}H \geqslant HF^{k-1}H + I$, which implies $F^{k-2} \neq F^{k-1}$, whereas the latter enforces $HF^{k-1}H \geqslant I$, which implies $F^{k-1} \neq 0$. Thus, the $n \times n$ square matrix F must satisfy the following conditions:

$$F^k = F^{k-1} \qquad F^{k-2} \neq F^{k-1} \qquad F^{k-1} \neq 0 \tag{15}$$

These conditions imply that the minimal polynomial of F must be equal to the polynomial $p_k(x) = x^k - x^{k-1}$; i.e., $\mathrm{m}_F(x) = x^k - x^{k-1}$. In order to show this, we first observe that $F^k = F^{k-1}$ means that the polynomial $p_k(x)$ annihilates the matrix F. So $\mathrm{m}_F(x)$ divides $p_k(x)$. Writing $p_k(x) = (x-1)x^{k-1}$ as a product of irreducible factors, we see that if $\mathrm{m}_F(x) \neq p_k(x)$ (i.e., $\mathrm{m}_F(x)$ is a proper divisor of $p_k(x)$ of degree at most $k-1$), then $\mathrm{m}_F(x)$ must divide the polynomial $(x-1)x^{k-2}$ or the polynomial x^{k-1} (depending on whether $(x-1)$ occurs as a factor in $\mathrm{m}_F(x)$ or not). As in both cases the corresponding polynomial annihilates F, we obtain $F^{k-2} = F^{k-1}$ or $F^{k-1} = 0$, contradicting (15). Consequently, $p_k(x)$ must indeed be the minimal polynomial of F, and since it divides the characteristic polynomial of F, the degree of the latter must be greater than or equal to the degree of the former, that is, $n \geqslant k$. □

Remark 11. If one explicitly requires monotonicity of the matrix interpretation $\mathcal{M}$ in Lemma 10, then the condition $F^{k-1} \neq 0$ is automatically satisfied, such that rule (14) becomes superfluous in this case.

Lemma 10 shows that no matrix interpretation of dimension less than k can be weakly compatible with the TRS $\mathcal{T}_k$. However, $\mathcal{T}_k$ can be shown terminating by a compatible matrix interpretation of dimension k.

Theorem 12. *Let $k \geqslant 2$.*

1. *The TRS $\mathcal{T}_k$ is terminating. In particular, $\mathcal{T}_k$ is compatible with a monotone matrix interpretation over $\mathbb{N}$ of dimension k.*
2. *There cannot be an (incremental) termination proof of $\mathcal{T}_k$ using only monotone matrix interpretations of dimension less than k.*

3. *No matrix interpretation of dimension less than k succeeds on the DP problem $(\_, \mathcal{T}_k)$.*

Proof. The last two claims are immediate consequences of Lemma 10. The first claim holds by the following monotone k-dimensional matrix interpretation over $\mathbb{N}$ that is compatible with $\mathcal{T}_k$:

$$\mathsf{a}_{\mathcal{M}} = \mathsf{c}_{\mathcal{M}} = (1, 0, \dots, 0)^{\mathrm{T}} \qquad \mathsf{b}_{\mathcal{M}} = 0 \qquad \mathsf{d}_{\mathcal{M}} = 2\,\mathsf{a}_{\mathcal{M}}$$
$$+_{\mathcal{M}}(\boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{x} + \boldsymbol{y} \qquad \mathsf{f}_{\mathcal{M}}(\boldsymbol{x}) = F\boldsymbol{x} \qquad \mathsf{h}_{\mathcal{M}}(\boldsymbol{x}) = H\boldsymbol{x} + \boldsymbol{h}$$

where $\boldsymbol{h} = (1, \dots, 1)^{\mathrm{T}}$, all rows of H have the shape $(1, 2, 1, \dots, 1)$ and F is zero everywhere except for the entries F_{11} and $F_{i,i+1}$, $i = 1, \dots, k-1$, which are all set to one:

$$F = \begin{pmatrix} 1 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & 0 & 1 & 0 \\ 0 & \cdots & 0 & 0 & 0 & 1 \\ 0 & \cdots & 0 & 0 & 0 & 0 \end{pmatrix} \qquad H = \begin{pmatrix} 1 & 2 & 1 & 1 & \cdots & 1 \\ 1 & 2 & 1 & 1 & \cdots & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & 2 & 1 & 1 & \cdots & 1 \\ 1 & 2 & 1 & 1 & \cdots & 1 \\ 1 & 2 & 1 & 1 & \cdots & 1 \end{pmatrix}$$

□

5 Conclusion

In this paper we have established two hierarchies of matrix interpretations. On the one hand, there is the domain hierarchy stating that matrix interpretations over the real numbers are more powerful with respect to proving termination than matrix interpretations over the rational numbers, which are in turn more powerful than matrix interpretations over the natural numbers (cf. Figure 1). On the other hand, we have established a hierarchy of matrix interpretations with respect to matrix dimension, which was shown to be infinite, with each level properly subsuming its predecessor (cf. Figure 2). Both hierarchies hold in the context of direct termination (using matrix interpretations as a stand-alone termination method) as well as in the setting of the DP framework. Concerning the latter, we remark that the corresponding results in Theorems 6, 9 and 12 do not only hold for standard reduction pairs (as described in Section 2) but also for reduction pairs incorporating the *basic* version of usable rules [3], where the set of usable rules of a DP problem $(\mathcal{P}, \mathcal{S})$ is computed as follows. First, for each defined symbol f occurring in the right-hand side of some rule of $\mathcal{P}$, all f-rules of $\mathcal{S}$ are marked as usable. Then, whenever a rule is usable and its right-hand side contains a defined symbol g, all g-rules of $\mathcal{S}$ become usable as well. In this way, all rules of the TRSs $\mathcal{R}_1$, $\mathcal{R}_2$ and $\mathcal{T}_k$ are usable. It is an easy exercise to make our TRSs also withstand reduction pairs that incorporate usable rules with (implicit) *argument filters* [10] (induced by matrix interpretations).

Our results concerning the domain hierarchy provide a definitive answer to a question raised in [17] whether *rational numbers are somehow unnecessary when*

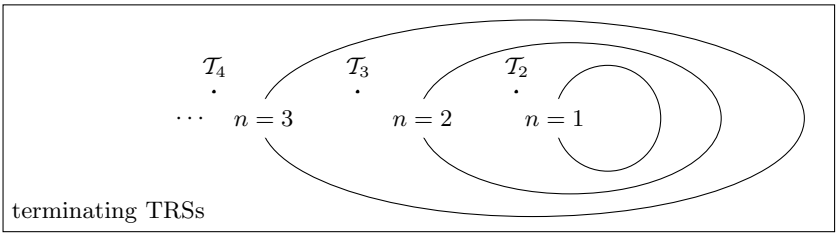


Fig. 2. The dimension hierarchy

dealing with matrix interpretations. The answer is in the negative, so the attempt of [17] to simulate matrix interpretations over $\mathbb{Q}$ with matrix interpretations over $\mathbb{N}$ (of higher dimension) must necessarily remain incomplete.

Moreover, we remark that the results of this paper do not only apply to the standard variant of matrix interpretations of Endrullis et al. [6] (though the technical part of the paper refers to it) but also to the kinds of matrix interpretations recently introduced in [19] (which are based on various different well-founded orders on vectors of natural numbers) and extensions thereof to vectors of non-negative rational and real numbers. On the technical level, this is due to the fact that our main Lemmata 5, 7 and 10 only require weak compatibility (rather than strict) and do not demand monotonicity of the respective matrix interpretations. Also note that the interpretations given in the proofs of Theorems 6, 9 and 12 can be conceived as matrix interpretations over the base order $>^{w}_{\Sigma}$, which relates two vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ if and only if there is a weak decrease in every single component of the vectors and a strict decrease with respect to the sum of the components of $\boldsymbol{x}$ and $\boldsymbol{y}$ (cf. [19]). We expect our results to carry over to the matrix interpretations of [5]. For linear interpretations, this should be possible without further ado, whereas non-linear interpretations conceivably require the addition of new rules enforcing linearity of the interpretations of *some* function symbols (e.g. by using techniques from [18]).

We conclude with a remark on future work and related work. For future work, we mention the extension of the results of this paper to more restrictive classes of TRSs like left-linear ones and SRSs. In this context we also note that the partial result of [8] showing that the dimension hierarchy is infinite applies without further ado since the underlying construction is based on SRSs in contrast to our approach of Section 4.

Acknowledgements. We thank Bertram Felgenhauer for his helpful comments in the early stages of this work.

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. Amitsur, A., Levitzki, J.: Minimal identities for algebras. Proceedings of the American Mathematical Society 1(4), 449–463 (1950)
3. Arts, T., Giesl, J.: Termination of term rewriting using dependency pairs. TCS 236(1-2), 133–178 (2000)

4. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press (1998)
5. Courtieu, P., Gbedo, G., Pons, O.: Improved Matrix Interpretation. In: van Leeuwen, J., Muscholl, A., Peleg, D., Pokorný, J., Rumpe, B. (eds.) SOFSEM 2010. LNCS, vol. 5901, pp. 283–295. Springer, Heidelberg (2010)
6. Endrullis, J., Waldmann, J., Zantema, H.: Matrix interpretations for proving termination of term rewriting. JAR 40(2–3), 195–220 (2008)
7. Gebhardt, A., Hofbauer, D., Waldmann, J.: Matrix evolutions. In: WST 2007, pp. 4–8 (2007)
8. Gebhardt, A., Waldmann, J.: Weighted automata define a hierarchy of terminating string rewriting systems. Acta Cybernetica 19(2), 295–312 (2009)
9. Giesl, J., Thiemann, R., Schneider-Kamp, P.: The Dependency Pair Framework: Combining Techniques for Automated Termination Proofs. In: Baader, F., Voronkov, A. (eds.) LPAR-11 2004. LNCS (LNAI), vol. 3452, pp. 301–331. Springer, Heidelberg (2005)
10. Giesl, J., Thiemann, R., Schneider-Kamp, P., Falke, S.: Mechanizing and improving dependency pairs. JAR 37(3), 155–203 (2006)
11. Hirokawa, N., Middeldorp, A.: Automating the dependency pair method. I&C 199(1-2), 172–199 (2005)
12. Hofbauer, D.: Termination Proofs by Context-Dependent Interpretations. In: Middeldorp, A. (ed.) RTA 2001. LNCS, vol. 2051, pp. 108–121. Springer, Heidelberg (2001)
13. Hofbauer, D., Waldmann, J.: Termination of $\{aa \to bc, bb \to ac, cc \to ab\}$. IPL 98(4), 156–158 (2006)
14. 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)
15. Horn, R., Johnson, C.: Matrix Analysis. Cambridge University Press (1990)
16. 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)
17. Lucas, S.: From Matrix Interpretations over the Rationals to Matrix Interpretations over the Naturals. In: Autexier, S., Calmet, J., Delahaye, D., Ion, P.D.F., Rideau, L., Rioboo, R., Sexton, A.P. (eds.) AISC 2010. LNCS, vol. 6167, pp. 116–131. Springer, Heidelberg (2010)
18. Neurauter, F., Middeldorp, A.: Polynomial interpretations over the reals do not subsume polynomial interpretations over the integers. In: RTA 2010. LIPIcs, vol. 6, pp. 243–258 (2010)
19. Neurauter, F., Middeldorp, A.: Revisiting matrix interpretations for proving termination of term rewriting. In: RTA 2011. LIPIcs, vol. 10, pp. 251–266 (2011)
20. Neurauter, F., Zankl, H., Middeldorp, A.: Revisiting Matrix Interpretations for Polynomial Derivational Complexity of Term Rewriting. In: Fermüller, C.G., Voronkov, A. (eds.) LPAR-17 2010. LNCS, vol. 6397, pp. 550–564. Springer, Heidelberg (2010)
21. Rose, H.E.: Linear Algebra: A Pure Mathematical Approach. Birkhäuser (2002)
22. Terese: Term Rewriting Systems. Cambridge Tracts in Theoretical Computer Science, vol. 55. Cambridge University Press (2003)
23. Thiemann, R.: The DP Framework for Proving Termination of Term Rewriting. PhD thesis, RWTH Aachen, available as Technical Report AIB-2007-17 (2007)
24. Zankl, H., Middeldorp, A.: Satisfiability of Non-linear (Ir)rational Arithmetic. In: Clarke, E.M., Voronkov, A. (eds.) LPAR-16 2010. LNCS, vol. 6355, pp. 481–500. Springer, Heidelberg (2010)

iSat: Structure Visualization for SAT Problems

Ezequiel Orbe, Carlos Areces, and Gabriel Infante-López*

Grupo de Procesamiento de Lenguaje Natural
FaMAF, Universidad Nacional de Córdoba, Argentina
{orbe,areces,gabriel}@famaf.unc.edu.ar

Abstract. We present **iSat**, a Python command line tool to analyze and find structure in propositional satisfiability problems. **iSat** offers an interactive shell to control propositional SAT solvers and generate graph representations of the internal structure of the search space explored by them for visualization, with the final aim of providing a unified environment for propositional solving experimentation. **iSat** was designed to enable simple integration of both new SAT solvers and new visualization graphs and statistics with a minimum of coding overhead.

1 Introduction

iSat[1] (interactive SAT) is a command line tool implemented in Python that helps users to analyze and find structure in propositional satisfiability problems. It can be used, for example, to investigate the behavior of different provers over a given test set. The main service offered by **iSat** is a unified interface for experimentation with different propositional SAT solvers and visualization graphs. Moreover, it can be use to mechanize the repetitive tasks often performed during the development of SAT solvers (e.g., fine tuning heuristics) or the selection of the appropriate configuration options for a given solver working on a particular satisfiability problem. **iSat** computes different visualization graphs (e.g., Variable-Clause , Variable, Interaction, etc.) over the current clause set at different points during the exploration of the search space, and computes related statistics over these graphs (degree mean/max/min/standard deviation, clique number, clustering, number of cliques, etc.). **iSat** was designed to facilitate the integration of new SAT solvers, visualization graphs and statistics with a minimum of coding overhead. **iSat** is distributed under a GPL license and currently supports two SAT solvers out of the box: Minisat [3] and CryptoMinisat [11].

1.1 A Brief Overview on SAT Solving

Propositional satisfiability is the problem of deciding whether there exists a Boolean assignment to variables, such that all clauses in a given propositional formula evaluate to true. Despite its complexity, current SAT solvers (e.g., [3,8,5]) efficiently solve many instances of the SAT problem.

* Consejo Nacional de Investigaciones Científicas y Técnicas.
[1] Available at https://cs.famaf.unc.edu.ar/~ezequiel/software/isat/

N. Bjørner and A. Voronkov (Eds.): LPAR-18, LNCS 7180, pp. 335–342, 2012.

Current SAT solvers can be classified in two broad classes: *incomplete* and *complete* systems. Incomplete solvers perform different kinds of stochastic local search to find a satisfying valuation; if the search is successful, satisfiability is established, but search failure does not imply unsatisfiability. Complete SAT solvers, on the other hand, perform an exhaustive, systematic search, and hence can establish both satisfiability and unsatisfiability. Most of them implement variants of the Davis-Putnam-Logemann-Loveland algorithm (DPLL) [2,1]. Complete SAT solvers can be further classified into conflict-driven and look-ahead. Conflict-driven SAT solvers augment DPLL with conflict analysis, clause learning, non-chronological backtracking and restarts as its principal components. Look-ahead SAT solvers, are also based on DPLL but invest substantial efforts choosing first the branching variable to be used (the different choice options are called decision heuristics) and then the truth value this variable will be assigned (using so called direction heuristics) aiming to achieve the largest reduction of the remaining search space. [4] provides an excellent overview of the area.

The SAT solving community is large and very active, with strong industrial involvement on application areas like planning [6], test pattern generation [7], etc. As a result of this demand, new algorithms and heuristics are being constantly developed, and the available solvers tuned to obtain the best behavior on particular problem domains. But interacting with solvers to gather statistics and explore their behavior when solving particular problems in order to find the best configuration parameters for a given problem class is a burdensome task.

iSat is a command line tool developed in Python that provides an interactive shell for multiple SAT solvers and is capable of producing visualization graphs and statistics, with the final aim of providing a unified environment for SAT solving experimentation. Currently, **iSat** provides access to the Minisat [3] and the CryptoMinisat [11] solvers; produces Variable-Clause graphs, Variable graphs, Literal-Clause graphs, Literal graphs and Interaction graphs that can be exported in `gml` and `dot` format and can be visualized using, for example, Cytoscape[2]; and computes statistics over these graphs like degree mean/max/min/standard deviation, clique number, clustering and number of cliques. Moreover, the architecture of **iSat** has been designed to enable simple integration of new SAT solvers and new analysis tools (i.e., new visualizations and statistics).

2 A Sample Session

We will describe a typical session with **iSat** to illustrate its capabilities. The screen capture of the interaction can be seen in Figure 1.

Consider the case of a researcher who wants to visualize how the structure of a pigeon hole problem instance evolves during search. She suspects that if she can identify some structural properties of the problem, she could develop new

[2] `http://www.cytoscape.org/`

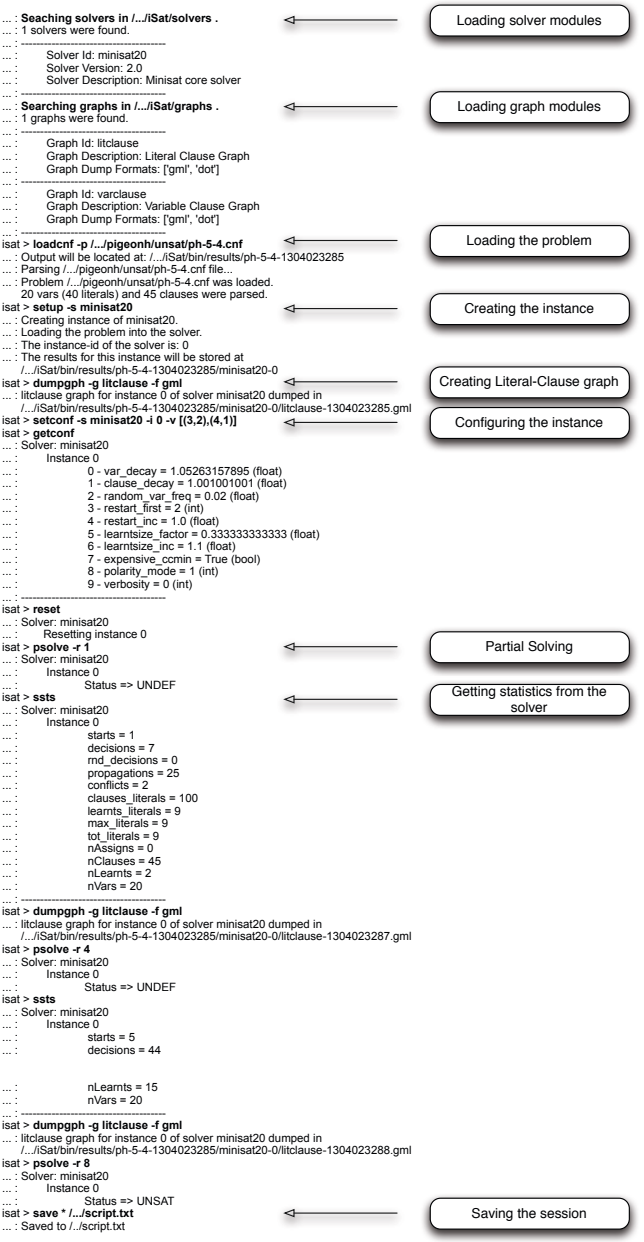


Fig. 1. A typical session

heuristics to improve the search. She uses **iSat** with Minisat as SAT solver and the Literal-Clause graph to visualize the problem structure.

She loads (command **loadcnf**) the problem instance into the tool. **iSat** first parses and loads the problem, and then creates an output folder where result files of the session will be stored. Then she creates an instance of Minisat (command **setup**) and **iSat** loads the problem into the solver and creates an output folder where files related to this instance will be stored. Then she builds the Literal-Clause graph (command **dumpgph**) for the original problem. This will be the baseline against which to compare the different graphs generated during the rest of the session. Before solving the problem, she configures the solver instance (command **setconf**) to make it restart search after 2 conflicts have been found and to keep this bound constant during execution. Next, she checks that the other configuration options of the instance are correctly set (command **getconf**).

Now, she starts exploring how Minisat solves the problem. As she wants to see the structure of the problem at different points, she runs a partial solving (command **psolve**)[3]. Once this ends, she retrieves some statistics from the solver (command **ssts**) to check how the search is performing, and then builds another graph representation from the current state of the problem. She repeats this process until the problem is proved to be unsatisfiable. Finally, she saves the session a script (command **save**) so she can easily re-run her experiments later.

After quitting **iSat**, she will be able to visualize and analyze the generated graphs (see Figure 2) using a suitable graph analysis tool like Cytoscape, looking for structural properties that can be used in the heuristic she is developing.

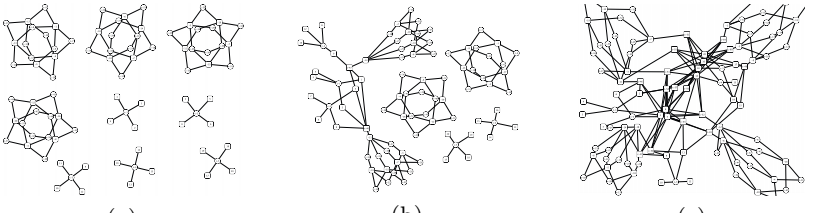


Fig. 2. Evolution of the problem structure: a) Original problem. b) After one restart. c) After five restarts.

3 Services Provided by iSat

Most modern SAT solvers are complex procedures that iteratively modify the internal state of the set of clauses still to be solved together with a partial assignment. The computation starts with the initial state given by the set of clauses in the original formula and an empty assignment. The solver modifies

[3] Currently, **psolve** in Minisat stops search after a specific number of restarts. Other options are possible (e.g., returning the control to **iSat** after a fix number of steps).

C1	`addc`: Adds a list of clauses to a given instance. If no instance is given, they are added to all instances. `simplify`: Simplifies the set of clauses in a given instance. If no instance is given, it simplifies all instances. `solve`: Attempts to solve the problem in a given instance; it lets the solver compute a final state. If no instance is given, it solves all instances. `psolve`: Performs a partial solve in a given instance. If no instance is given, it partially solves all instances using the same number of restarts. `reset`: Resets the internal state. If no instance is given, it resets all instances.
C2	`ssts`: Gather statistics from the given instance. If no instance is specified, it gathers statistics from all instances. `gsts`: Gathers statistics from a specific visualization graph. If no instance is given, it gathers statistics from all instances. `dumpgph`: Generates a file with the visualization graph of the current state of a given instance. If no instance is specified, it generates visualization graphs for every instance. `getconf`: Gets the current configuration options from a given instance. If no instance is specified, it gets the current configuration options from all instances.
C3	`loadcnf`: Loads a problem into memory and creates a folder where the results of the session are stored. `setup`: Creates an instance of a solver and feeds it with the last problem that was loaded using `loadcnf`. For each instance, it creates a subfolder where it outputs information particular to the instance. `setconf`: Configures a SAT solver instance with the given configuration options.
C4	`save`: Saves the current session as a script. `load`: Loads and execute a script file.

Fig. 3. iSat commands

this state step by step adding and removing clauses, and assigning variables. **iSat** is a tool that instruments this computation. It enables users to retrieve the solver state, explore it, represent it as a graph and manually modify it.

iSat can run many instances of the same or different SAT solvers in the same session. In this way, users can compare different intermediate states that might come from different solvers or from the same solver at different stages. Since **iSat** groups instances according to the underlying SAT solver, it is possible to interact with one specific instance, with all instances of one specific solver or with all instances of all solvers. The current version of **iSat** can interact with Minisat and CryptoMinisat, but other SAT solvers can be easily integrated.

Users can also save sessions and reproduce them as scripts. The shell interface (implemented with `cmd2`[4]) enables users to retrieve command history, search the history, and execute Python code and shell commands. **iSat** provides means to inspect the computation state through visualization graphs. Currently, **iSat** can compute Interaction graphs [10], Variable-Clause graphs [9], Variable graphs [9], Literal-Clause graphs, and Literal graphs (the last two are similar to the previous two but using literals as nodes instead of variables). Users can select the type of graph and the instance to export as a file for further analysis.

iSat commands can be grouped in four categories: C1) those that handle and modify the state of the SAT solver, C2) those used to inspect the current state (by means of relevant statistics or visualization graphs), C3) commands to create instances of a problem; and C4) commands to save sessions and to execute saved sessions. Most relevant commands are described in Figure 3.

[4] `http://packages.python.org/cmd2/`

iSat is a powerful tool that offers users relevant information. It let them interact with different solver instances easily and intuitively. Moreover, its architecture enables easy integration of new solvers and visualization graphs.

4 Extending iSat

iSat has been designed to be easily extended to include new SAT solvers and different visualization graphs, with their respective statistics.

iSat uses a Client/Services architecture. The Client layer implements the user interface as an interactive shell. The Services layer provides the interface to different solvers and visualization graphs. Services can be either Solvers or Graphs. A graphical description of the architecture is shown in Figure 4.

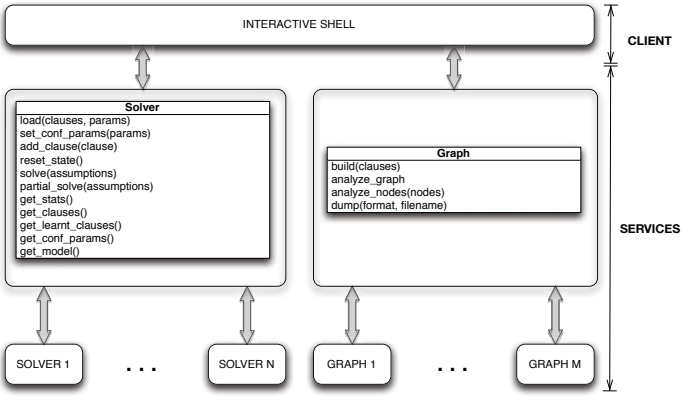


Fig. 4. The architecture of **iSat**

Two components are needed to integrate a new SAT solver into **iSat**. The first wraps the API of the SAT solver into Python and provides Python bindings. Since this wrapper only translates the SAT solver's API into Python, the resulting bindings might not be the ones required by the interactive shell. The second component addresses this issue adapting the wrapper to the specific needs of the interactive shell. Both components are SAT solver dependent and both have to be implemented when a new solver is added to **iSat**.

For SAT solvers developed in C/C++ (this is the case for most current SAT solvers), the first component can be defined with the help of tools like the Simplified Wrapper and Interface Generator (Swig)[5] or the Boost libraries[6] which assist in the definition of bindings for a number of target programming languages, including Python. But even with the help of these tools, defining this component requires careful work and knowledge of the particular solver involved.

[5] http://www.swig.org/
[6] http://www.boost.org/

In particular it is in this component where we should ensure that the resulting Python bindings provide all the necessary basic functionality required by **iSat** (like the ability to stop the run of the solver at a certain point, and retrieve the current state). For example, to build the Python bindings for Minisat, its original C++ API was first extended in the native language to provide the missing functionality required by **iSat**. Besides providing a simplified interface to some of the methods already present in the solver, this extension includes a partial solving method that continues the search till the next restart, and methods that returns the current set of clauses and learnt clauses. The Python bindings for this extension were then obtained using Swig.

The second component, on the other hand, is mostly bookkeeping, and adapts the previous functionality to the concrete function interfaces and datatypes used by **iSat**. In particular, implementing this component boils down to the definition of a subclass of the Python class `Solver` and uses the functionality provided by the wrapper in its implementation. The `Solver` class interface is shown in Figure 4. Methods in this class can be grouped into three categories: configuration methods, information methods, and solving methods. In the configuration category we have methods to load a problem in the solver (`load` and `add_clause`), methods to configure the solver parameters (`set_conf_params`), and methods to reset the internal state of the solver (`reset_state`). In the information category, we have methods that grant access to the internal state of the solver (`get_clauses`, `get_learnt_clauses`), a method to obtain statistics (`get_stats`), a method that returns a model of the current problem, if available (`get_model`), and a method that returns the current configuration parameters (`get_conf_params`). Finally, the solving category includes methods to run the solver, or to run it till a certain predefined condition is met, over the current problem (`solve` and `partial_solve`).

Integrating new visualization graphs into **iSat** follows a similar strategy but is usually simpler as we don't have to deal with the internal complexity of a SAT solver. The common interface is defined by the class `Graph` also shown in Figure 4. This interface includes methods to build the graph (`build`), dump the graph to a file (`dump`), and gather statistics both at graph and at node level (`analyze_graph` and `analyze_node`). There is no restriction on how the graph is implemented internally, or on how the statistics are gathered. Graphs modules already implemented in **iSat** have been developed using the Python package NetworkX[7] to build graphs and to gather associated statistics.

5 Conclusions

iSat is an interactive command line tool that can be used to investigate the internal structure of the search space explored by propositional SAT solvers. It can be used to assist developing new heuristics and to compare different stages of the same or different solvers. **iSat** generates different graph representations of the current problem state, and related statistics. The current version of **iSat** provides the general architecture, integration with the Minisat and CryptoMinisat solvers,

[7] `http://networkx.lanl.gov/`

and implementations for computing Variable-Clause, Variable, Literal-Clause, Literal and Interaction graphs; it provides access to the statistics obtained from Minisat and CryptoMinisat (number of decisions made, propagations, conflicts detected, current number of variables, etc.) together with statistics over the graphs computed (degree mean/max/min/standard deviation, clique number, clustering, number of cliques, etc.). **iSat** was developed with two concrete design goals in mind: to simplify extensibility and to provide an agile interaction with different provers. The outcome is a unified interface for experimentation where new SAT solvers and visualization graphs can be easily integrated.

As far as we known, there exists only one similar tool called DPViz [10]. It offers a tightly integrated environment to visualize different runs of the DPLL procedure and the structure of propositional satisfiability problems. In contrast to our tool, DPViz constructs only one type of graph, and its emphasis is on displaying the internal structure of the problem by using advanced laying out algorithms. Moreover DPViz, does not provide a suitable extension mechanism that allows the user to add new SAT solvers or visualization graphs.

This is the first release of **iSat**. We are currently working on the integration of new SAT solvers, and different graph visualizations. The flexibility offered by the current implementation opens the way to many customizations possibilities. It would be interesting to see in which ways the SAT solving community will make use of **iSat** and contribute to its development.

References

1. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Comm. of the ACM 5(7), 394–397 (1962)
2. Davis, M., Putnam, H.: A computing procedure for quantification theory. J. of the ACM 7(3), 201–215 (1960)
3. Eén, N., Sörensson, N.: An Extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)
4. Heule, M.: SmArT solving: Tools and techniques for satisfiability solvers. PhD thesis, TU Delft (2008)
5. Heule, M., van Maaren, H.: March_dl: Adding adaptive heuristics and a new branching strategy. J. on Sat., Boolean Modeling and Comp. 2, 47–59 (2006)
6. Kautz, H., Selman, B.: Planning as satisfiability. In: Proc. of ECAI 1992. John Wiley and Sons, Inc. (1992)
7. Marques-Silva, J., Sakallah, K.: Robust search algorithms for test pattern generation. In: Proc. of the Fault-Tolerant Computing Symp. IEEE (1997)
8. Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proc. of the 38th Design Automation Conf. (2001)
9. Nudelman, E., Leyton-Brown, K., Hoos, H., Devkar, A., Shoham, Y.: Understanding Random SAT: Beyond the Clauses-to-Variables Ratio. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 438–452. Springer, Heidelberg (2004)
10. Sinz, C., Dieringer, E.-M.: DPvis – A Tool to Visualize the Structure of SAT Instances. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 257–268. Springer, Heidelberg (2005)
11. Soos, M.: CryptoMiniSat — a SAT solver for cryptographic problems (2009), http://www.msoos.org/cryptominisat2

Linear Constraints over Infinite Trees

Martin Hofmann and Dulma Rodriguez

Department of Computer Science, University of Munich
Oettingenstr. 67, D-80538 München, Germany
{martin.hofmann,dulma.rodriguez}@ifi.lmu.de

Abstract. In this paper we consider linear arithmetic constraints over infinite trees whose nodes are labelled with nonnegative real numbers. These constraints arose in the context of resource inference for object-oriented programs but should be of independent interest. It is as yet open whether satisfiability of these constraint systems is at all decidable. For a restricted fragment motivated from the application to resource inference we are however able to provide a heuristic decision procedure based on regular trees. We also observe that the related problem of optimising linear objectives over these infinite trees falls into the area of convex optimisation.

Keywords: Constraints, Infinite trees, Resource analysis.

1 Introduction

In this paper we present a new algorithmic problem related to linear arithmetic over $\mathbb{D} = \mathbb{R}^+ \cup \{\infty\}$. Indeed, it can be seen as a special case of linear arithmetic with infinitely many variables (with some schematic notation so as to make instances of the problem finite objects).

While in general linear arithmetic with infinitely many variables is easily seen to be undecidable (introduce a variable x_{it} for every position i and time t of a computation on a Turing machine) the question of decidability for our special case remains open. We do, however, provide a heuristic solution for an important subcase motivated by practical considerations.

We begin with an informal description of our constraint systems. We have arithmetic variables that take on values in $\mathbb{D} = \mathbb{R}^+ \cup \{\infty\}$ and *tree variables* whose values are infinite trees whose nodes are labelled with elements of $\mathbb{D}$. We fix a finite set $\mathcal{L} = \{\ell_1, \ldots, \ell_n\}$ of labels to address the children of a node, e.g. $\mathcal{L} = \{L, R\}$ for infinite binary trees and $\mathcal{L} = \{tl\}$ for infinite lists.

Such trees can be added, scaled, and compared componentwise; furthermore, we have an operation $\Diamond(.)$ that extracts the root label of a tree, thus if t is a tree expression then $\Diamond(t)$ is an arithmetic expression. Finally, if t is a tree expression and $l \in \mathcal{L}$ then $l(t)$ is a tree expression denoting the l-labelled immediate subtree of t.

Given a system of constraints built from these constructions we can ask for satisfiability and for values of selected arithmetic variables. Asking for values of tree

N. Bjørner and A. Voronkov (Eds.): LPAR-18, LNCS 7180, pp. 343–358, 2012.

variables makes no sense in general as these are infinite objects. We can also ask for the optimum value of some linear combination of the arithmetic variables.

In Figure 1 two infinite trees t_1, t_2 over label set $\mathcal{L} = \{L, R\}$ are defined. It also contains two infinite trees over label set $\mathcal{L} = \{tl\}$ which are effectively infinite lists. Within one and the same constraint system we can only use trees over one and the same label set. These trees satisfy for example: $L(t_1) = R(t_1) = t_1$, $t_1 \sqsubseteq t_2$, $l_2 \sqsubseteq l_1$, $\Diamond(t_1) = 1$, $\Diamond(t_2) = 2$. We also have $t_1 + t_1 = t_2$ and $2t_1 = t_2$ and $tl(l_1) = l_1 + l_2$.

Now, the constraint system $tl(x) \sqsupseteq x \wedge \Diamond(x) \geq 1$ is satisfiable, for example with $x = l_1$ and its optimum value with respect to the objective $c = \Diamond(x)$ to be minimised equals 1. The constraint system $L(x) \sqsupseteq x \wedge R(x) \sqsupseteq x \wedge \Diamond(x) \geq 6$ is satisfiable, for example with $x = 6t_1$.

The constraint system $\Diamond(x) \geq 1 \wedge 2tl(x) = tl(x)$ is also satisfiable, namely by $x = 10^\omega$, but $\Diamond(x) \geq 1 \wedge 2tl(x) = tl(x) \wedge x = tl(x)$, however, is unsatisfiable.

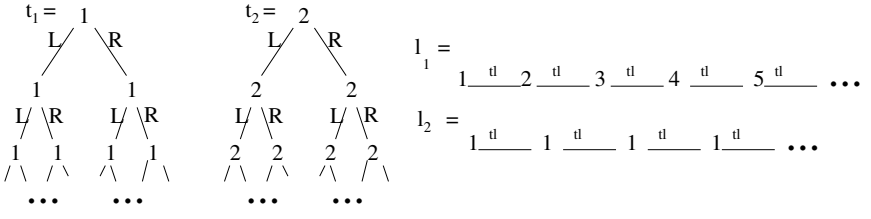


Fig. 1. Some infinite trees

As already mentioned, we currently do not know whether satisfiability of such constraint systems is in general decidable, but the heuristic method we shall present covers all the constraint systems given so far. This is because, the trees witnessing satisfiability were *regular* in the sense that their set of subtrees is finite. So, t_1, t_2, l_2 are regular, but l_1 is not. Accordingly, a constraint system like $\Diamond(x) \geq 1 \wedge tl(x) \sqsupseteq x \wedge tl(y) \sqsupseteq x + y$ is not amenable to our heuristic as it does not admit a regular solution.

In order to decide satisfiability of constraints in general it is tempting to use Büchi tree automata; however, in order to represent our "arithmetic" trees as a tree whose nodes are labelled with letters from a finite alphabet, we would have to represent the numerical annotations using extra branches and then primitive predicates such as equality cannot be recognised by a Büchi tree automaton. Indeed, we conjecture that the algebraic structure of arithmetic trees is not "automatic" in the sense of [BG00].

Nevertheless, we believe that satisfiability of our constraint systems is decidable; in support of this conjecture, we can enlist the fact that the set of solutions to a constraint system is *convex* in the sense that if $\boldsymbol{t}_1$ and $\boldsymbol{t}_2$ are both solutions then so is $(1 - \lambda)\boldsymbol{t}_1 + \lambda\boldsymbol{t}_2$ for $\lambda \in [0, 1]$. Furthermore, constraint systems can be reduced by algebraic manipulations and elimination steps to canonical forms from which solutions can be read off.

We encountered these constraint systems as part of our endeavour of developing an automatic type inference for the object-oriented resource type system presented in [HJ06, HR09]. Indeed, we were able to reduce the type inference problem for that system to satisfiability of arithmetic tree constraint systems. While we do not describe this rather intricate reduction in this paper we try to give a rough indication of it so as to provide further motivation for the potential usefulness of arithmetic tree constraint systems.

The type system presented in *loc.cit.* ascribes refined types to objects in such a way that a concrete object together with its type defines a nonnegative number—the *potential* of that object. The typing rules are formulated in such a way that the potential of all reachable objects under their current typing furnishes an upper bound on the resource usage of any subsequent statement plus the potential of all reachable objects after its execution. In this way, by telescoping, the potential of the initial heap configuration furnishes an upper bound on the total resource usage of a program and this then can be used to read off an input dependent resource bound, e.g. in the form of a linear function of the resource consumption of a program. Through a very coarse lens we can represent the refined type of an object of some class C with fields L and R also of class C as an arithmetic tree over label set $\{L, R\}$. The potential of such an object is then given as the sum of all non-null access paths. If, e.g., the "type" of some object o is t_2 from Fig. 1 and its L, R fields are both *null* then this object carries a potential of 2. If, on the other hand, object o' satisfies $o'.L = o'.R = o$ then the potential of o will equal 6 (and not 4 because ascription of potential is oblivious to aliasing).

In order to infer types one can then introduce an appropriate tree variable wherever a type is required and generated constraints from side conditions of typing rules. Constraints of the form $t \sqsupseteq t'$ arise from subtyping, whereas constraints of the form $t \sqsupseteq t' + t''$ arise from *sharing*, i.e. multiple use of a variable.

We hope, though, that due to their compact and general formulation our arithmetic tree constraint systems will find other applications beyond type inference as well.

We were surprised to find practically no directly related work. One notable exception is [DV07] where constraint satisfaction problems with infinitely many variables are introduced and studied. The difference to our work is twofold: first, the range of individual variables in *loc.cit.* is finite, e.g. Boolean in contrast to $\mathbb{D}$ in our case; secondly, the access policy is much more general and leads to undecidability in general. Interestingly, the near absence of related work has also been noted in *loc.cit.*

2 Infinite Trees

In this section we present infinite trees labelled with nonnegative real numbers. Fix a finite set of labels $\mathcal{L} = \{l_1, \ldots, l_n\}$. The set $\mathsf{T}^{\mathcal{L}}_{\mathbb{D}}$ of infinite trees is given by $\mathsf{T}^{\mathcal{L}}_{\mathbb{D}} = \{t \mid t : \mathcal{L}^* \to \mathbb{D}\}$ where $\mathbb{D} = \mathbb{R}^+ \cup \{\infty\}$ with $0 \in \mathbb{R}^+$. We will refer to elements $w \in \mathcal{L}^*$ as *paths*. We write $|w|$ for the length of w, where $|\epsilon| = 0$ and $|lw| = |w| + 1$. A tree t' is a *sub-tree* of a tree t if there exists $w \in L^*$ so that $t'(p) = t(w\,p)$ for all $p \in L^*$. Further, we say that an infinite tree is *regular*

if it contains a finite number of different sub-trees. The set $\mathsf{T}^{\mathcal{L}}_{\mathbb{D}}$ carries a final coalgebra structure consisting of the function

$$\begin{array}{ll} \langle \Diamond, \mathsf{step} \rangle : & \mathsf{T}^{\mathcal{L}}_{\mathbb{D}} \to \mathbb{D} \times (\mathcal{L} \to \mathsf{T}^{\mathcal{L}}_{\mathbb{D}}) \\ t & \mapsto \langle t(\epsilon), \lambda l\, w\, .\, t(l\, w) \rangle \end{array}$$

where $\mathsf{step}\, l_i$ returns the ith subtree, and $\Diamond$ gives the label of the root node tree [SR10]. We write l_i as a short notation for $\mathsf{step}\, l_i$. For any domain U, every family of functions $lt_i : U \to U$ and $o : U \to \mathbb{D}$ defines a unique function $h : U \to \mathsf{T}^{\mathcal{L}}_{\mathbb{D}}$, such that $\Diamond(h(x)) = o(x)$ and $l_i(h(x)) = h(lt_i(x))$. We define a preorder $\sqsubseteq$ between trees as follows:

Definition 1. *Let $t, t' \in \mathsf{T}^{\mathcal{L}}_{\mathbb{D}}$. We define $t \sqsubseteq t'$ coinductively by $t \sqsubseteq t' \iff \Diamond(t) \leq \Diamond(t')$ and $l_i(t) \sqsubseteq l_i(t')$ for all $l_i \in \mathcal{L}$.*

Alternatively, we can define the same preorder pointwise by:

Definition 2. *Let $t, t' \in \mathsf{T}^{\mathcal{L}}_{\mathbb{D}}$. Then $t \sqsubseteq_{\mathsf{ind}} t' \iff$ for all $w \in L^*\ .\ t(w) \leq t'(w)$.*

Lemma 1. *$t \sqsubseteq_{\mathsf{ind}} t' \iff t \sqsubseteq t'$.*

We define addition of trees $(+ : \mathsf{T}^{\mathcal{L}}_{\mathbb{D}} \times \mathsf{T}^{\mathcal{L}}_{\mathbb{D}} \to \mathsf{T}^{\mathcal{L}}_{\mathbb{D}})$ by: $\Diamond(t + t') = \Diamond(t) + \Diamond(t')$ and $l_i(t + t') = l_i(t) + l_i(t')$ and multiplication of trees with a nonnegative scalar $(\cdot : \mathbb{R}^+ \times \mathsf{T}^{\mathcal{L}}_{\mathbb{D}} \to \mathsf{T}^{\mathcal{L}}_{\mathbb{D}})$ by: $\Diamond(c \cdot t') = c \cdot \Diamond(t')$ and $l_i(c \cdot t') = c \cdot l_i(t')$ for each $l_i \in L$.

Defining a Complete Lattice For the following we recall that the domain $\mathbb{D} = \mathbb{R}^+ \cup \{\infty\}$ is a complete lattice under its usual order by the completeness axiom for $\mathbb{R}$ and because it has top and bottom elements: ∞ and 0. For each $d \in \mathbb{D}$ we define $\widehat{d} \in \mathsf{T}^{\mathcal{L}}_{\mathbb{D}}$ by $\Diamond\left(\widehat{d}\right) = d$ and $l_i(\widehat{d}) = \widehat{d}$ for each $l_i \in \mathcal{L}$. Then, $\widehat{\infty}$ is the top element in $\mathsf{T}^{\mathcal{L}}_{\mathbb{D}}$ and $\widehat{0}$ the bottom. We will show that $(\mathsf{T}^{\mathcal{L}}_{\mathbb{D}}, \sqsubseteq)$ is a complete lattice. For each subset of $\mathsf{T}^{\mathcal{L}}_{\mathbb{D}}$, we define its least upper bound and its greatest lower bound as follows.

- $\bigwedge : \mathcal{P}(\mathsf{T}^{\mathcal{L}}_{\mathbb{D}}) \to \mathsf{T}^{\mathcal{L}}_{\mathbb{D}}$ is totally determined by $\Diamond(\bigwedge T) = \min_{t \in T}(\Diamond(t))$ and $l_i(\bigwedge T) = \bigwedge l_i(T)$.
- $\bigvee : \mathcal{P}(\mathsf{T}^{\mathcal{L}}_{\mathbb{D}}) \to \mathsf{T}^{\mathcal{L}}_{\mathbb{D}}$ is totally determined by: $\Diamond(\bigvee T) = \max_{t \in T}(\Diamond(t))$ and $l_i(\bigvee T) = \bigvee l_i(T)$.

Lemma 2 (Complete Lattice). *Let $t \in \mathsf{T}^{\mathcal{L}}_{\mathbb{D}}$ and $T \subseteq \mathsf{T}^{\mathcal{L}}_{\mathbb{D}}$. Then:*

1. *$\widehat{\infty} \sqsubseteq t$ and $t \sqsubseteq \widehat{0}$.*
2. *$\bigvee T$ is the least upper bound of T and $\bigwedge T$ is the greatest lower bound of T.*
3. *$(\mathsf{T}^{\mathcal{L}}_{\mathbb{D}}, \sqsubseteq)$ is a complete lattice.*

3 Constraints

Next, we consider a system of inequalities among tree expressions and a system of linear arithmetic constraints. Let X be a fixed, countably infinite set of variables

and Λ be a fixed countably infinite set of arithmetic variables where $X \cap \Lambda = \emptyset$. We write TAExp to denote the set of tree expressions that represent a path. We call these expressions *atomic*. The set TExp denotes expressions that represent either a path or a sum of paths. We call expressions in TExp, that are not atomic, *compound*. Moreover, we write AExp to denote linear arithmetic expressions. An arithmetic expression is either a number n, an arithmetic variable λ, an expression representing a potential found at some path $\Diamond(\mathsf{tae})$ or a sum of two expressions $\mathsf{ae}_1 + \mathsf{ae}_2$. We build the sets of *valid* expressions TExp and AExp by the following grammar, where $x \in X$, $n \in \mathbb{D}$, $\lambda \in \Lambda$ and $l \in \mathcal{L}$.

$$\begin{array}{lll} \mathsf{tae} ::= x \mid l(\mathsf{tae}) & & \in \mathsf{TAExp} \\ \mathsf{te} \;\; ::= \mathsf{tae} \mid \mathsf{te} + \mathsf{te} & & \in \mathsf{TExp} \\ \mathsf{ae} \;\; ::= n \mid \lambda \mid \Diamond(\mathsf{tae}) \mid \mathsf{ae} + \mathsf{ae} & & \in \mathsf{AExp} \\ \mathsf{tc} \;\; ::= \mathsf{te} \sqsubseteq \mathsf{te} & & \in \mathsf{TConstr} \\ \mathsf{ac} \;\; ::= \mathsf{ae} \leq \mathsf{ae} & & \in \mathsf{AConstr} \end{array}$$

A system of constraints is a set of valid tree constraints and arithmetic constraints, i.e. a pair $\mathcal{C} = (\mathcal{TC}, \mathcal{AC})$ where $\mathcal{TC}$ and $\mathcal{AC}$ are finite subsets of $\mathsf{TConstr}$ and $\mathsf{AConstr}$ respectively. We write $\mathsf{Vars}(\mathsf{te}) \subseteq X$ for the set of tree variables that occur in the tree expression te and $\mathsf{Vars}(\mathsf{ae}) \subseteq X \cup \Lambda$ for the set of tree and arithmetic variables that appear in the arithmetic expression ae. Moreover, we write $\mathsf{Vars}(\mathcal{C})$ for the set of tree and arithmetic variables that appear in $\mathcal{C}$. Sometimes we write $\mathcal{C}(\boldsymbol{x}, \boldsymbol{\lambda})$ as a short notation for $\mathsf{Vars}(\mathcal{C}) = \boldsymbol{x}, \boldsymbol{\lambda}$.

Meaning of Constraints. Let $\pi = (\pi_t, \pi_a)$ where $\pi_t : X \to \mathsf{T}^{\mathcal{L}}_{\mathbb{D}}$ and $\pi_a : \Lambda \to \mathbb{D}$. The meaning of arithmetic expressions $\pi(\mathsf{ae}) : \mathbb{D}$ is defined in the obvious way, e.g. $\pi(\lambda) = \pi_a(\lambda)$ and $\pi(\Diamond(\mathsf{tae})) = \Diamond(\pi(\mathsf{tae}))$. The meaning of tree expressions $\pi(\mathsf{te}) : \mathsf{T}^{\mathcal{L}}_{\mathbb{D}}$ is defined as one might expect, e.g. $\pi(x) = \pi_t(x)$ and $\pi(l(\mathsf{tae})) = l(\pi(\mathsf{tae}))$. Then, π satisfies a tree constraint $\mathsf{te} \sqsubseteq \mathsf{te}'$ (written $\pi \models \mathsf{te} \sqsubseteq \mathsf{te}'$) if $\pi(\mathsf{te}) \sqsubseteq \pi(\mathsf{te}')$. Similarly, π satisfies an arithmetic expression $\mathsf{ae}_1 \leq \mathsf{ae}_2$ ($\pi \models \mathsf{ae}_1 \leq \mathsf{ae}_2$) if $\pi(\mathsf{ae}_1) \leq \pi(\mathsf{ae}_2)$. Finally, we say π satisfies a system of constraints $\mathcal{C} = (\mathcal{TC}, \mathcal{AC})$ if $\pi \models \mathsf{tc}$ for each $\mathsf{tc} \in \mathcal{TC}$ and $\pi \models \mathsf{ac}$ for each $\mathsf{ac} \in \mathcal{AC}$.

We say that the variable x occurs *only positively* in the system of constraints $\mathcal{C}$ (and write $\mathcal{C}(x^+)$) when it appears only on the right hand side of constraints. Conversely, we say that it appears *only negatively* (and write $\mathcal{C}(x^-)$) when it appears only on the left hand side. Finally, if the variable appears sometimes on the left, sometimes on the right, we write $\mathcal{C}(x^+, x^-)$.

Lemma 3. *Let $\mathcal{C}(x^+)$ and $\mathcal{D}(x^-)$ be systems of constraints and $t, \hat{t} \in \mathsf{T}^{\mathcal{L}}_{\mathbb{D}}$ with $t \sqsubseteq \hat{t}$.*

1. *If $\pi[x \mapsto t] \models \mathcal{C}(x^+)$ then $\pi[x \mapsto \hat{t}] \models \mathcal{C}$.*
2. *If $\pi[x \mapsto \hat{t}] \models \mathcal{D}(x^-)$ then $\pi[x \mapsto t] \models \mathcal{D}$.*

Given a tree expression te and a path w we define $\mathsf{te}_w : \mathsf{AExp}$ inductively by $\mathsf{te}_\epsilon = \Diamond(\mathsf{te})$ and $\mathsf{te}_{lw} = l(\mathsf{te})_w$. The resulting expression may not be valid, but it can easily be transformed into an equivalent valid one with the following transformations

$$l(\mathsf{tae}_1 + \mathsf{tae}_2) = l(\mathsf{tae}_1) + l(\mathsf{tae}_2) \qquad \Diamond(\mathsf{tae}_1 + \mathsf{tae}_2) = \Diamond(\mathsf{tae}_1) + \Diamond(\mathsf{tae}_2) \quad (3.1)$$

For example, $(x + y)_l = \Diamond(l(x + y))$ is not valid but it is equivalent to $\Diamond(l(x)) + \Diamond(l(y))$. Moreover, we define substitution of tree variables with tree expressions in constraints $\mathcal{C}\,[\mathsf{te}/x]$ as usual and ensure that the resulting constraints are valid, again by the transformations (3.1).

3.1 Algorithmic Problems

In this section we discuss algorithmic problems regarding a system of constraints $\mathcal{C}$ whose study would be of interest.

Satisfiability. One important problem, with a direct application to type inference for the RAJA typing system [HJ06], is satisfiability. That is, if we have given a system of constraints, we would like to know whether it is satisfiable. Moreover, we would like to obtain a valuation π that satisfies the constraints. Here we give a slightly weaker definition of the satisfiability problem. We are interested in a *finite* set of arithmetic constraints that is satisfiable iff the system of constraints $\mathcal{C}$ is satisfiable. Since the trees we are studying are infinite, it is not possible to obtain a valuation $\pi_t : X \to \mathsf{T}^{\mathcal{L}}_{\mathbb{D}}$ in general. However, we will see in Section 4 that we can effectively deliver a valuation π_t when all the values in $\mathrm{ran}(\pi_t)$ are regular trees.

Reducing the satisfiability problem to the problem of satisfying a finite set of arithmetic constraints is advantageous because there are effective ways of solving linear arithmetic constraints. Moreover, we remark that the problem of obtaining an *infinite* set of arithmetic constraints equivalent to $\mathcal{C}$ is trivial. If we follow the definition of inequality ($\sqsubseteq_{\mathsf{ind}}$) we notice that a set of inequalities over trees $\mathcal{TC} = \bigcup_i \mathsf{te}_i \sqsubseteq \mathsf{te}'_i$ is satisfiable iff the following set of arithmetic constraints is satisfiable: $\Gamma(\mathcal{TC}) = \{\mathsf{te}_{iw} \leq \mathsf{te}'_{iw} \mid w \in \mathcal{L}^*\}$. In Section 4 we provide an algorithm for solving satisfiability that is sound in all cases and complete for constraints systems of a restricted form.

Example 1. Let $\mathcal{L} = \{l\}$ and $\mathcal{TC} = \{x \sqsubseteq l(x), l(x) + l(x) \sqsubseteq z\}$ and $\mathcal{AC} = \{1 \leq \Diamond(x)\}$. The set $\mathcal{AC}' = \{1 \leq \lambda, \lambda + \lambda \leq \delta\}$ is equivalent to $(\mathcal{TC}, \mathcal{AC})$. This example can be analysed by our algorithm.

Elimination of a Tree Variable. The problem of eliminating a variable x from a system of constraints $\mathcal{C}$ while keeping the satisfiability of the constraints (Fig. 2) is interesting for various reasons. The first one is efficiency. Eliminating variables can reduce significantly the size of a system of constraints. Thus, it is a good idea to eliminate variables first, and then try to solve the resulting system. On the other hand, eliminating variables can help in bringing constraints in a form that is particularly suitable for applying a given algorithm (see Section 4.3). In Section 5 we give an algorithm for variable elimination that, however, does not succeed in eliminating all variables. If we had an algorithm that solved

Satisfiability
Given: A finite system of constraints $\mathcal{C} = (\mathcal{TC}, \mathcal{AC})$.
Wanted: A finite set of linear arithmetic constraints $\mathcal{AC}'$ such that: there is π_a with $\pi_a \models \mathcal{AC}'$ iff there is π_t such that $(\pi_t, \pi_a) \models \mathcal{C}$.

Optimisation
Given: A finite system of constraints $\mathcal{C} = (\mathcal{TC}, \mathcal{AC})$ and a linear objective function f defined on the arithmetic variables.
Wanted: A valuation π_a of the arithmetic variables such that $(\pi_t, \pi_a) \models \mathcal{C}$ for some valuation of the tree variables and whenever $(\pi'_t, \pi'_a) \models \mathcal{C}$ then $f(\pi'_a) \leq f(\pi_a)$.

Elimination of a tree variable
Given: A finite system of constraints $\mathcal{C} = (\mathcal{TC}, \mathcal{AC})$ and a variable $x \in X$.
Wanted: A finite system of constraints $\mathcal{C}'$ with $x \notin \mathsf{Vars}(\mathcal{C}') \subseteq \mathsf{Vars}(\mathcal{C})$ and $\pi \models \mathcal{C}'$ iff $\exists t.\pi[x \mapsto t] \models \mathcal{C}$.

Fig. 2. Algorithmic problems

the elimination problem, the algorithm would solve satisfiability as well, since a finite system of constraints without tree variables is automatically a finite set of arithmetic constraints.

Example 2. Assume we wish to eliminate y from $\mathcal{C} = \{x \sqsubseteq y, y \sqsubseteq l(x)\}, \{1 \leq \Diamond(y)\}$. Then our algorithm would return $\mathcal{C}' = \{x \sqsubseteq l(x)\}, \{1 \leq \Diamond(l(x))\}$ which is equivalent to $\mathcal{C}$. However, our algorithm is not able to eliminate x from $\mathcal{C}'$.

4 Solving a System of Constraints

In this section we present an algorithm for solving a system of constraints $\mathcal{C} = (\mathcal{TC}, \mathcal{AC})$. The linear arithmetic constraints $\mathcal{AC}$ can be solved easily by an LP-Solver. Thus, the challenge is to deal with constraints over trees. Our goal is to reduce the problem of solving these constraints to the problem of solving a finite set of linear arithmetic constraints. We noticed in last section that the canonical set $\Gamma(\mathcal{TC})$ is infinite. But in some particular cases when the constraints admit regular solutions, we can obtain a finite set of arithmetic constraints. Our algorithm seeks solutions to the constraints in the case that the trees must also satisfy some (given) regular structure. When the algorithm is given a regular structure for the tree variables that occur in $\mathcal{TC}$, that we call a *tree schema* Ts, it calculates a finite set of arithmetic constraints. We prove that the algorithm is sound. Clearly, the algorithm is not complete in the general case since not all constraints admit a regular solution. Further, we give in Lemma 4 an upper bound on the size of the resulting set of arithmetic constraints in terms of the sizes of $\mathcal{TC}$ and $\mathsf{Vars}(\mathcal{TC})$.

Tree Constraints in Normal Form. We say that tree expressions are in *normal form* when they are either atomic or a compound expression of the restricted form: $\mathsf{tae} + \mathsf{tae}'$. Moreover, we say that a tree constraint $\mathsf{tc} = \mathsf{te}_1 \sqsubseteq \mathsf{te}_2$

is in normal form if te_1 and te_2 are in normal form and only one of them is compound. Arbitrary tree constraints $\mathsf{tc} \in \mathsf{TConstr}$ can be brought into this form by introducing new variables, for example the tree constraint $x \sqsubseteq y + z + w$ is equivalent to $\{x \sqsubseteq y + v, v = z + w\}$. In the following section we assume that the tree constraints are in normal form. This will simplify our computation of $|\Gamma_{\mathsf{Ts}}(\mathcal{TC})|$ because we will be able to use the fact that $|\mathsf{Vars}(\mathsf{tc})| \leq 3$ for each constraint tc.

4.1 Tree Schema Substitution and $\Delta_{\mathsf{Ts}}(\mathcal{C})$

In the following we define tree schemas: a finite set of tree variables, a finite set of regular trees and a pair of maps, which represent a regular structure for a set of infinite trees.

Definition 3 (Tree Schema). *A* tree schema Ts *consists of*

- *a finite subset* $\mathsf{Ts}.X \subseteq X$.
- *a finite subset* $\mathsf{Ts}.\mathsf{T}^{\mathcal{L}}_{\mathbb{D}} \subseteq \mathsf{T}^{\mathcal{L}}_{\mathbb{D}}$ *closed under* $l(.)$ *for every* $l \in \mathcal{L}$.
- *a total map* $\mathsf{Ts}.\mathsf{next} : \mathcal{L} \times \mathsf{Ts}.X \to \mathsf{Ts}.X \cup \mathsf{Ts}.\mathsf{T}^{\mathcal{L}}_{\mathbb{D}}$.
- *a total injective map* $\mathsf{Ts}.\Diamond : \mathsf{Ts}.X \to \Lambda$.

A valuation $\pi = (\pi_t, \pi_a)$ matches *tree schema* Ts *if the following conditions hold for every* $x \in \mathsf{Ts}.X$*:*

- *if* $\mathsf{Ts}.\Diamond(x) = \lambda \in \Lambda$ *then* $\Diamond(\pi_t(x)) = \pi_a(\lambda)$;
- *if* $\mathsf{Ts}.\mathsf{next}(l, x) = y \in \mathsf{Ts}.X$ *then* $l(\pi_t(x)) = \pi_t(y)$.
- *if* $\mathsf{Ts}.\mathsf{next}(l, x) = t \in \mathsf{T}^{\mathcal{L}}_{\mathbb{D}}$ *then* $l(\pi_t(x)) = t$.

Example 3 (Tree schema). Assume $x_1, x_2 \in X$ and $\lambda_1, \lambda_2 \in \Lambda$ and $\mathcal{L} = \{l\}$. Let Ts be a tree schema defined by $\mathsf{Ts}.X = \{x_1, x_2\}$ and $\mathsf{Ts}.\Diamond(x_i) = \lambda_i$ for $i \in \{1, 2\}$ and $\mathsf{Ts}.\mathsf{next}(l, x_1) = x_2$ and $\mathsf{Ts}.\mathsf{next}(l, x_2) = x_1$. Now define the trees t_1 and t_2 by $\Diamond(t_1) = 1$, $l(t_1) = t_2$ and $\Diamond(t_2) = 2$, $l(t_2) = t_1$. The valuation π given by $\pi_t(x_i) = t_i$ and $\pi_a(\lambda_i) = i$ matches Ts.

The reason why the set $\Gamma(\mathcal{TC})$ is infinite is that it contains expressions te_w for each $w \in L^*$. The main advantage of having a tree schema is that we can eliminate expressions containing labels (like $x_{1ll} = l(l(x_1))$) from a set of constraints. The substitution of such expressions with tree schemas delivers a variable. In this case $l(l(x_1))[\mathsf{Ts}]$ delivers x_1 because $\mathsf{Ts}.\,\mathsf{next}(l, x_1) = x_2$ and $\mathsf{Ts}.\,\mathsf{next}(l, x_2) = x_1$. We define the functions $\mathsf{tae}[\mathsf{Ts}] : X \cup \mathsf{T}^{\mathcal{L}}_{\mathbb{D}}$, $\mathsf{te}[\mathsf{Ts}] : \mathsf{TExp}$ and $\mathsf{ae}[\mathsf{Ts}] : \mathsf{AExp}$ formally in Fig. 3. These functions simplify the given expressions with respect to a particular tree schema so that $\mathcal{TC}[\mathsf{Ts}]$ returns a set of constraints over trees with no (sub)expressions of the form $l(\mathsf{tae})$, while $\mathcal{AC}[\mathsf{Ts}]$ returns a set of arithmetic constraints that contains no tree variables.

In Fig. 3 we also define the set $\Gamma_{\mathsf{Ts}}(\mathcal{TC})$, a set of arithmetic constraints whose satisfiability implies satisfiability of $\mathcal{TC}$. We build the set $\Gamma_{\mathsf{Ts}}(\mathcal{TC})$ as follows: for each constraint $\mathsf{te} \sqsubseteq \mathsf{te}' \in \mathcal{TC}$ and each path $w \in \mathcal{L}^*$, we add the arithmetic constraints $\mathsf{te}_w[\mathsf{Ts}] \leq \mathsf{te}'_w[\mathsf{Ts}]$ to the set. The use of tree schema substitution

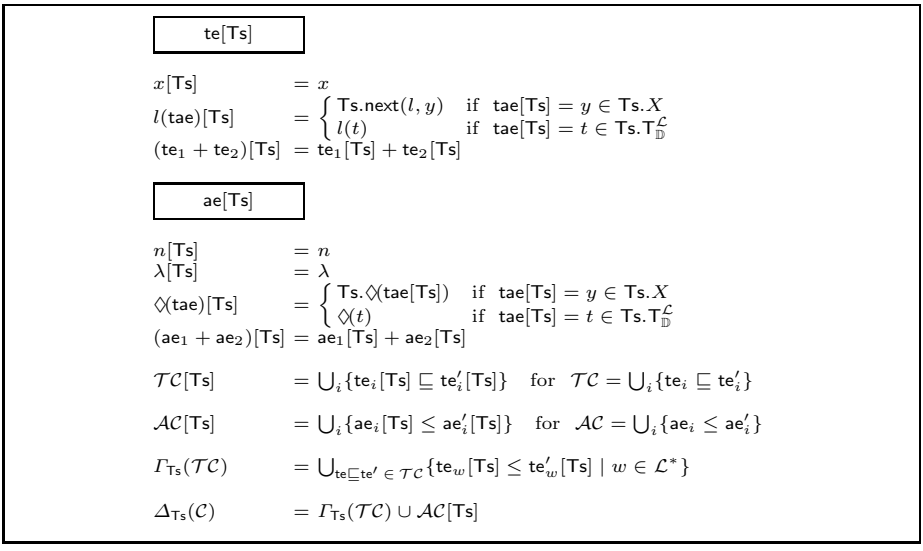


Fig. 3. Tree schema substitution and $\Gamma_{\mathsf{Ts}}(\mathcal{TC})$ and $\Delta_{\mathsf{Ts}}(\mathcal{C})$

ensures that $\Gamma_{\mathsf{Ts}}(\mathcal{TC})$ is finite, in contrast to $\Gamma(\mathcal{TC})$. In the following Lemma we compute an upper bound on the size of $\Gamma_{\mathsf{Ts}}(\mathcal{TC})$ as a function of the sizes of $\mathcal{TC}$ and $\mathsf{Vars}(\mathcal{TC})$.

Lemma 4 (Cardinality of the Set $\Gamma_{\mathsf{Ts}}(\mathcal{TC})$). *Let $\mathcal{TC}$ be a set of constraints and Ts a tree schema with $\mathsf{Ts}.X = \mathsf{Vars}(\mathcal{TC})$. Then $|\Gamma_{\mathsf{Ts}}(\mathcal{TC})| \leq |\mathcal{TC}| \cdot |\mathsf{Ts}.X|^3$.*

The set of arithmetic constraints $\Delta_{\mathsf{Ts}}(\mathcal{C})$, also defined in Fig. 3, is obtained by adding the constraints in $\mathcal{AC}$, after their substitution with the tree schema Ts, to the set $\Gamma_{\mathsf{Ts}}(\mathcal{TC})$. Thus, $\Delta_{\mathsf{Ts}}(\mathcal{C})$ is a finite set of arithmetic constraints without tree variables. We will show below that satisfiability of $\Delta_{\mathsf{Ts}}(\mathcal{C})$ implies satisfiability of $\mathcal{C}$.

Example 4. Let $X, \Lambda, \mathcal{L}$ and Ts be defined as in Example 3. Moreover, let $\mathcal{C} = \{l(x_1) \sqsubseteq x_2, l(x_2) \sqsubseteq x_1\}, \{1 \leq \Diamond(x_1), 2 \leq \Diamond(x_2)\}$. Then $\Gamma_{\mathsf{Ts}}(\mathcal{TC}) = \{\lambda_2 \leq \lambda_2, \lambda_1 \leq \lambda_1\}$ and $\Delta_{\mathsf{Ts}}(\mathcal{C}) = \{\lambda_2 \leq \lambda_2, \lambda_1 \leq \lambda_1, 1 \leq \lambda_1, 2 \leq \lambda_2\}$.

In the following we would like to show the soundness of the algorithm for computing $\Delta_{\mathsf{Ts}}(\mathcal{C})$: if we have a solution for $\Delta_{\mathsf{Ts}}(\mathcal{C})$, we can also find a solution for $\mathcal{C}$. This result is based on the following Lemma, which states that, given a tree schema Ts and a valuation $\pi = (\pi_t, \pi_a)$ that matches Ts, π satisfies $\mathcal{C}$ iff π_a satisfies $\Delta_{\mathsf{Ts}}(\mathcal{TC})$. Moreover we show that all the trees in $\mathrm{ran}(\pi_t)$ are regular.

Lemma 5. *Let Ts be a tree schema with $\mathsf{Ts}.X = \mathsf{Vars}(\mathcal{TC})$ and $\pi = (\pi_t, \pi_a)$ be a valuation that matches Ts. Then:*

1. $\pi_a \models \Gamma_{\mathsf{Ts}}(\mathcal{TC}) \iff \pi \models \mathcal{TC}$.
2. $\pi_a \models \Delta_{\mathsf{Ts}}(\mathcal{C}) \iff \pi \models \mathcal{C}$.
3. *if $t \in \mathrm{ran}(\pi_t)$ then t is regular.*

$$
\begin{array}{l}
\mathsf{subst}(x, \mathsf{Ts}, \pi_a) = t \\
\quad \text{where } \langle\!\langle t \rangle\!\rangle = \pi_a(\mathsf{Ts}.\langle\!\langle x \rangle\!\rangle) \\
\quad \text{and } l(t) \quad = \begin{cases} \mathsf{subst}(y, \mathsf{Ts}, \pi_a) & \text{if } \ \mathsf{Ts.next}(l, x) = y \in \mathsf{Ts}.X \\ t' & \text{if } \ \mathsf{Ts.next}(l, x) = t' \in \mathsf{Ts}.\mathsf{T}_{\mathbb{D}}^{\mathcal{L}} \end{cases} \\
\qquad\qquad \text{for each } l \in \mathcal{L} \\
\\
\mathsf{Ts}[\pi_a] \qquad = \{x \mapsto \mathsf{subst}(x, \mathsf{Ts}, \pi_a) \mid x \in \mathsf{Ts}.X\}
\end{array}
$$

Fig. 4. Extending a tree schema Ts to a valuation $\mathsf{Ts}[\pi_a] : X \to \mathsf{T}_{\mathbb{D}}^{\mathcal{L}}$

Given a tree schema Ts and a valuation π_a that satisfies $\Delta_{\mathsf{Ts}}(\mathcal{C})$, we can build a valuation $\mathsf{Ts}[\pi_a] : \mathsf{Ts}.X \to \mathsf{T}_{\mathbb{D}}^{\mathcal{L}}$, as shown in Fig. 4, such that the valuation $(\mathsf{Ts}[\pi_a], \pi_a)$ matches Ts. Thus, by Lemma 5, $(\mathsf{Ts}[\pi_a], \pi_a)$ satisfies $\mathcal{C}$.

Theorem 1 (Soundness of $\Delta_{\mathsf{Ts}}(\mathcal{C})$). *Let Ts be a tree schema with $\mathsf{Ts}.X = \mathsf{Vars}(\mathcal{TC})$ and $\pi_a : \Lambda \to \mathbb{D}$ be a valuation with $\pi_a \models \Delta_{\mathsf{Ts}}(\mathcal{C})$. Then there exists a valuation $\pi_t : \mathsf{Ts}.X \to \mathsf{T}_{\mathbb{D}}^{\mathcal{L}}$ such that $(\pi_t, \pi_a) \models \mathcal{C}$ and if $t \in \mathrm{ran}(\pi_t)$ then t is regular.*

Lemma 5 also provides a sufficient condition on $\mathcal{C}$ which guarantees that its satisfiability implies satisfiability of $\Delta_{\mathsf{Ts}}(\mathcal{C})$. If it is possible to construct a tree schema such that there is a satisfying valuation for $\mathcal{C}$ that matches it, then $\Delta_{\mathsf{Ts}}(\mathcal{C})$ is satisfiable. Moreover, it follows that $\mathcal{C}$ must admit regular solutions.

Lemma 6 (Condition for Completeness of $\Delta_{\mathsf{Ts}}(\mathcal{C})$). *Let Ts be a tree schema with $\mathsf{Ts}.X = \mathsf{Vars}(\mathcal{TC})$ and let $\pi \models \mathcal{C}$ with π matches Ts. Then $\pi_a \models \Delta_{\mathsf{Ts}}(\mathcal{C})$.*

4.2 Computation of $\Delta_{\mathsf{Ts}}(\mathcal{C})$

In last section we described the set $\Delta_{\mathsf{Ts}}(\mathcal{C})$ and proved that its satisfiability implies the satisfiability of $\mathcal{C}$. The natural question that arises is how to compute $\Delta_{\mathsf{Ts}}(\mathcal{C})$. Computing $\mathcal{AC}[\mathsf{Ts}]$ is simple, the challenge is the computation of $\Gamma_{\mathsf{Ts}}(\mathcal{TC})$. Adding constraints to the set for each path $w \in \mathcal{L}^*$ according to the definition is clearly infeasible since there are infinitely many paths. However, we can calculate the desired set by iteration: we build a set $\Gamma^i_{\mathsf{Ts}}(\mathcal{TC})$ iteratively. In the i-th step of the iteration the set contains exactly the constraints corresponding to the paths w with $|w| \leq i$. We prove that the iteration terminates, i.e. that there is an index j with $\Gamma^j_{\mathsf{Ts}}(\mathcal{TC}) = \Gamma^{j+1}_{\mathsf{Ts}}(\mathcal{TC})$ and that this set contains all the constraints in $\Gamma_{\mathsf{Ts}}(\mathcal{TC})$.

The sets $\Gamma^i_{\mathsf{Ts}}(\mathcal{TC})$ are useful for proving the soundness of the iteration and for understanding how it works. However, actually building the sets in each iteration would be inefficient. Instead, we build a set of tree constraints $\mathcal{TC}^i_{\mathsf{Ts}}$ iteratively (Fig. 5), by adding new constraints in each step, so that the following invariant holds: for all i, $\Gamma^0_{\mathsf{Ts}}(\mathcal{TC}^i_{\mathsf{Ts}}) = \Gamma^i_{\mathsf{Ts}}(\mathcal{TC})$. In the following, we prove the soundness of the iteration: $\Gamma_{\mathsf{Ts}}(\mathcal{TC}) = \Gamma^0_{\mathsf{Ts}}(\mathcal{TC}^\infty_{\mathsf{Ts}})$ that follows directly from the invariant.

$$
\begin{array}{ll}
\Gamma^i_{\mathsf{Ts}}(\mathcal{TC}) & = \bigcup_{\mathsf{te} \sqsubseteq \mathsf{te}' \in \mathcal{TC}} \{\mathsf{te}_w[\mathsf{Ts}] \sqsubseteq \mathsf{te}'_w[\mathsf{Ts}] \mid w \in \mathcal{L}^*, |w| \leq i\} \\
\\
\mathsf{treeConstrs}(\mathcal{TC}) & = \{l(\mathsf{te})[\mathsf{Ts}] \sqsubseteq l(\mathsf{te}')[\mathsf{Ts}] \mid \mathsf{te} \sqsubseteq \mathsf{te}' \in \mathcal{TC}, l \in \mathcal{L}\} \\
\\
\mathcal{TC}^0_{\mathsf{Ts}} & = \mathcal{TC}[\mathsf{Ts}] \\
\mathcal{TC}^{i+1}_{\mathsf{Ts}} & = \mathcal{TC}^i_{\mathsf{Ts}} \cup \mathsf{treeConstrs}(\mathcal{TC}^i_{\mathsf{Ts}}) \\
\mathcal{TC}^{\infty}_{\mathsf{Ts}} & = \bigcup_{i \geq 0} \mathcal{TC}^i_{\mathsf{Ts}}
\end{array}
$$

Fig. 5. $\Gamma^i_{\mathsf{Ts}}(\mathcal{TC})$ and $\mathcal{TC}^i_{\mathsf{Ts}}$

Lemma 7 (Soundness of Iteration). *Let $\mathcal{TC}$ be a set of constraints and* Ts *be a tree schema. Then:*

1. *For all i, $\Gamma^0_{\mathsf{Ts}}(\mathcal{TC}^i) = \Gamma^i_{\mathsf{Ts}}(\mathcal{TC})$.*
2. *$\Gamma_{\mathsf{Ts}}(\mathcal{TC}) = \Gamma^0_{\mathsf{Ts}}(\mathcal{TC}^\infty)$.*

Next, we prove termination of the iteration. The proof consists of two parts. First we notice that, since $\mathcal{TC}^i_{\mathsf{Ts}} \subseteq \mathcal{TC}^{i+1}_{\mathsf{Ts}}$ for all i, if there exists an index n_0 with $\mathsf{treeConstrs}(\mathcal{TC}^{n_0}_{\mathsf{Ts}}) \subseteq \mathcal{TC}^{n_0}_{\mathsf{Ts}}$, then $\mathcal{TC}^{n_0}_{\mathsf{Ts}} = \mathcal{TC}^{n_0+1}_{\mathsf{Ts}}$ and for all $i \geq n_0$ $\mathcal{TC}^i_{\mathsf{Ts}} = \mathcal{TC}^{n_0}_{\mathsf{Ts}}$. The second part of the proof consists in showing that such an index exists for this sequence. It follows from the soundness of the iteration and from the fact that the set $\Gamma_{\mathsf{Ts}}(\mathcal{TC})$ is finite.

Lemma 8 (Termination of Iteration). *Let $\mathcal{TC}$ be a set of constraints and* Ts *be a tree schema. Then:*

1. *If there is n_0 with $\mathsf{treeConstrs}(\mathcal{TC}^{n_0}_{\mathsf{Ts}}) \subseteq \mathcal{TC}^{n_0}_{\mathsf{Ts}}$ then $\forall i \geq n_0 \,.\, \mathcal{TC}^i_{\mathsf{Ts}} = \mathcal{TC}^{n_0}_{\mathsf{Ts}}$.*
2. *There is n_0 with $\mathcal{TC}^{n_0}_{\mathsf{Ts}} = \mathcal{TC}^{n_0+1}_{\mathsf{Ts}}$ and $\mathcal{TC}^{\infty}_{\mathsf{Ts}} = \mathcal{TC}^{n_0}_{\mathsf{Ts}}$.*

Example 5 (Computation of $\Gamma_{\mathsf{Ts}}(\mathcal{TC})$). Let Ts and $\mathcal{L}$ be defined as in Example 3 and $\mathcal{TC}$ be defined as in Example 4. Then, we can build $\Gamma_{\mathsf{Ts}}(\mathcal{TC})$ as follows: $\mathcal{TC}^0_{\mathsf{Ts}} = \{x_2 \sqsubseteq x_2, x_1 \sqsubseteq x_1\}$ and $\mathcal{TC}^1_{\mathsf{Ts}} = \{x_1 \sqsubseteq x_1, x_2 \sqsubseteq x_2\}$. Since $\mathcal{TC}^0_{\mathsf{Ts}} = \mathcal{TC}^1_{\mathsf{Ts}}$, it follows $\mathcal{TC}^\infty_{\mathsf{Ts}} = \mathcal{TC}^1_{\mathsf{Ts}}$. Moreover $\Gamma_{\mathsf{Ts}}(\mathcal{TC}) = \Gamma^0_{\mathsf{Ts}}(\{x_1 \sqsubseteq x_1, x_2 \sqsubseteq x_2\}) = \{\lambda_1 \leq \lambda_1, \lambda_2 \leq \lambda_2\}$.

4.3 Linear Constraint System (LCS)

We proved in a previous section that our algorithm for solving satisfiability for a system of constraints is sound for *any* given tree schema. We also noticed that the algorithm can not be complete, because it imposes a regularity condition on the solutions. In this section we study the following questions: Is there a subset of TConstr, for which the algorithm is complete? Then, how do we find the right tree schemas?

A set of tree constraints $\mathcal{TC}$ induces a graph $G = (V, E)$ whose vertices V are the tree variables occurring in $\mathcal{TC}$. The set of edges E is defined as follows: for each $\mathsf{te} \sqsubseteq \mathsf{te}' \in \mathcal{TC}$, for each $x \in \mathsf{Vars}(\mathsf{te})$ and $y \in \mathsf{Vars}(\mathsf{te}')$, we add (x, y) to E.

Then, we say that a set of tree constraints $\mathcal{TC}$ *contains a loop*, when its corresponding graph G contains a closed path. Moreover, we say that a subset $\mathcal{TC}' \subseteq \mathcal{TC}$ *is a loop*, if the graph G contains a closed path P and for all $\mathsf{tc} \in \mathcal{TC}'$ there exists a variable $x_i \in P$ with $x_i \in \mathsf{Vars}(\mathsf{tc})$ and for all $x_i \in P$ there exists $\mathsf{tc} \in \mathcal{TC}'$ with $x_i \in \mathsf{Vars}(\mathsf{tc})$.

We wish to describe a subset LCS of $\mathsf{TConstr}$ such that for each $\mathcal{C} \in \text{LCS}$ we can effectively construct a tree schema $\mathsf{Ts}_{\mathcal{C}}$ with the following property: there exists a valuation π matching $\mathsf{Ts}_{\mathcal{C}}$ and satisfying $\mathcal{C}$. We will describe that set as a collection of loops of a certain restricted form together with constraints defining relations between the variables that appear in the loops. In particular, the loops should not contain compound expressions. Moreover, every variable x that appear in a loop may appear in arithmetic constraints only in subexpressions $\Diamond(x)$. The following grammar describes the restricted sets of tree constraints $\mathsf{LTConstr}$, $\mathsf{RTConstr}$ and the restricted set of arithmetic constraints $\mathsf{LAConstr}$.

$$\begin{array}{lll} \mathsf{ltc} & ::= l(x) \sqsubseteq x & \in \mathsf{LTConstr} \\ \mathsf{rtc} & ::= x \sqsubseteq l(x) & \in \mathsf{RTConstr} \\ \mathsf{pae} & ::= n \mid \lambda \mid \Diamond(x) \mid \mathsf{pae} + \mathsf{pae} & \in \mathsf{PAExp} \\ \mathsf{lac} & ::= \mathsf{pae} \leq \mathsf{pae} & \in \mathsf{LAConstr} \end{array}$$

Definition 4 (Linear Loop). *Let $\mathcal{TC}' \subseteq \mathcal{TC}$ be a loop.*

1. *We say that $\mathcal{TC}'$ is a* left linear loop *if $\mathcal{TC}' \subseteq \mathsf{LTConstr}$ and for all $x \in \mathcal{TC}'$ holds x occurs only positively in $\mathcal{TC} \setminus \mathcal{TC}'$.*
2. *Further, we say that $\mathcal{TC}'$ is a* right linear loop *if $\mathcal{TC}' \subseteq \mathsf{RTConstr}$ and for all $x \in \mathcal{TC}'$ holds x occurs only negatively in $\mathcal{TC} \setminus \mathcal{TC}'$.*
3. *We say that $\mathcal{TC}'$ is a* linear loop *if it is a left linear or a right linear loop and for all $x \in \mathsf{Vars}(\mathcal{TC}')$ holds if $x \in \mathsf{Vars}(\mathsf{ac})$ for some arithmetic constraint ac then $\mathsf{ac} \in \mathsf{LAConstr}$.*

Definition 5 (Linear Constraint System (LCS)). *We say that $\mathcal{TC}$ is* linear *if $\mathcal{TC} = \mathcal{TC}' \cup (\bigcup_{i=1,\ldots,n} \mathcal{TC}_i)$ where each $\mathcal{TC}_i$ is a linear loop with $\mathsf{Vars}(\mathcal{TC}_i) \cap \mathsf{Vars}(\mathcal{TC}_j) = \emptyset$ for $i \neq j$ and $\mathsf{Vars}(\mathcal{TC}') \subseteq \mathsf{Vars}(\bigcup_{i=1,\ldots,n} \mathcal{TC}_i)$.*

$\mathcal{TC}'$ does not contain loops. This follows by the definition since the variables in $\mathcal{TC}$ must appear either only positively or only negatively in $\mathcal{TC}'$.

Example 6. Let $\mathcal{L} = \{l\}$. Let $\mathcal{C} = \mathcal{TC} = \{l(x_1) \sqsubseteq x_2, l(x_2) \sqsubseteq x_1\}, \{\Diamond(x_1) \leq \Diamond(x_2) + 1, 1 \leq \Diamond(x_2)\}$. Then, $\mathcal{TC}$ is a left linear loop and $\mathcal{C} \in \text{LCS}$.

In Fig. 6 we define a tree schema for a LCS $\mathcal{C}$. We show in the following Lemma that if $\mathcal{C}$ is satisfiable there is a valuation that both satisfies $\mathcal{C}$ and matches the tree schema $\mathsf{Ts}_{\mathcal{C}}$. For the construction of such valuation we use a valuation $\pi_a^{(\pi_t, \mathsf{Ts})} : \Lambda \to \mathbb{D}$ (Fig. 6) that we build on the basis of another valuation π_t and a tree schema Ts.

Lemma 9. *Let $\mathcal{C} = (\mathcal{TC}, \mathcal{AC})$ be a satisfiable LCS. Then there is a valuation π' with $\pi' \models \mathcal{C}$ and π' matches $\mathsf{Ts}_{\mathcal{C}}$.*

$$
\begin{array}{ll}
\mathsf{Ts}_{\mathcal{C}}.X & = \mathsf{Vars}(\mathcal{TC}) \\
\mathsf{Ts}_{\mathcal{C}}.\mathsf{T}^{\mathcal{L}}_{\mathbb{D}} & = \{\widehat{0}, \widehat{\infty}\} \\
\forall\, x_i \in \mathsf{Ts}_{\mathcal{C}}.X : & \\
\quad \mathsf{Ts}_{\mathcal{C}}.\Diamond(x_i) & = \lambda_i \ \text{ where } \lambda_i \notin \mathsf{Vars}(\mathcal{AC}) \\
\quad \forall l_j \in \mathcal{L}\,.\,\mathsf{Ts}_{\mathcal{C}}.\mathsf{next}(l_j, x_i) & = \begin{cases} x_k & \text{if } (l_j(x_i) \sqsubseteq x_k) \in \mathcal{TC} \text{ or } (x_k \sqsubseteq l_j(x_i)) \in \mathcal{TC} \\ \widehat{\infty} & \text{otherwise, if } x_i \text{ occurs in a left linear loop.} \\ \widehat{0} & \text{otherwise, if } x_i \text{ occurs in a right linear loop.} \end{cases}
\end{array}
$$

$$\pi_a^{(\pi_t, \mathsf{Ts})} = \{\delta \mapsto \Diamond(\pi_t(x)) \mid x \in \mathsf{Ts}.X, \mathsf{Ts}.\Diamond(x) = \delta\}$$

Fig. 6. Tree schema for a LCS $\mathcal{C} = (\mathcal{TC}, \mathcal{AC})$

Proof. We have given a valuation $\pi = (\pi_t, \pi_a)$ with $\pi \models \mathcal{C}$. Let $\mathcal{TC} = \mathcal{TC}' \cup (\bigcup_{j=1,\ldots,m} \mathcal{TC}_j(\boldsymbol{x_j}))$. For ease of notation, let us assume that $|\boldsymbol{x_j}| = 1$. Thus, $\boldsymbol{x_j} = x_j$ and let $\pi(x_j) = t_j$. We define $\hat{t}_j$ by:

Case $\mathcal{TC}_j = l(x_j) \sqsubseteq x_j)$. We set $l_k(\hat{t}_j) = \widehat{\infty}$ for $l_k \neq l \in \mathcal{L}$.
Case $\mathcal{TC}_j = x_j \sqsubseteq l(x_j)$. We set $l_k(\hat{t}_j) = \widehat{0}$ for $l_k \neq l \in \mathcal{L}$.

Moreover we set $\Diamond(\hat{t}_j) = \Diamond(t_j)$ and $l(\hat{t}_j) = \hat{t}_j$. Now we set $\hat{\pi}_t = \pi_t[x_j \mapsto \hat{t}_j]$ and $\hat{\pi}_a = \pi_a \cup \pi_a^{(\hat{\pi}_t, \mathsf{Ts}_{\mathcal{C}})}$ and $\hat{\pi} = (\hat{\pi}_t, \hat{\pi}_a)$. We show $\hat{\pi} \models \mathcal{C}$ and $\hat{\pi}$ matches $\mathsf{Ts}_{\mathcal{C}}$: $\hat{\pi}$ matches $\mathsf{Ts}_{\mathcal{C}}$ by construction, $\hat{\pi} \models \mathcal{TC}_j$ follows by construction and $\hat{\pi} \models \mathcal{AC}$ follows by $\pi \models \mathcal{AC}$ and $\Diamond(\hat{t}_j) = \Diamond(t_j)$. Moreover, $\hat{\pi} \models \mathcal{TC}'$ follows by Lemma 3 because if $\mathcal{TC}_j$ is a left linear loop then $\mathcal{TC}'(x_j^+)$ and $t_i \sqsubseteq \hat{t}_i$ and if $\mathcal{TC}_j$ is a right linear loop then $\mathcal{TC}'(x_j^-)$ and $\hat{t}_i \sqsubseteq t_i$.

Theorem 2 (Completeness of $\Delta_{\mathsf{Ts}}(\mathcal{C})$). *Let $\mathcal{C}$ be a satisfiable LCS. Then there is a tree schema* Ts *and a valuation π_a with $\pi_a \models \Delta_{\mathsf{Ts}}(\mathcal{C})$.*

Proof. By Lemma 9 we obtain a valuation $\pi \models \mathcal{C}$ with $\pi = (\pi_t, \pi_a)$ matches $\mathsf{Ts}_{\mathcal{C}}$. Moreover, by Lemma 6, we obtain $\pi_a \models \Delta_{\mathsf{Ts}_{\mathcal{C}}}(\mathcal{C})$.

The restriction to linear constraint systems could seem very strong. However, we will show an algorithm for eliminating variables from constraints while maintaining their satisfiability in the next section. In most cases we are able to eliminate the variables that are not part of a loop with that procedure. Further, we can often bring the loops in the required form by eliminating intermediate variables. For example, the loop $\{l(x) \sqsubseteq y, y \sqsubseteq x\}$ can be transformed into $l(x) \sqsubseteq x$ if we eliminate y. On the other hand, there are systems such as $\{x + x \sqsubseteq l(x)\}, \{1 \leq \Diamond(x)\}$ that can not be transformed into an equivalent linear one. In fact, there is no regular solution for that system.

5 Elimination of Tree Variables

In this section we define an algorithm for eliminating tree variables from a set of tree constraints while keeping their satisfiability.

$$\frac{\mathcal{C}(y^+) \text{ or } \mathcal{C}(y^-)}{\text{erase } y \text{ from } \mathcal{C}} \ (\triangleright \mathsf{Prune}) \qquad \frac{(\bigcup_{i=1..n}\{y \sqsubseteq \mathsf{te}_i\}) \cup \mathcal{D}(y^+), \mathcal{AC}(y^+)}{\bigcup_{i=1..n}(\mathcal{D}, \mathcal{AC})\,[\mathsf{te}_i/y]} \ (\triangleright \mathsf{Elim}^+)$$

$$\frac{(\bigcup_{i=1..n}\{\mathsf{te}_i \sqsubseteq y\}) \cup \mathcal{D}(y^-), \mathcal{AC}(y^-)}{\bigcup_{i=1..n}(\mathcal{D}, \mathcal{AC})\,[\mathsf{te}_i/y]} \ (\triangleright \mathsf{Elim}^-)$$

$$\frac{\mathcal{C}(y^+, y^-) \qquad \mathcal{C}(y^{\mathsf{proj}}) \cap \mathcal{C}(y^{\mathsf{whole}}) = \emptyset \qquad l_i \in \mathcal{L} \text{ and } \boldsymbol{z}, \lambda \text{ new}}{\mathcal{C}(y^{\mathsf{proj}}) \cup \mathsf{unfold}(\mathcal{C}(y^{\mathsf{whole}}))\,[z_i/l_i(y)][\lambda/\Diamond\!\!\!\!-(y)]} \ (\triangleright \mathsf{Elim}^{+/-})$$

Fig. 7. Elimination of tree variables from a set of tree constraints

We say that a variable x occurs *projected* in a set of tree constraints when x appears exclusively in (sub)expressions $l(\mathsf{tae})$, $\Diamond\!\!\!\!-(\mathsf{tae})$. If x appears exclusively as a variable (sub)expression "x" we say that x occurs *as a whole*. We write $\mathcal{C}(x^{\mathsf{proj}})$ for the subset of $\mathcal{C}$ where x occurs projected and we write $\mathcal{C}(x^{\mathsf{whole}})$ for the subset of $\mathcal{C}$ where x appears as a whole. The following function $\mathsf{unfold}(\mathcal{TC})$ unrolls the definition of inequality ($\sqsubseteq$) in the constraints once. The validity of the resulting constraints is ensured by applying the transformations (3.1).

Definition 6 (Unfold Constraints). *Let $\mathcal{TC}$ be a set of tree constraints. We define a function* $\mathsf{unfold}(\mathcal{TC})$ *by unfolding the definition of inequality:*
$\mathsf{unfold}(\mathcal{TC}) = \bigcup_{\mathsf{te} \sqsubseteq \mathsf{te}' \in \mathcal{TC}} \bigcup_{l \in \mathcal{L}} \{l(\mathsf{te}) \sqsubseteq l(\mathsf{te}')\}, \bigcup_{\mathsf{te} \sqsubseteq \mathsf{te}' \in \mathcal{TC}} \{\Diamond\!\!\!\!-(\mathsf{te}) \leq \Diamond\!\!\!\!-(\mathsf{te}')\}$.

In the following we define the algorithm $\mathsf{elim}_{\cdot}(.)$ as a set of inference rules (Fig. 7). If the tree variable y appears only positively or negatively in the constraints then it can be safely removed altogether from the system of constraints ($\triangleright \mathsf{Prune}$). If the variable appears in a constraint such as $\mathsf{tae}_1(y) + \mathsf{tae}_2 \sqsubseteq \mathsf{tae}$, then we return $\mathsf{tae}_2 \sqsubseteq \mathsf{tae}$. Otherwise, when it appears in a constraint $\mathsf{tae}_1 \sqsubseteq \mathsf{tae}_2$ then we remove the whole constraint. Further, if the variable appears in an arithmetic constraint $\Diamond\!\!\!\!-(\mathsf{tae}(y)) + \mathsf{ae}_2 \leq \mathsf{ae}$, we return $\mathsf{ae}_2 \leq \mathsf{ae}$.

Next, we consider the case when the variable has at least one upper or lower bound and appears otherwise only positively ($\triangleright \mathsf{Elim}^+$) or only negatively ($\triangleright \mathsf{Elim}^-$) in the constraints. Then, the elimination takes place by substituting the variable in the constraints with its upper bounds, if the variable occurs only positively, or with its lower bounds, if the variable appears only negatively.

The last and more complicated case is when the variable appears both positively and negatively. Then we calculate $\mathcal{C}(y^{\mathsf{proj}})$ and $\mathcal{C}(y^{\mathsf{whole}})$. If they are disjoint sets, we unfold $\mathcal{C}(y^{\mathsf{whole}})$ and substitute $l_i(y)$ and $\Diamond\!\!\!\!-(y)$ with fresh variables z_i and λ, respectively. If $\mathcal{C}(y^{\mathsf{proj}})$ and $\mathcal{C}(y^{\mathsf{whole}})$ are not disjoint sets, then the variable cannot be eliminated. The reason for this restriction is that, with this rule, we create new variables and we want to eliminate them as well. However, if $\mathcal{C}(y^{\mathsf{proj}})$ and $\mathcal{C}(y^{\mathsf{whole}})$ are not disjoint sets, we would keep eliminating variables and creating new ones without ever coming to an end. Suppose we have the constraint

Table 1. Experimental results. $|\mathsf{Vars}(\mathcal{C})|$ represents the number of tree variables in the constraints before the elimination. $|\mathsf{Ts}.X|$ represents the number of variables remaining after the elimination which is equal to the number of variables in the created tree schema Ts.

Program	**LoC**	$\lvert\mathsf{Vars}(\mathcal{C})\rvert$	$\lvert\mathsf{Ts}.X\rvert$
List Duplication	37	362	4
Doubly-linked Lists	47	568	6
Constant-time List Append	60	674	12
Insertion Sort	66	872	32
List Append	80	1116	8
Merge Sort	127	2818	10
Bank Account	200	3566	10

$l(x) \sqsubseteq x$ and $\mathcal{L} = \{l\}$ and we want to eliminate the variable x. If we applied the rule $(\triangleright\mathsf{Elim}^{+/-})$ we would obtain $l(l(x)) \sqsubseteq l(x)$ after unfolding and $l(z) \sqsubseteq z$ after substituting $l(x)$ with a fresh variable z. If we now tried to eliminate z, we would go through the same procedure again and would never be able to eliminate the variable. The correctness of the elimination procedure follows from the fact that the trees form a complete lattice.

Theorem 3 (Correctness of $\mathsf{elim}()$). *Let $\mathcal{C}(\boldsymbol{x}, y, \boldsymbol{\lambda})$ be a system of constraints. If $\mathsf{elim}_y(\mathcal{C}) = \mathcal{C}'(\boldsymbol{x}, \boldsymbol{\lambda})$ then for all $\pi : \pi \models \mathcal{C}' \iff$ there exists t with $\pi \cup \{y \mapsto t\} \models \mathcal{C}$.*

6 Applications to Resource Analysis

We have implemented the algorithms described in this paper in Ocaml and used them for solving the constraints that arose during our static heap-space analysis of object-oriented programs. Our implementation consists of the following steps:

1. Eliminate variables from the constraints until the only remaining variables are those that appear in a loop.
2. Check if the resulting constraint system is a LCS. In the positive case, construct the tree schema as described in Section 4.3, otherwise construct a tree schema using a heuristic procedure.
3. Compute $\Delta_{\mathsf{Ts}}(\mathcal{C})$ and solve it with an LP-Solver.

Table 1 shows the programs that we could analyse with our tool. For each example, we could solve the constraints and resultantly provide a (linear) upper bound for its heap-space requirements. Notice that the number of tree variables that were generated is proportional to the size of the programs, while the number of variables that remain after the elimination reflects the amount of loops in the constraints and the amount of variables in the loops. There is a demo website where all the examples can be analysed and downloaded [raj].

7 Conclusions

We have presented a system of constraints over infinite trees and we have studied their satisfiability and elimination problems. We have given an algorithm that solves satisfiability for a subcase. Moreover, we have presented a correct algorithm that eliminates a tree variable in most cases. We hope to settle the question of decidability of our tree constraints in general and plan to identify larger tractable subproblems relevant for resource analysis.

Acknowledgements. We acknowledge support by the DFG Graduiertenkolleg 1480 Programm- und Modell-Analyse (PUMA). We also thank Luke Ong for valuable comments.

References

BG00. Blumensath, A., Grädel, E.: Automatic structures. In: LICS, pp. 51–62 (2000)
DV07. Dantchev, S., Valencia, F.D.: On infinite csp's (2007)
HJ06. Hofmann, M.O., Jost, S.: Type-Based Amortised Heap-Space Analysis. In: Sestoft, P. (ed.) ESOP 2006. LNCS, vol. 3924, pp. 22–37. Springer, Heidelberg (2006)
HR09. Hofmann, M., Rodriguez, D.: Efficient Type-Checking for Amortised Heap-Space Analysis. In: Grädel, E., Kahle, R. (eds.) CSL 2009. LNCS, vol. 5771, pp. 317–331. Springer, Heidelberg (2009)
raj. http://raja.tcs.ifi.lmu.de
SR10. Silva, A., Rutten, J.J.M.M.: A coinductive calculus of binary trees. Inf. Comput. 208(5), 578–593 (2010)

E-Matching with Free Variables

Philipp Rümmer

Department of Information Technology, Uppsala University, Sweden

Abstract. E-matching is the most commonly used technique to handle quantifiers in SMT solvers. It works by identifying characteristic subexpressions of quantified formulae, named triggers, which are matched during proof search on ground terms to discover relevant instantiations of the quantified formula. E-matching has proven to be an efficient and practical approach to handle quantifiers, in particular because triggers can be provided by the user to guide proof search; however, as it is heuristic in nature, e-matching alone is typically insufficient to establish a complete proof procedure. In contrast, free variable methods in tableau-like calculi are more robust and give rise to complete procedures, e.g., for first-order logic, but are not comparable to e-matching in terms of scalability. This paper discusses how e-matching can be combined with free variable approaches, leading to calculi that enjoy similar completeness properties as pure free variable procedures, but in which it is still possible for a user to provide domain-specific triggers to improve performance.

1 Introduction

SAT and SMT solvers form the backbone of many of today's verification systems, responsible for discharging verification conditions that encode correctness properties of hardware or software designs. Such verification conditions are often generated in the context of intricate theories, including various kinds of arithmetic, uninterpreted functions and equality, the theory of arrays, or the theory of quantifiers. Despite much research over the past years, efficient and scalable reasoning in the combination of such theories remains challenging: in particular for handling quantifiers, most state-of-the-art SMT solvers have to resort to heuristic techniques like e-matching and triggers [7,8]. E-matching is a popular method due to its simplicity and performance, but offers little completeness guarantees and is sensitive to syntactic manipulations of input formulae.

This paper takes the standpoint that heuristics like e-matching should be considered as *optimisations,* and triggers as *hints,* possibly affecting the performance, but not the completeness of an SMT solver. In other words, the set of formulae that a solver can prove should be independent from chosen triggers. Working towards this goal, the paper presents calculi integrating constraint-based free variable reasoning with e-matching, the individual contributions being (i) a free variable sequent calculus for first-order logic (Sect. 3), with support for e-matching and user-provided triggers to guide instantiation of quantified formulae, partly inspired by the positive unit hyper-resolution calculus [14,15]; (ii) a

N. Bjørner and A. Voronkov (Eds.): LPAR-18, LNCS 7180, pp. 359–374, 2012.

similar calculus for first-order logic modulo linear integer arithmetic (Sect. 5), extending the calculus in [23]; (iii) as a component of both calculi, an approach to capture functions and congruence closure procedures (commonly used in SMT) as uninterpreted predicates (Sect. 4); (iv) a complete implementation of the calculus in (ii), called PRINCESS, and experimental evaluation against SMT solvers competing in the SMT competition 2011 (AUFLIA category) (Sect. 6).

The calculus in (i) is sound and complete for first-order logic, while (ii) is sound and complete for fragments such as Presburger arithmetic, the universal and the existential fragment of first-order logic modulo integers, and the languages accepted by related methods like $\mathcal{ME}(\mathrm{LIA})$ [4] and the complete instantiation method in [9]. The completeness results are significantly stronger than those guaranteed by most SMT solvers, and hold *independently* from the application of e-matching or the choice of triggers in proofs.

1.1 Introductory Example

We start by illustrating e-matching and free variable methods using an example. The first-order theory of non-extensional arrays [16] is often encoded using uninterpreted function symbols *sel* and *sto* by means of the following axioms:

$$\forall x, y, z.\, sel(\underline{sto(x, y, z)}, y) \doteq z \tag{1}$$

$$\forall x, y_1, y_2, z.\, \big(y_1 \doteq y_2 \vee \underline{sel(sto(x, y_1, z), y_2)} \doteq sel(x, y_2)\big) \tag{2}$$

Intuitively, $sel(x, y)$ retrieves the element of array x stored at position y, while $sto(x, y, z)$ denotes the array that is identical to x, except that position y stores value z. In order to prove that some formula holds over the theory of arrays, the underlined expressions can be used as *triggers* that determine when and how the axioms should be instantiated. Generally, triggers consist of a single or multiple expressions (normally sub-expressions in the body of the quantified formula) that contain all quantified variables. For instance, to prove that the implication

$$b \doteq sto(a, 1, 2) \;\rightarrow\; sel(b, 2) \doteq sel(a, 2) \tag{3}$$

holds over the theory of arrays, we can observe that the term $sel(sto(a, 1, 2), 2)$ occurs in the implication, modulo some equational reasoning. This term matches the underlined pattern in (2), and suggests to instantiate (2) to obtain the instance $1 \doteq 2 \vee \underline{sel(sto(a, 1, 2), 2)} \doteq sel(a, 2)$. In fact, (3) follows for this instance of (2), when reasoning in the theories of uninterpreted functions and arithmetic, which allows us to conclude the validity of (3).

Axioms and triggers as shown above are commonly used in SMT solvers, and give rise to efficient decision procedures for ground problems over arrays.[1] However, in the presence of quantifiers, e-matching might be unable to determine the right instantiations, possibly because required instantiations do not exist as ground terms in the formula. For instance, variants of (3) might include:

$$b \doteq sto(a, 1, 2) \;\rightarrow\; \exists x.\, sel(b, x) \doteq sel(a, 2) \tag{4}$$

$$b \doteq sto(a, 1, 2) \;\rightarrow\; \exists x.\, sel(b, x + 1) \doteq sel(a, 2) \tag{5}$$

$$b \doteq sto(a, 1, 2) \;\rightarrow\; \exists x.\, sel(b, x) \doteq sel(a, x) \tag{6}$$

Although the formulae are still valid, the match $sel(sto(a,1,2),2)$ used previously has been eliminated, which makes proof search more intricate. The state-of-the-art e-matching-based SMT solver CVC3 ([3], version 2.4.1) is able to solve (3), but none of (4), (5), (6). A more realistic example, though similar in nature to the formulae shown here, was reported in [12, Sect. 3.3], where a simple modification (Skolemisation) of a small formula prevented Z3 [19] from finding a proof. The goal of the calculus developed in this paper (and of our implementation PRINCESS) is to obtain a system that is more robust against such modifications, by combining e-matching with constraint-based free variable reasoning, while retaining the scalability of SMT solvers.

The general philosophy of free variable methods [11] is to delay the choice of instantiations for quantified formulae with the help of symbolic reasoning. For example, we could instantiate the formula $\exists x.sel(b,x+1) \doteq sel(a,2)$ using a free variable X, resulting in $sel(b,X+1) \doteq sel(a,2)$. Modulo equational reasoning, this creates the term $sel(sto(a,1,2),X+1)$, which can be unified with the trigger in (2) under the constraint $X \doteq 1$. It is then possible to proceed with the proof as described above. After closing the proof, we can conclude that (5) indeed holds, since the derived constraint $X \doteq 1$ is satisfiable: it is possible (retrospectively) to instantiate $\exists x.sel(b,x+1) \doteq sel(a,2)$ with the concrete term $X = 1$.

This example demonstrates that a free variable calculus can be used to compute answers to queries, in a manner similar to constraint logic programming. The system developed in this paper is more general than "ordinary" logic programming, however, since no restrictions on the use of quantifiers are imposed.

2 Background

2.1 Syntax and Semantics of Considered Logics

We assume familiarity with classical first-order logic (FOL, e.g., [11]). Let x range over an infinite set X of variables, c over an infinite set C of constant symbols, p over a set P of uninterpreted predicates with fixed arity, f over a set F of uninterpreted functions with fixed arity, and α over the set $\mathbb{Z}$ of integers. The syntax of the unityped logics in this paper is defined by the following grammar:

$$\begin{aligned} \phi &::= \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi \mid \forall x.\phi \mid \exists x.\phi \mid t \doteq 0 \mid t \leq 0 \mid p(t,\ldots,t) \\ t &::= \alpha \mid c \mid x \mid \alpha t + \cdots + \alpha t \mid f(t,\ldots,t) \end{aligned}$$

The symbol t denotes terms constructed using functions and arithmetic operations. A formula ϕ is called *closed* if all variables in ϕ are bound by quantifiers, and *ground* if it does not contain variables or quantifiers. A location within a formula ϕ is called *positive* if it is underneath an even number of negations $\neg$, otherwise *negative*. Simultaneous substitution of terms $\bar{t} = (t_1,\ldots,t_n)$ for variables $\bar{x} = (x_1,\ldots,x_n)$ in ϕ is denoted by $[\bar{x}/\bar{t}]\phi$; we assume that variable capture

[1] We are grateful to the anonymous referees pointing out that a further trigger (not shown here) is needed in (2) for a complete array procedure.

is avoided by renaming bound variables as necessary. For simplicity, we sometimes write $s \doteq t$ as a shorthand of $1 \cdot s + (-1) \cdot t \doteq 0$. The abbreviation *true* (*false*) stands for $0 \doteq 0$ ($1 \doteq 0$), and implication $\phi \rightarrow \psi$ for $\neg\phi \vee \psi$.

We consider fragments of the syntax shown above, including function-free first-order logic (Sect. 2.3, 3), full first-order logic (Sect. 4), and first-order logic with linear integer arithmetic (Sect. 5). Semantics of any such logic $\mathcal{L}$ is defined by identifying a class $\mathcal{S}_\mathcal{L}$ of structures (U, I), where U is a non-empty *universe,* and I is an *interpretation* that maps predicates $p \in P$ to relations over U, functions $f \in F$ to set-theoretic functions over U, and constants $c \in C$ to values in U. Given (U, I), the evaluation of terms and formulae is defined recursively as is common. A closed formula is called *valid* if it evaluates to *true* for all structures $(U, I) \in \mathcal{S}_\mathcal{L}$, and *satisfiable* if it evaluates to *true* for at least one structure.

2.2 Sequent Calculi with Constraints

Throughout the paper we will work with the *constraint sequent calculus* that is introduced in [23]. The calculus differs from normal Gentzen-style sequent calculi [11] in that every sequent $\Gamma \vdash \Delta$ is annotated with a constraint C (written $\Gamma \vdash \Delta \Downarrow C$) that captures unification conditions derived in a sub-proof. Such unification conditions come into play when free variables (which technically are treated as constants) are used to instantiate quantified formulae. All calculi in this paper are designed such that constraints cannot contain uninterpreted predicates or functions, so that validity/satisfiability of constraints is decidable. Proof procedures and refinements for the calculi are discussed in [23,22].

More formally, if Γ, Δ are finite sets of closed formulae (the *antecedent* and *succedent*) and C is a closed formula, then $\Gamma \vdash \Delta \Downarrow C$ is called a *constrained sequent.* A sequent $\Gamma \vdash \Delta \Downarrow C$ is called *valid* if the formula $(\bigwedge \Gamma \wedge C) \rightarrow \bigvee \Delta$ is valid. A calculus rule is a binary relation between finite sets of sequents (the premises) and single sequents (the conclusion). Proof trees are defined as is common as trees growing upwards in which each node is labelled with a constrained sequent, and in which each node that is not a leaf is related with the nodes directly above through an instance of a calculus rule. A proof is closed if it is finite, and if all leaves are justified by a rule instance without premises.

2.3 The Basic Calculus for Function-Free First-Order Logic

At the core of all calculi introduced in this paper is a calculus for first-order logic with equality, at this point including uninterpreted predicates, but no functions:

$$\phi_{\mathrm{FOL}} \;::=\; \phi \wedge \phi \;\big|\; \phi \vee \phi \;\big|\; \neg\phi \;\big|\; \forall x.\phi \;\big|\; \exists x.\phi \;\big|\; s \doteq s \;\big|\; p(\bar{s}) \qquad\qquad s \;::=\; c \;\big|\; x$$

Since functions and arithmetic are not included in the logic, terms can only be (symbolic) constants or bound variables. Semantics is defined over the class $\mathcal{S}_{\mathrm{FOL}}$ of structures (U, I) with arbitrary non-empty universe U. The constraint calculus PredEq^C for the logic is shown in Fig. 1, with constraints consisting of (possibly negated) equalities, Boolean connectives, and quantifiers. The validity

$$\frac{\Gamma, \phi \vdash \Delta \Downarrow C \quad \Gamma, \psi \vdash \Delta \Downarrow D}{\Gamma, \phi \vee \psi \vdash \Delta \Downarrow C \wedge D}\ \vee\text{L} \qquad \frac{\Gamma, \phi, \psi \vdash \Delta \Downarrow C}{\Gamma, \phi \wedge \psi \vdash \Delta \Downarrow C}\ \wedge\text{L} \qquad \frac{\Gamma \vdash \phi, \Delta \Downarrow C}{\Gamma, \neg\phi \vdash \Delta \Downarrow C}\ \neg\text{L}$$

$$\frac{\Gamma \vdash \phi, \Delta \Downarrow C \quad \Gamma \vdash \psi, \Delta \Downarrow D}{\Gamma \vdash \phi \wedge \psi, \Delta \Downarrow C \wedge D}\ \wedge\text{R} \qquad \frac{\Gamma \vdash \phi, \psi, \Delta \Downarrow C}{\Gamma \vdash \phi \vee \psi, \Delta \Downarrow C}\ \vee\text{R} \qquad \frac{\Gamma, \phi \vdash \Delta \Downarrow C}{\Gamma \vdash \neg\phi, \Delta \Downarrow C}\ \neg\text{R}$$

$$\frac{\Gamma, [x/c]\phi, \forall x.\phi \vdash \Delta \Downarrow [x/c]C}{\Gamma, \forall x.\phi \vdash \Delta \Downarrow \exists x.C}\ \forall\text{L} \qquad \frac{\Gamma, [x/c]\phi \vdash \Delta \Downarrow [x/c]C}{\Gamma, \exists x.\phi \vdash \Delta \Downarrow \forall x.C}\ \exists\text{L} \qquad \frac{\Gamma \vdash \Delta \Downarrow C}{\Gamma, s \doteq t \vdash \Delta \Downarrow s \not\doteq t \vee C}\ =\text{L}$$

$$\frac{\Gamma \vdash [x/c]\phi, \exists x.\phi, \Delta \Downarrow [x/c]C}{\Gamma \vdash \exists x.\phi, \Delta \Downarrow \exists x.C}\ \exists\text{R} \qquad \frac{\Gamma \vdash [x/c]\phi, \Delta \Downarrow [x/c]C}{\Gamma \vdash \forall x.\phi, \Delta \Downarrow \forall x.C}\ \forall\text{R} \qquad \frac{*}{\Gamma \vdash s \doteq t, \Delta \Downarrow s \doteq t}\ =\text{R}$$

$$\frac{*}{\Gamma, p(s_1, \ldots, s_n) \vdash p(t_1, \ldots, t_n), \Delta \Downarrow \bigwedge_i s_i \doteq t_i}\ \textsc{pc} \qquad \frac{[s/t]\Gamma, s \doteq t \vdash [s/t]\Delta \Downarrow C}{\Gamma, s \doteq t \vdash \Delta \Downarrow C}\ =\textsc{red}$$

Fig. 1. The rules of the calculus PredEq^C for first-order predicate logic. In all rules, c is a constant that does not occur in the conclusion: in contrast to the use of Skolem functions and free variables in tableaux, the same kinds of symbols (constants) are used to handle both existential and universal quantifiers. Arbitrary renaming of bound variables is allowed in the constraints when necessary to avoid variable capture.

of formulae of this kind is decidable by quantifier elimination [11]. The calculus is analytic and contains two rules for each formula constructor, as well as a closure rule PC to unify complementary literals. As an optimisation, the rule $=$RED can be used to destructively apply equations; the rule is not necessary to establish completeness, but relevant (together with further refinements) to turn PredEq^C into a practical calculus [23,22].

Lemma 1 (Soundness [22]). *If a sequent $\Gamma \vdash \Delta \Downarrow C$ is provable in PredEq^C, then it is valid (holds in all $\mathcal{S}_{FOL}$-structures).*

In particular, proving a sequent $\Gamma \vdash \Delta \Downarrow C$ with a valid constraint C implies that also the implication $\bigwedge \Gamma \to \bigvee \Delta$ is valid. This gives rise to a constraint-based proof procedure that iteratively constructs proof trees for an input sequent $\Gamma \vdash \Delta \Downarrow ?$ with a yet unknown constraint. The constraints in a proof can be filled in once all proof branches have been closed. In each iteration, the procedure checks whether the constraint generated by the current proof is valid, in which case the procedure can terminate with the result that the input problem has been proven; otherwise, the current proof has to be unfolded further. Strategies for generating proofs (without the need for backtracking, i.e., undoing previous proof steps) are discussed in [23].

Example 2. We show how to prove $\neg\forall x.(\neg p(x) \vee x \doteq c) \vee \neg p(d) \vee p(c)$, in which $p \in P$ is a unary predicate and $c, d \in C$ are constants:

$$\dfrac{\dfrac{\dfrac{\dfrac{*}{p(d) \vdash p(a) \Downarrow d \doteq a}\ \textsc{pc}}{\neg p(a), p(d) \vdash \ldots \Downarrow d \doteq a}\ \neg\text{L} \quad \dfrac{\dfrac{*}{a \doteq c, p(d) \vdash p(c) \Downarrow d \doteq c}\ \textsc{pc}}{a \doteq c, p(d) \vdash p(c) \Downarrow a \not\doteq c \vee d \doteq c}\ =\text{L}}{\ldots, \neg p(a) \vee a \doteq c, p(d) \vdash p(c) \Downarrow d \doteq a \wedge (a \not\doteq c \vee d \doteq c)}\ \vee\text{L}}{\forall x.(\neg p(x) \vee x \doteq c), p(d) \vdash p(c) \Downarrow R}\ \forall\text{L}}{\vdash \neg\forall x.(\neg p(x) \vee x \doteq c) \vee \neg p(d) \vee p(c) \Downarrow R}\ \vee\text{R}*, \neg\text{R}*$$

In order to instantiate the universal quantifier, the fresh constant a is introduced; the constant is quantified existentially in the derived constraints, and therefore can be seen as a "free variable." The constraints on the right-hand side of $\Downarrow$ are practically filled in *after* closing the proof using PC. The validity of the original formula follows from the validity of $R = \exists x.(d \doteq x \wedge (x \not\doteq c \vee d \doteq c))$.

Lemma 3 (Completeness [24]). *Suppose ϕ is closed and valid. Then there is a valid constraint C such that $\vdash \phi \Downarrow C$ is provable in PredEq^C.*

3 Positive Unit Hyper-Resolution

As argued in Sect. 1.1, axioms and quantified formulae (in particular in verification problems) are often manually formulated with a clear, directed application strategy in mind. This makes it possible to systematically instantiate axioms in a manner that more resembles the execution of a functional or logic program than the search for a proof. From a practical point of view, providing support for this style of reasoning (even if it is only applicable to a subset of input problems) is crucial to achieve the scalability needed for applications. We integrate such user-guided reasoning into our calculus with the help of concepts from the *positive unit hyper-resolution* (PUHR) calculus, an approach first used in the SATCHMO theorem prover [14,15]. PUHR will be used in Sect. 4 to simulate the e-matching method common in SMT solvers.

PUHR is a tableau procedure in which clauses are instantiated by matching negative literals on (ground) literals already present on a proof branch. Starting from the calculus PredEq^C defined in the last section, we introduce a similar rule in our hyper-resolution sequent calculus PredEqHR^C, instantiating quantified formulae that are "guarded" by negative literals $\neg p_1(\bar{t_1}), \ldots, \neg p_n(\bar{t_n})$ using symbols from matching literals $p_1(\bar{s_1}), \ldots, p_n(\bar{s_n})$ in the antecedent of a sequent:

$$\frac{\Gamma, \{p_i(\bar{s_i})\}_{i=1}^n,\ \forall \bar{x}.\,(\bigvee_{i=1}^n \neg p_i(\bar{t_i}) \vee \phi),\ \mathit{simp}(\forall \bar{x}.\,(\bigvee_{i=1}^n \bar{s_i} \not\doteq \bar{t_i} \vee \phi)) \vdash \Delta \Downarrow C}{\Gamma, \{p_i(\bar{s_i})\}_{i=1}^n,\ \forall \bar{x}.\,(\bigvee_{i=1}^n \neg p_i(\bar{t_i}) \vee \phi) \vdash \Delta \Downarrow C}\ \forall\text{L-M}$$

Given literals $\{p_i(\bar{s_i})\}_{i=1}^n$ in a sequent, a quantified formula $\forall \bar{x}.\,(\bigvee_{i=1}^n \neg p_i(\bar{t_i}) \vee \phi)$ can be instantiated using the argument terms $\bar{s_i}$ by simultaneously solving the systems $\bar{s_i} \doteq \bar{t_i}$ of equalities. In contrast to the original PUHR [14], we do not require formulae to be range restricted. Note that the formula ϕ might be *false* and disappear, and that the literals $\{p_i(\bar{s_i})\}_{i=1}^n$ are not necessarily distinct. The solving of equalities is formulated using a recursive simplification function *simp*:

$$\begin{array}{rcll}
\mathit{simp}(\forall \bar{x}.(t \not\doteq t \vee \phi)) & = & \mathit{simp}(\forall \bar{x}.\phi) & \\
\mathit{simp}(\forall \bar{x}.(x_i \not\doteq t \vee \phi)) & = & \mathit{simp}(\forall \bar{x}.[x_i/t]\phi) & (x_i \neq t) \\
\mathit{simp}(\forall \bar{x}.(t \not\doteq x_i \vee \phi)) & = & \mathit{simp}(\forall \bar{x}.[x_i/t]\phi) & (x_i \neq t) \\
\mathit{simp}(\forall \bar{x}.(s \not\doteq t \vee \phi)) & = & s \not\doteq t \vee \mathit{simp}(\forall \bar{x}.\phi) & (s, t \notin \bar{x}) \\
\mathit{simp}(\forall \bar{x}.\phi) & = & \forall (\bar{x} \cap \mathit{fv}(\phi)).\,\phi & (\text{otherwise})
\end{array}$$

A rule $\exists$R-M similar to $\forall$L-M is introduced for existentially quantified formulae $\exists \bar{x}.\,(\bigwedge_{i=1}^n p_i(\bar{t_i}) \wedge \phi)$ in the succedent. The soundness of the new rules is

immediate, since the rules only introduce instances of quantified formulae already present in a sequent. After adding $\forall$L-M and $\exists$R-M, it is possible to impose the side-condition that the rule $\forall$L is no longer allowed to be applied to formulae $\forall \bar{x}.\, (\bigvee_{i=1}^{n} \neg p_i(\bar{t_i}) \vee \phi)$; similarly for $\exists$R. In other words, the ordinary rules $\forall$L and $\exists$R may only be applied to formulae that do not start with negative literals. We denote the resulting calculus by PredEqHR^C.

Example 4. We show how the proof from Example 2 can be carried over to PredEqHR^C. To this end, observe that the formula $\forall x.(\neg p(x) \vee x \doteq c)$ in the antecedent is amenable to hyper-resolution, so that it is no longer necessary to introduce the constant a in the proof. Also proof splitting can now be avoided:

$$\dfrac{\dfrac{\dfrac{\dfrac{*}{d \doteq c, p(d) \;\vdash\; p(c) \;\Downarrow\; d \doteq c}\,\text{PC}}{\ldots, d \doteq c, p(d) \;\vdash\; p(c) \;\Downarrow\; d \not\doteq c \vee d \doteq c}\,{=}\text{L}}{\forall x.(\neg p(x) \vee x \doteq c), p(d) \;\vdash\; p(c) \;\Downarrow\; \mathit{true}}\,\forall\text{L-M}}{\vdash\; \neg\forall x.(\neg p(x) \vee x \doteq c) \vee \neg p(d) \vee p(c) \;\Downarrow\; \mathit{true}}\,\vee\text{R}*, \neg\text{R}*$$

$\forall$L-M introduces the formula $\mathit{simp}(\forall x.(d \not\doteq x \vee x \doteq c))$, which can be simplified to $d \doteq c$. A further optimisation is the use of $=$RED to minimise constraints.

Lemma 5 (Completeness [24]). *Suppose ϕ is closed and valid. Then there is a valid constraint C such that $\vdash\ \phi \Downarrow C$ is provable in PredEqHR^C.*

Importantly for efficiency, a variety of refinements [22] restricting applications of $\exists$R-M, $\forall$L-M can be imposed, without losing this completeness result.

4 E-Matching through Relational Encoding

For practical applications, uninterpreted functions are more common and often more important than uninterpreted predicates. Uninterpreted functions and equalities are in SMT solvers normally represented using congruence closure methods [21], which build a *congruence graph* (also called *e-graph*) containing nodes for all function terms present in a problem, with edges representing asserted equalities. More formally, given a finite subterm-closed set T of terms and a finite set E of equalities, the congruence graph is the undirected graph (T, E'), where $E' \supseteq E$ is the smallest transitive and reflexive set of edges satisfying:

> if $f(s_1, \ldots, s_n), f(t_1, \ldots, t_n) \in T$ are nodes with $\{(s_1, t_1), \ldots, (s_n, t_n)\} \subseteq E'$, then also $(f(s_1, \ldots, s_n), f(t_1, \ldots, t_n)) \in E'$.

The relation E' can be constructed by fixed-point iteration, starting from the given equalities E. Congruence graphs can be used to efficiently decide whether an equality $s \doteq t$ follows from the set E of equalities. The congruence graph is also used as the underlying datastructure for e-matching, since matching terms (modulo equations) can efficiently be found using the congruence graph. We discuss in this section how both congruence closure and e-matching can be understood as an encoding of functions as uninterpreted predicates, enabling the integration of e-matching with free variables, without preventing the implementation of congruence closure with the help of efficient native datastructures.

4.1 Relational Encoding of Functions

We consider first-order logic including function symbols, which means that the grammar for terms shown in the beginning of Sect. 2.3 is extended to:

$$s \;::=\; c \;\big|\; x \;\big|\; f(s,\ldots,s)$$

where $f \in F$ ranges over function symbols. For the purpose of the encoding of functions into relations, we assume that a fresh $(n+1)$-ary uninterpreted predicate $f_p \in P$ exists for every n-ary uninterpreted function $f \in F$, representing the graph of f. The relation f_p satisfies two axioms, *functionality* and *totality:*

$$\mathit{Fun}_f = \forall \bar{x}, y_1, y_2.\, \big(\neg f_p(\bar{x}, y_1) \vee \neg f_p(\bar{x}, y_2) \vee y_1 \doteq y_2\big), \quad \mathit{Tot}_f = \forall \bar{x}. \exists y.\, f_p(\bar{x}, y)\,.$$

We can then translate from formulae ϕ over the functional vocabulary F (and relational vocabulary P) to formulae ϕ_{Rel} purely over the relational vocabulary P. This can be done by means of the following rewriting rules:

$$\begin{array}{lll} \exists\text{-}\mathit{enc}: & \psi[f(\bar{t})] & \rightsquigarrow\; \exists x.\, (f_p(\bar{t}, x) \wedge \psi[x]) \\ \forall\text{-}\mathit{enc}: & \psi[f(\bar{t})] & \rightsquigarrow\; \forall x.\, (\neg f_p(\bar{t}, x) \vee \psi[x]) \end{array}$$

Both rules have the side condition that rewritten occurrences of $f(\bar{t})$ must not be in the scope of quantifiers binding variables in the terms $\bar{t}$; furthermore, the variable x must be fresh in $\psi[f(\bar{t})]$. It is possible, however, to apply the rewriting rules to arbitrary sub-formulae of a given formula ϕ; in other words, the predicate and quantifier that encode a function application $f(\bar{t})$ can be placed arbitrarily in the rewritten formula, as long as the function application remains in the scope of the quantifier. Rewriting strategies are discussed later in this section.

Lemma 6. *Suppose ϕ is a closed formula over the vocabulary F, and ϕ_{Rel} is a function-free formula obtained from ϕ by application of the rewriting rules $\exists$-enc and $\forall$-enc. Then ϕ is valid iff $\bigwedge_{f \in F} \big(\mathit{Fun}_f \wedge \mathit{Tot}_f\big) \rightarrow \phi_{Rel}$ is valid.*

Since the calculi PredEq^C and PredEqHR^C are sound and complete for first-order logic without function symbols, we can therefore construct calculi for first-order logic including functions by first encoding functions as relations.

4.2 Ground Reasoning and Congruence Closure

We first concentrate on quantifier-free first-order formulae with functions. In this setting, it is easy to see that the hyper-resolution calculus PredEqHR^C, in combination with the functionality axioms Fun_f for functions f, is able to simulate congruence closure procedures. This is supported by the following strengthened version of Lem. 6, which observes that totality axioms are not necessary when solving essentially ground formulae:

Lemma 7. *Suppose ϕ is a closed formula over the vocabulary F, and ϕ_{Rel} a function-free formula obtained from ϕ by application of the rewriting rules $\exists$-enc and $\forall$-enc that contains $\forall$-quantifiers only in positive positions, and $\exists$-quantifiers only in negative positions. Then ϕ is valid iff $\bigwedge_{f \in F} \mathit{Fun}_f \rightarrow \phi_{Rel}$ is valid.*

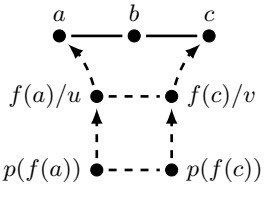
The assumptions of the lemma require that the rewriting rule $\forall$-*enc* is only applied in positive, and $\exists$-*enc* only in negative positions when deriving ϕ_{Rel} from ϕ. As a result, there are only two kinds of quantifiers in the last formula in Lem. 7: quantifiers in ϕ_{Rel} that can be eliminated with the help of the rules $\exists$L and $\forall$R by means of Skolem symbols, and the quantifiers in the axioms Fun_f. Since the latter can be handled using $\forall$L-M, formulae $\bigwedge_{f\in F} Fun_f \to \phi_{Rel}$ can be proven in the calculus PredEqHR^C purely through ground reasoning, without ever resorting to the rules $\forall$L/$\exists$R that introduce existentially quantified constants. This style of reasoning closely corresponds to congruence closure, with literals $f_p(\bar{t}, s)$ in the antecedent of sequents representing equivalence classes of nodes of the congruence graph (T, E'), and instantiation of axioms Fun_f simulating the addition of further edges to the congruence relation E'.

Example 8. We show how $\phi = (p(f(a)) \wedge a \doteq b \wedge b \doteq c \to p(f(c)))$ is proven using the relational encoding. The corresponding formula ϕ_{Rel} is obtained by replacing the function terms $f(a), f(b)$ with fresh quantified variables x, y:

$$\phi_{Rel} = \quad \forall x, y.\big(f_p(a, x) \wedge f_p(c, y) \wedge p(x) \wedge a \doteq b \wedge b \doteq c \;\to\; p(y)\big)$$

We can then construct a proof of $Fun_f \to \phi_{Rel}$ using the rules $=$RED and $\forall$L-M. The central step in the proof is to conclude $u \doteq v$ by instantiating the axiom Fun_f using the symbols occurring in the literals $f_p(c, u)$ and $f_p(c, v)$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{*}{Fun_f, u \doteq v, f_p(c,v), p(v) \;\vdash\; p(v)}\,\text{PC}}{Fun_f, u \doteq v, f_p(c,u), f_p(c,v), p(u), \ldots \;\vdash\; p(v)}\,{=}\text{RED}}{Fun_f, f_p(c,u), f_p(c,v), p(u), a \doteq c, b \doteq c \;\vdash\; p(v)}\,\forall\text{L-M}}{Fun_f, f_p(b,u), f_p(c,v), p(u), a \doteq b, b \doteq c \;\vdash\; p(v)}\,{=}\text{RED}}{Fun_f, f_p(a,u), f_p(c,v), p(u), a \doteq b, b \doteq c \;\vdash\; p(v)}\,{=}\text{RED}}{\vdash\; Fun_f \to \phi_{Rel}}\,\forall\text{R}, \ldots$$



The constraint $\Downarrow$ *true* of each of the sequents has been left out. The proof can also be visualised using the congruence graph shown on the right.

4.3 Relational E-Matching and Free Variables to Handle Quantifiers

E-matching instantiates quantified formulae $\forall x.\phi$ by means of pattern matching: triggers are identified in the matrix ϕ, and are compared with the expressions occurring in the congruence graph to determine relevant instances of the formula. This process can be simulated using the relational function encoding, in combination with the hyper-resolution calculus PredEqHR^C, by deliberately choosing whether literals $f_p(\bar{t}, x)$ in the relational formula $\forall x.\phi_{Rel}$ are introduced with positive or negative sign: since the unit-hyper-resolution rule $\forall$L-M only considers *negative* literals in the matrix ϕ_{Rel} of $\forall x.\phi_{Rel}$ for matching, it is possible to encode triggers by negating the respective literals $f_p(\bar{t}, x)$ (i.e., by using the rewriting rule $\forall$-*enc* to generate such literals), and keeping all other literals positive using the rule $\exists$-*enc*.

Example 9. Consider the quantified formula $\forall x.f(x) \doteq g(x)$. Four possible ways of encoding the formula using relations, corresponding to different strategies when applying the rules $\forall$-*enc* and $\exists$-*enc*, are:

$$\forall x.\exists y,z.\big(f_p(x,y) \wedge g_p(x,z) \wedge y \doteq z\big) \tag{7}$$
$$\forall x,y.\big(\neg f_p(x,y) \vee \exists z.(g_p(x,z) \wedge y \doteq z)\big) \tag{8}$$
$$\forall x,z.\big(\neg g_p(x,z) \vee \exists y.(f_p(x,y) \wedge y \doteq z)\big) \tag{9}$$
$$\forall x,y,z.\big(\neg f_p(x,y) \vee \neg g_p(x,z) \vee y \doteq z\big) \tag{10}$$

Each of the relational formulae corresponds to a particular selection of triggers in $\forall x.f(x) \doteq g(x)$:

- in (7), no triggers have been chosen, with the result that the hyper-resolution rule $\forall$L-M is not applicable. Instantiation of (7) is only possible using the rule $\forall$L, replacing the bound variable x with an existentially quantified constant that can later unified with some term.
- in (8), the term $f(x)$ (corresponding to the negative literal $f_p(x,y)$) has been selected as trigger. In the calculus PredEqHR^C, (8) can only be instantiated using the rule $\forall$L-M, and only in case a literal $f_p(s,t)$ occurs in the antecedent of a sequent, substituting the terms s,t for the variables x,y. This corresponds to e-matching the expression $f(x)$ on a node $f(t)$ of a congruence graph. No free variables are needed to instantiate (8).
- similarly, in (9) the term $g(x)$ is trigger.
- in (10), both $f(x)$ and $g(x)$ have been chosen as a *multi-trigger*, which means that (10) only can be instantiated if literals $f_p(s,t)$ and $g_p(s',t')$ occur in an antecedent. In this case, the instance $s \not\doteq s' \vee t \doteq t'$ will be generated, expressing that the equality $t \doteq t'$ can be assumed if s and s' are unifiable. In terms of e-graphs, the formula would only be instantiated if the e-graph contains nodes $f(s), g(s')$ such that s, s' are in the same equivalence class.

The following proof fragment illustrates how (9) can be instantiated referring to a literal $g_p(a,b)$ in the antecedent, effectively adding $f_p(a,b)$ to the sequent:

$$\dfrac{\dfrac{\dfrac{g_p(a,b),(9),f_p(a,b),b \doteq u \;\vdash\; \Downarrow [y/u]C}{g_p(a,b),(9),f_p(a,u),u \doteq b \;\vdash\; \Downarrow [y/u]C}\,{=}\text{RED}}{g_p(a,b),(9),\exists y.(f_p(a,y) \wedge y \doteq b) \;\vdash\; \Downarrow \forall y.C}\,\exists\text{L},\wedge\text{L}}{g_p(a,b),(9) \;\vdash\; \Downarrow \forall y.C}\,\forall\text{L-M} \tag{11}$$

The way in which a formula ϕ is translated to ϕ_{Rel} determines how quantified sub-formulae are instantiated, in the same way as SMT solvers can be guided by specifying triggers (Alg. 1 shows how the translation can be done systematically, for a given set of triggers). However, it can be observed that the four encodings (7)–(10) are all equivalent w.r.t. provability of theorems: in combination with the axioms Fun_f, Fun_g, Tot_f, Tot_g each of the formulae can simulate each other formula. *The choice of triggers in formulae therefore only influences efficiency, not completeness.* For instance, formula (9) in (11) can be replaced

Algorithm 1. EncodeTrigger: relational encoding of a quantified formula for a specific set of triggers

Input: Formula $\forall \bar{x}.\phi$, set T of trigger terms with variables from $\bar{x}$
Output: Relational formula ϕ_{Rel}

$qvars \leftarrow \{x \mid x \in \bar{x}\}$;
$premises \leftarrow \emptyset$;
while T *contains function terms* **do**
  pick (sub)term $f(\bar{t})$ in T s.t. $\bar{t}$ does not contain functions;
  pick fresh variable y;
  $qvars \leftarrow qvars \cup \{y\}$;
  $premises \leftarrow premises \cup \{f_p(\bar{t}, y)\}$;
  substitute y for $f(\bar{t})$ everywhere in T and ϕ;
end
apply $\exists$-*enc* exhaustively to ϕ;
return $\forall_{x \in qvars}.\big(\bigvee_{p \in premises} \neg p \vee \phi\big)$;

with (8) in the following way (the constraints of the sequents have been left out for sake of brevity):

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{*}{\ldots \vdash x \doteq a}\,{=}\text{R} \quad \dfrac{f_p(x,b), g_p(x,b), g_p(a,b), v \doteq b \vdash}{f_p(x,v), g_p(x,v), g_p(a,b), v \doteq b \vdash}\,{=}\text{RED}}{\ldots, f_p(x,v), g_p(x,v), g_p(a,b), x \not\doteq a \vee v \doteq b \vdash}\,\vee\text{L}, \neg\text{L}}{Fun_g, f_p(x,v), g_p(x,v), v \doteq u, g_p(a,b) \vdash}\,\forall\text{L-M}}{Fun_g, f_p(x,u), g_p(x,v), u \doteq v, g_p(a,b) \vdash}\,{=}\text{RED}}{Fun_g, \ldots, f_p(x,u), \exists z.(g_p(x,z) \wedge u \doteq z), g_p(a,b) \vdash}\,\exists\text{L}, \wedge\text{L}}{Fun_g, Tot_f, f_p(x,u), g_p(a,b), (8) \vdash}\,\forall\text{L-M}}{Fun_g, Tot_f, g_p(a,b), (8) \vdash}\,\forall\text{L}, \exists\text{L}$$

This illustrates that PUHR/e-matching-based reasoning (through $\forall$L-M and $\exists$R-M) can be mixed freely with free variable reasoning (through $\forall$L and $\exists$R). Proofs constructed without applying the rules $\forall$L and $\exists$R closely correspond to the ground reasoning in an SMT solver, while each application of $\forall$L or $\exists$R conceptually introduces a free variable that, at a later point during proof construction, can be unified with other terms, extracting unification conditions in the form of constraints.

5 Extension to Linear Integer Arithmetic

All techniques discussed so far carry over to first-order logic modulo the theory of linear integer arithmetic (FOL(LIA)), via integration into the calculus defined in [23]. The syntax of FOL(LIA) is defined by the grammar in the beginning of Sect. 2.1 and combines first-order logic (with uninterpreted predicates and functions) with arithmetic terms and predicates. Semantics is defined over structures $(\mathbb{Z}, I)$ with the set of integers as universe.

$$\frac{\Gamma, t \doteq 0 \vdash \phi[s + \alpha \cdot t], \Delta \Downarrow C}{\Gamma, t \doteq 0 \vdash \phi[s], \Delta \Downarrow C} =\text{RED-}\mathbb{Z} \qquad \frac{\Gamma, s \leq 0, t \leq 0, \alpha s + \beta t \leq 0 \vdash \Delta \Downarrow C}{\Gamma, s \leq 0, t \leq 0 \vdash \Delta \Downarrow C} \leq\text{L-}\mathbb{Z}$$

$$\frac{\Gamma, p(s_1, \ldots, s_n) \vdash p(t_1, \ldots, t_n), \bigwedge_i s_i - t_i \doteq 0, \Delta \Downarrow C}{\Gamma, p(s_1, \ldots, s_n) \vdash p(t_1, \ldots, t_n), \Delta \Downarrow C} \text{ PU-}\mathbb{Z}$$

$$\frac{*}{\Gamma, \phi_1, \ldots, \phi_n \vdash \psi_1, \ldots, \psi_m, \Delta \Downarrow \neg\phi_1 \vee \cdots \vee \psi_1 \vee \cdots} \text{ CLOSE}$$

Fig. 2. A selection of rules of the calculus PresPred^C; for a complete list see [23]. In =RED-$\mathbb{Z}$, α is a literal; we write $\phi[s]$ in the succedent to denote that s occurs in an arbitrary formula in the sequent, which can in particular also be in the antecedent. In $\leq$L-$\mathbb{Z}$, $\alpha, \beta > 0$ are positive literals. In CLOSE-$\mathbb{Z}$, the formulae $\phi_1, \ldots, \phi_n, \psi_1, \ldots, \psi_m$ do not contain uninterpreted predicates.

As for FOL, we first introduce a calculus for the function-free fragment of FOL(LIA). The integration of functions is then done in the same way as in Sect. 3, 4 with the help of a relational encoding. The calculus PresPred^C for the function-free fragment consists of the rules in Fig. 1, together with a number of rules specific for linear integer arithmetic, a selection of which are shown in Fig. 2 (as a result, the rules =L, =R, =RED, and PC of the first-order calculus can be removed); in the full calculus, also simplification and splitting rules are needed [23]. A more general closure rule CLOSE has to be used than in PredEq^C to support disjunctive constraints. Constraints in PresPred^C are always formulae in Presburger arithmetic (PA), i.e., do not contain uninterpreted predicates.

Lemma 10 (Soundness [23]). *If a sequent $\Gamma \vdash \Delta \Downarrow C$ can be proven in PresPred^C, then it is valid.*

The logic FOL(LIA) subsumes Presburger arithmetic. Since the logic of quantified Presburger arithmetic with predicates is Π^1_1-complete [10], no complete calculi can exist for FOL(LIA); however, it can be shown that the calculi introduced in this section are complete for relevant and non-trivial fragments:

Lemma 11 (Completeness [23]). *Suppose ϕ is a closed formula without functions or constants in one of the following fragments:*

(i) ϕ does not contain uninterpreted predicates (i.e., in Presburger arithmetic);
(ii) ϕ contains universal (exist.) quantifiers only in positive (negative) positions;
(iii) ϕ contains universal (exist.) quantifiers only in negative (positive) positions;
(iv) ϕ is of the form $\forall \bar{x}.(\sigma \to \psi)$, where σ is a formula in Presburger arithmetic (without uninterpreted predicates) that has only finitely many solutions in $\bar{x}$, and ψ contains universal (existential) quantifiers only in negative (positive) positions (i.e., a formula accepted by the $\mathcal{ME}$(LIA) calculus [4]).

Then there is a valid constraint C such that $\vdash \phi \Downarrow C$ is provable in PresPred^C.

Practically, it can be observed that PresPred^C can often also be applied successfully to formulae outside of those fragments.

5.1 Hyper-Resolution and E-Matching for FOL(LIA)

The unit hyper-resolution rule ∀L-M (and similarly the rule ∃R-M) defined in Sect. 3 can be integrated in the calculus PresPred^C in the same way as in the earlier first-order calculi, in order to instantiate formulae $\forall \bar{x}.\, (\bigvee_{i=1}^{n} \neg p_i(\bar{t_i}) \vee \phi)$ by matching. In this context, the simplification function *simp* can be (but does not have to be) replaced with a function tailored to integer arithmetic, i.e., a function that is able to solve the system $\bigvee_{i=1}^{n} \bar{s_i} \neq \bar{t_i}$ modulo integer arithmetic.

The calculus PresPredHR^C is derived from PresPred^C by adding the rules ∀L-M and ∃R-M, and by imposing the side condition that the rule ∀L is no longer applied to formulae of the shape $\forall \bar{x}.\, (\bigvee_{i=1}^{n} \neg p_i(\bar{t_i}) \vee \phi)$; similarly for the rule ∃R. As before, the soundness of the rules ∀L-M and ∃R-M is immediate. We can also observe that PresPredHR^C is relatively complete, in the sense that formulae that are provable in PresPred^C can also be proven using PresPredHR^C:

Lemma 12. *Suppose $\Gamma \vdash \Delta \Downarrow C$ is provable in PresPred^C, where C is valid. Then there is a valid constraint C' so that PresPredHR^C can prove $\Gamma \vdash \Delta \Downarrow C'$.*

Encoding of functions. The relational encoding of functions from Sect. 4 can be used to obtain a calculus for the full logic FOL(LIA) with functions. Although there are no complete calculi for the full logic, we can observe that PresPred^C (and therefore, by Lem. 12, PresPredHR^C) can handle at least all formulae that can be proven by considering a finite set of ground instances:

Lemma 13. *Suppose $\exists \bar{x}.\phi$ is a closed formula in FOL(LIA), with functions taken from a finite set F, such that ϕ is quantifier-free. If there is a valid disjunction $\bigvee_{i=1}^{n} [\bar{x}/\bar{t_i}]\phi$ of ground instances of $\exists \bar{x}.\phi$, then there is a valid constraint C such that $\{Fun_f, Tot_f\}_{f \in F} \vdash (\exists \bar{x}.\phi)_{Rel} \Downarrow C$ is provable in PresPred^C.*

The lemma directly generalises to disjunctions of existentially quantified formulae, which in particular entails that PresPred^C is complete for the class of *essentially uninterpreted formulae* F (modulo linear integer arithmetic) with finite ground instantiation $F*$ defined in [9], and thus also for the array property fragment [6] (PresPred^C cannot easily be turned into a decision procedure, however, since it would be unclear how to ensure termination on invalid problems).

6 Experiments and Related Work

We have implemented the described calculus PresPredHR^C for FOL(LIA) in the theorem prover Princess,[2] and are in the process of adding further optimisations. Princess uses the relational encoding from Sect. 4 to represent functions, and heuristics similar to the ones in Simplify [7] to automatically identify triggers in quantified formulae; redundancy criteria [22] and theory propagation help to reduce the number of instances generated from quantified formulae. Princess is able to handle all of the examples discussed in Sect. 1.1.

[2] http://www.philipp.ruemmer.org/princess.shtml

	AUFLIA+p (193)	**AUFLIA-p** (193)
Z3	191	191
Princess	**145**	**137**
CVC3	132	128

Fig. 3. Number of solved benchmarks, out of 2×193 unsatisfiable (scrambled) AUFLIA benchmarks selected in the SMT competition 2011. Experiments with Princess were done on an Intel Core i5 2-core machine with 3.2GHz, with a timeout of 1200s, heap-space limited to 4Gb. The benchmarks in AUFLIA+p contain hand-written triggers for most of the quantified formulae, while all triggers have been removed in AUFLIA-p. The corresponding figures for Z3 and CVC3 are the results obtained during the SMT competition 2011 (`http://www.smtexec.org/exec/?jobs=856`).

To evaluate the overhead of the relational function encoding, we compared the performance of Princess with the SMT solvers CVC3 [3] and Z3 [19], using benchmarks selected in the SMT competition 2011. Since our work concentrates on proof construction, we only considered unsatisfiable benchmarks, removing 13 satisfiable AUFLIA problems in each category. The results show that Princess, while currently not being able to compete with the fastest SMT solver Z3, performs better than the (state-of-the-art) e-matching-based CVC3. This is promising, since Princess does not (yet) use SMT techniques like lemma learning, which are important for large or propositionally complex problems. Princess can solve most benchmarks using e-matching alone, but uses free variables in 17 of the (solved) benchmarks, typically in smaller (but harder) instances.

Related Work. E-matching is today used in most SMT solvers, based on techniques that go back to the Simplify prover [7] and The Stanford Pascal Verifier [20]; since then, various refinements of the e-matching approach have been published, for instance [8,18]. To the best of our knowledge, e-matching has not previously been combined with free variable methods. An instantiation method similar to e-matching, but with much stronger completeness results, has been published in [9] and is used in Z3; a comparison with our method is in Sect. 5.1.

There is a large body of work on integrating theories into resolution and superposition calculi (e.g., [25,2,13,1]), as well as on the integration of resolution into SMT [17]. These approaches completely avoid e-matching, offering stronger completeness guarantees but limiting the possibility of user-provided guidance.

The model evolution calculus has been extended to theories, including integer arithmetic [4,5]. Our approach resembles model evolution in that it also uses free variables in a tableaux setting, albeit in a more "rigid"/global manner. Further differences are that $\mathcal{ME}(\mathrm{LIA})$ works on clauses, only supports a restricted form of existential quantification, and has a more explicit representation of models.

References

1. Althaus, E., Kruglov, E., Weidenbach, C.: Superposition Modulo Linear Arithmetic SUP(LA). In: Ghilardi, S., Sebastiani, R. (eds.) FroCoS 2009. LNCS, vol. 5749, pp. 84–99. Springer, Heidelberg (2009)

2. Bachmair, L., Ganzinger, H., Waldmann, U.: Refutational theorem proving for hierarchic first-order theories. Appl. Algebra Eng. Commun. Comput. 5 (1994)
3. Barrett, C.W., Tinelli, C.: CVC3. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 298–302. Springer, Heidelberg (2007)
4. Baumgartner, P., Fuchs, A., Tinelli, C.: ME(LIA) - Model Evolution with Linear Integer Arithmetic Constraints. In: Cervesato, I., Veith, H., Voronkov, A. (eds.) LPAR 2008. LNCS (LNAI), vol. 5330, pp. 258–273. Springer, Heidelberg (2008)
5. Baumgartner, P., Tinelli, C.: Model Evolution with Equality Modulo Built-in Theories. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS, vol. 6803, pp. 85–100. Springer, Heidelberg (2011)
6. Bradley, A.R., Manna, Z., Sipma, H.B.: What's Decidable About Arrays? In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 427–442. Springer, Heidelberg (2005)
7. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: A theorem prover for program checking. Journal of the ACM 52(3) (2005)
8. Ge, Y., Barrett, C.W., Tinelli, C.: Solving Quantified Verification Conditions Using Satisfiability Modulo Theories. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 167–182. Springer, Heidelberg (2007)
9. Ge, Y., de Moura, L.: Complete Instantiation for Quantified Formulas in Satisfiabiliby Modulo Theories. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 306–320. Springer, Heidelberg (2009)
10. Halpern, J.Y.: Presburger arithmetic with unary predicates is Π_1^1 complete. Journal of Symbolic Logic 56 (1991)
11. Harrison, J.: Handbook of Practical Logic and Automated Reasoning. Cambridge University Press (2009)
12. Klebanov, V., Müller, P., Shankar, N., Leavens, G.T., Wüstholz, V., Alkassar, E., Arthan, R., Bronish, D., Chapman, R., Cohen, E., Hillebrand, M., Jacobs, B., Leino, K.R.M., Monahan, R., Piessens, F., Polikarpova, N., Ridge, T., Smans, J., Tobies, S., Tuerk, T., Ulbrich, M., Weiß, B.: The 1st Verified Software Competition: Extended experience report (2011)
13. Korovin, K., Voronkov, A.: Integrating Linear Arithmetic into Superposition Calculus. In: Duparc, J., Henzinger, T.A. (eds.) CSL 2007. LNCS, vol. 4646, pp. 223–237. Springer, Heidelberg (2007)
14. Manthey, R., Bry, F.: A hyperresolution-based proof procedure and its implementation in Prolog. In: GWAI, pp. 221–230. Springer, Heidelberg (1987)
15. Manthey, R., Bry, F.: SATCHMO: A Theorem Prover Implemented in Prolog. In: Lusk, E., Overbeek, R. (eds.) CADE 1988. LNCS, vol. 310, pp. 415–434. Springer, Heidelberg (1988)
16. McCarthy, J.: Towards a mathematical science of computation. In: Popplewell, C.M. (ed.) Information Processing 1962, pp. 21–28. North-Holland (1963)
17. de Moura, L., Bjørner, N.S.: Engineering DPLL(T) + Saturation. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 475–490. Springer, Heidelberg (2008)
18. de Moura, L., Bjørner, N.S.: Efficient E-Matching for SMT Solvers. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 183–198. Springer, Heidelberg (2007)
19. de Moura, L., Bjørner, N.S.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)

20. Nelson, G.: Techniques for program verification. Tech. Rep. CSL-81-10, Xerox Palo Alto Research Center (1981)
21. Nelson, G., Oppen, D.C.: Fast decision procedures based on congruence closure. J. ACM 27, 356–364 (1980)
22. Rümmer, P.: Calculi for Program Incorrectness and Arithmetic. Ph.D. thesis, University of Gothenburg (2008)
23. Rümmer, P.: A Constraint Sequent Calculus for First-Order Logic with Linear Integer Arithmetic. In: Cervesato, I., Veith, H., Voronkov, A. (eds.) LPAR 2008. LNCS (LNAI), vol. 5330, pp. 274–289. Springer, Heidelberg (2008)
24. Rümmer, P.: E-matching with free variables. Tech. rep (to appear, 2012)
25. Stickel, M.E.: Automated deduction by theory resolution. Journal of Automated Reasoning 1(4), 333–355 (1985)

Random: R-Based Analyzer for Numerical Domains

Gianluca Amato and Francesca Scozzari

Università "G. d'Annunzio" di Chieti–Pescara — Dipartimento di Scienze

Abstract. We present the tool `Random` (R-based Analyzer for Numerical DOMains) for static analysis of imperative programs. The tool is based on the theory of abstract interpretation and implements several abstract domains for detecting numerical properties, in particular integer loop invariants. The tool combines a statistical dynamic analysis with a static analysis on the new domain of parallelotopes. The tool has a graphical interface for tuning the parameters of the analysis and visualizing partial traces.

1 Introduction

In the abstract interpretation framework [14], the expressive power of an analyzer strictly depends on the choice of the abstract domain. In the last 20 years, many abstract interpretation frameworks have been proposed based on different semantics (see, for instance, [19,10,2]), equipped with many abstract domains, with different trade-offs between expressivity and efficiency. The expressivity of an abstract domain mostly depends on the kind of constraints (assertions) that the abstract domain can represent. The simplest constraint is a constant bound on the value of a program variable, such as $-20 \leq x \leq 100$. The abstract domain of intervals [13], which can handle conjunctions of these constraints, is very efficient but not very expressive, since it cannot prove relationships between variables, such as $x_1 + x_2 \leq 100$. On the contrary, the abstract domain of (convex) polyhedra [15] can represent any linear constraint between program variables, such as $a_1x_1 + a_2x_2 + \ldots + a_nx_n \leq b$, where $x_1, \ldots, x_n$ are program variables and $a_1, \ldots, a_n, b$ are numerical constants (which may be integer, rational or floating point). The abstract domain of polyhedra is very precise for linear constraints but its computational cost is very high.

Many other abstract domains, which reduce the expressive power of general polyhedra while improving efficiency, have been proposed. In most cases, new abstract domains are derived by considering linear constraints subject to syntactic restrictions. This is the case of the difference bound matrices domain [20], which allows only the constraints $a \leq x_1 \leq b$ and $a \leq x_1 - x_2 \leq b$, and for the octagon domain [21] which allows the constraints $a \leq x_1 + x_2 \leq b$ and $a \leq x_1 - x_2 \leq b$. A slight generalization is the two-variables per-inequality domain [26] whose constraints may only contain two variables, such as $a_1x_1 + a_2x_2 \leq b$, and the octahedron abstract domain [12], which can handle constraints whose coefficients are $0, 1, -1$, that is $\pm x_1 \pm x_2 + \ldots \pm x_n \leq b$.

N. Bjørner and A. Voronkov (Eds.): LPAR-18, LNCS 7180, pp. 375–382, 2012.

A different approach has been followed in the template polyhedra abstract domain [24]. This is a parametric domain which, given a finite set $\{a_{1i}x_1 + \ldots + a_{ni}x_n\}_i$ of linear forms fixed *a priori* (the template), allows the constraints $a_{1i}x_1 + \ldots + a_{ni}x_n \leq b_i$. The template approach may be viewed as a generalization of difference bound matrices, octagon and octahedron, since it allows any kind of linear constraints. However, the computational cost of its abstract operators is higher, since any algorithm should be able to deal with any kind and any number of constraints. Moreover, it remains the key problem of how to find the template.

The recently proposed abstract domain of template parallelotopes [3,5] tries to retain the advantages of both approaches. Like template polyhedra, it allows to represent any kind of linear constraint, but the number of constraints in the template is n – the number of variables in the program – and the constraints are required to be linearly independent. Bounding the number of constraints is the key to find very efficient algorithms for the abstract operators. In fact, parallelotopes can be thought of as intervals expressed in a non-standard basis in the vector space of variable's values.

Our tool `Random` (R-based Analyzer for Numerical DOMains) implements three different ways of using parallelotopes. First, we have implemented the template parallelotope domain, using a fixed template to analyze the whole program. In order to find the coefficients in the template, we use two statistical tools, namely the *Principal Component Analysis* (PCA) and the *Independent Component Analysis* (ICA) [17]. Second, we have implemented a template parallelotope approach where we can associate a template to each program point (or to some selected program points). Third, we provide an implementation of the parallelotope abstract domain (without templates), which exploits the full expressive power of parallelotopes. At the end, `Random` annotates each program point of the input program with the constraints discovered by the static analysis.

In the tool, we have privileged the implementation of efficient (and obviously correct) operators, disregarding optimality and completeness (see, for instance, [14,6,16,25]), in particular in the parallelotope abstract domain.

1.1 Template Parallelotopes

The tool can analyze a given program by using the domain of template parallelotopes. In order to find the template, we have implemented a technique based on the pre-analysis of the partial execution traces of the program. Consider the example program in Figure 1. Initially, we run an instrumented version of the program to collect the values of numerical variables in some specific program points for different inputs. Figure 2 shows the set of values collected at the program points ① and ②, where the grey rectangle is the abstraction on the interval domain. Then, we apply to the sample data a statistical technique in order to find the template. The tool implements two different techniques: the Principal Component Analysis and a new technique combining the PCA with the Independent Component Analysis.

```
x = 42
y = 0
while (x>y) {
  ①  x = x-1
  ②  y = y+2
}
```

Fig. 1. The example program

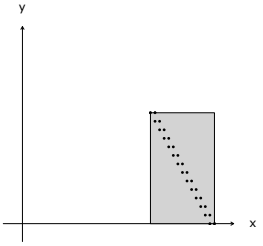


Fig. 2. Interval abstraction of a partial execution trace, observed at program points ① and ②

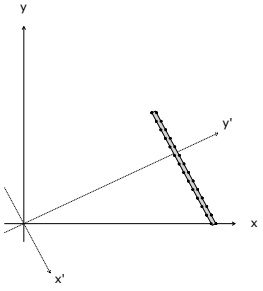


Fig. 3. Parallelotope abstraction with axes rotated according to PCA

The principal component analysis finds a new orthonormal coordinate system maximizing the variance of the collected values. More explicitly, PCA finds new axes such that the variance of the projection of the data points on the first axis is the maximum among all possible directions, the variance of the projection of the data points on the second axis is the maximum among all possible directions which are orthogonal to the first axis, and so on. For instance, if we apply PCA to the values collected from partial executions traces of the program in Figure 1, we get the new basis (x', y') in Figure 3.

On the contrary, the independent component analysis looks for components that are both independent and non-Gaussian. Two variables are independent if knowing something about the value of one variable does not yield any information about the value of the other one. Independence is a stronger property than uncorrelatedness, and it is immediate to see that if two variables are independent, then their covariance is zero. In practice, ICA cannot find a representation whose components are really independent, but it can at least find components that are as independent as possible. We have implemented a combination of PCA and ICA, where we combine the most promising PCA components (those with very low variance) with the most promising ICA components.

The result of the statistical analysis is further refined by a simplification procedure, which stabilizes the result and avoids approximation errors. We supply two simplification procedures which return an approximation of the principal components which are proportional to vectors of small integers. The first procedure, namely the *Orthogonal Simple Component Analysis* (OSCA) [7], minimizes the angle between a principal component and its approximation, while the second procedure analyzes the ratio between the coefficients in a single component.

For the program shown in Figure 1, OSCA finds the change of basis matrix $\begin{bmatrix} 1 & 2 \\ -2 & 1 \end{bmatrix}$ whose columns correspond to the axes (x', y') in Figure 3. With this template, we can represent the constraints $a \leq x - 2y \leq b$ and $c \leq 2x + y \leq d$, thus proving that $2x + y = 84$ holds at the program point ①. Note that none of these constraints may be expressed either in the interval domain or in octagon. The result of the analysis is shown in Figure 4.

```
{
    [ x=0 , y=0 : -x+2*y=0 , 2*x+y=0 ( ) ]
    x = 42
    [ x=42 , y=0 : -x+2*y=-42 , 2*x+y=84 ( ) ]
    y = 0
    [ x=42 , y=0 : -x+2*y=-42 , 2*x+y=84 ( ) ]
    while ({
        [ 27<=x<=42 , 0<=y<=30 : -42<=-x+2*y<=33 , 2*x+y=84 ( ) ]
        x > y
    }) {
        [ 28<=x<=42 , 0<=y<=28 : -42<=-x+2*y<=28 , 2*x+y=84 ( ) ]
        x = x - 1
        [ 27<=x<=41 , 0<=y<=28 : -41<=-x+2*y<=29 , 2*x+y=82 ( ) ]
        y = y + 2
        [ 27<=x<=41 , 2<=y<=30 : -37<=-x+2*y<=33 , 2*x+y=84 ( ) ]
    }
    [ 27<=x<=28 , 28<=y<=30 : 28<=-x+2*y<=33 , 2*x+y=84 ( ) ]
}
```

Fig. 4. The result of the static analysis

1.2 Multiple Template Parallelotopes

The tool is able to change the template in specific program points, marked by a call to the function `.tag(n)`. The parameter `n` is optional and can be used to glue the collected data coming from different program points, thus creating a virtual program point.

`Random` uses statistical tools to compute different templates for any (virtual) program point, and then selects each template in the corresponding program point. In order to switch from a template to the next one, the tool projects the data on the new template with standard algebraic operations. In practice, this approach amounts to partition the source code in fragments, and to apply a different template to each fragment. Thus, in each fragment we can represent up to n different constraints, which must be linearly independent, and do not need to be related to the constraints in other fragments.

Using multiple templates improves the precision of the template parallelotope domain, but can significantly reduce efficiency.

1.3 Parallelotope Abstract Domain

We have implemented the full domain of parallelotopes. The domain changes the template at each program point, according to the operation to be performed. The key points in the design of this domain are the join and widening operators. Both are implemented as a variant of the inverse join [23], which allows us to discover new constraints at a reasonable computational cost. It is also crucial, at least in our implementation, to use delayed widening, so that new invariants may be discovered without being immediately discarded. For instance, consider the following program.

```
x = 1
y = 1
while (y < 100) {
   y = y + y
   y = y + y
   x = x + x
   x = x + x
}
```

The tool starts the analysis with the standard interval domain. After the first two lines, the constraints are $x = 1, y = 1$. At the end of the first while iteration we obtain the constraints $x = 4, y = 4$. Since we use delayed widening, in the first iteration we simply join the two constraints. The inverse join of $x = 1, y = 1$ and $x = 4, y = 4$ yields $-x + y = 0$, $1 \leq x \leq 4$ and $1 \leq y \leq 4$. The heuristic we have developed chooses two linearly independent constraints from the result, in this case $-x + y = 0$ and $1 \leq x \leq 4$. The constraint $-x + y = 0$ is preferred to $1 \leq y \leq 4$ since it is saturated by both constraints $x = 1, y = 1$ and $x = 4, y = 4$.

When processing the assignment $y = y + y$ the analyzer changes the template by transforming $-x + y = 0$ in the new constraint $-2x + y = 0$. After the second assignment $y = y+y$, we get $-4x+y = 0$. Now we process the assignment $x = x+x$ and get the constraints $-2x + y = 0, 2 \leq x \leq 8$, and after the last assignment we get the constraints $-x + y = 0, 4 \leq x \leq 16$. By applying the widening operator, we discard all the constraints which are changed w.r.t. the previous iteration, and we get $-x + y = 0, 1 \leq x$ which is the final result of the analysis.

2 The Tool Interface

Figure 5 shows the tool's interface. On the left side, there are the four main panels. The Source code panel allows to upload and edit a program, while the Analysis result shows the result of the analysis. The Matrix panel shows the templates used in the analysis, while the Partial Trace panel shows the collected values to be analyzed with the statistical engines.

On the right side we may choose the abstract domains and tune the precision. In the first section Partial Execution Traces, we instruct the tool on the program points to be considered and we may select a subset of the variables for the analysis. In the Trace Analysis section we choose the statistical engine (either PCA or ICA), or we can provide a user-defined template matrix. We can also choose whether to use a single template for the whole program or to change the template at the program points selected in the previous section.

In the Trace Analysis Simplification section we choose the simplification strategy to be applied to the result of the statistical engine. The OSCA procedure can be fine-tuned by choosing several parameters. The most important is the Threshold, which misures the distance between the original matrix and the simplified one. In the Static Analysis section we choose the abstract domains to be used in the analysis: the intervals, the template parallelotopes, a combination of the two, or the parallelotope domain. In the last section Output options we can tune

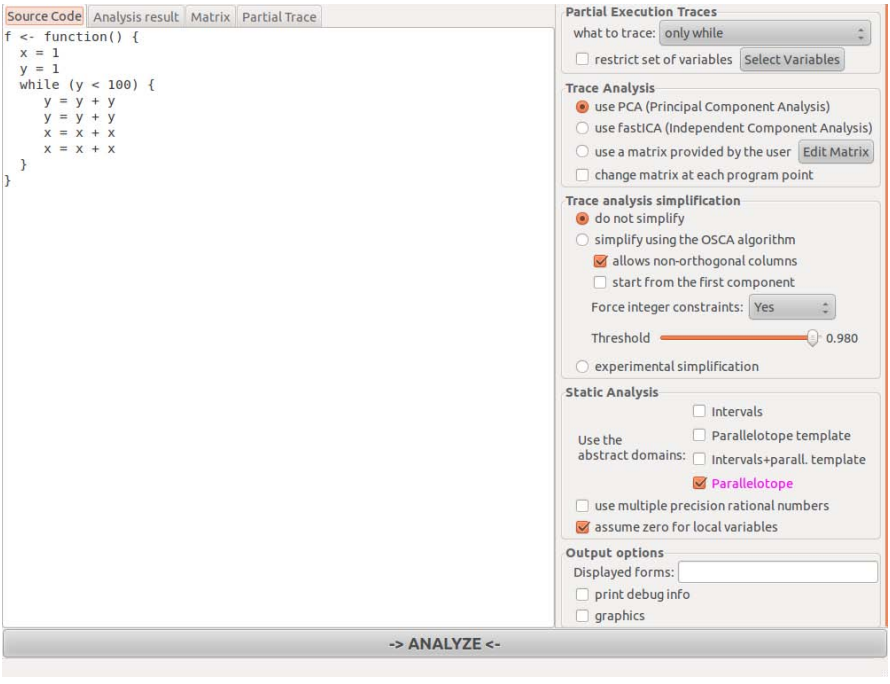


Fig. 5. A screenshot of `Random`

the output of the analysis. The Displayed forms textbox allows to insert a list of linear forms (such as `3*x+2*z`). The result of the analysis is projected on these linear forms. This may be useful to compare the result of analyses performed with different templates. The print debug info option shows, on the console, the intermediate computations of the static analysis, and the graphics option draws the values in the partial execution traces and the principal components.

3 Implementation Details

The tool is available at `http://www.sci.unich.it/~amato/random` under the terms of the GNU GPLv3. A previous and partial version of the tool appeared in [4], without the multiple template parallelotopes, the general parallelotope abstract domain, the ICA analysis, the graphical user interface, the graphical display of partial execution traces and most of the options.

The tool is written in R, a language and environment for statistical computing [22]. It is a functional language with powerful meta-programming features and a vast library of statistical functions. However, it does not excel in efficiency and convenience of debugging facilities.

We analyze programs written in an imperative fragment of the R language, which includes assignments, conditionals and while loops. In addition, the programmer can use the built-in functions `brandom()`, which returns a random

boolean value, and `assume(·)`, in order to make assumptions on program variables, such as `assume(x>0)`. The function `.tag(·)` allows to declare specific program points to be traced. The programmer can insert in the source code the calls `.tag(0)`, `.tag(1)`,`.tag(2)`, ..., even multiple times, in order to create virtual program points. In case of programs with parameters, the user should provide some input values, in order to generate the partial execution traces. The analyzer instruments the program to record the values of the variables in specific program points, computes the partial execution traces starting from the input values, performs the PCA or ICA statistical analysis, simplifies the results, and finally executes the static analysis. PCA and ICA are computed using respectively the standard built-in functions of R and the `fastICA` package.

The static analyzer uses a *recursive chaotic iteration strategy* on the *weak topological ordering* induced by the program structure (see [9]). Correctness of all the abstract operators is ensured by using either rational arithmetic through the *GNU Multiple Precision Arithmetic Library* or, when it is possible, by changing the rounding mode of the floating point arithmetic. To this aim, we have written an auxiliary R package `ieeeround` [1] to control the rounding mode of the CPU.

4 Conclusion and Future Work

In order to improve usefulness and to ease further developments of `Random`, many changes are necessary. First of all, the domains should be ported to `C/C++`, preferably inside well known libraries such as PPL [8] or APRON [18]. The analyzer engine and the program tracer should be ported to a faster and more robust language than R. Finally, the analyzer should support a mainstream target language. To this aim, we could exploit Frama-C [11], an extensible platform for source-code analysis of `C` programs, or the Clang static analyzer (`http://clang-analyzer.llvm.org/`).

References

1. Amato, G.: ieeeround: Functions to set and get the IEEE rounding mode (2011), R package version 0.2-0, `http://CRAN.R-project.org/package=ieeeround`
2. Amato, G., Lipton, J., McGrail, R.: On the algebraic structure of declarative programming languages. Theoretical Computer Science 410(46), 4626–4671 (2009)
3. Amato, G., Parton, M., Scozzari, F.: Deriving Numerical Abstract Domains via Principal Component Analysis. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 134–150. Springer, Heidelberg (2010)
4. Amato, G., Parton, M., Scozzari, F.: A Tool Which Mines Partial Execution Traces to Improve Static Analysis. In: Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G., Roşu, G., Sokolsky, O., Tillmann, N. (eds.) RV 2010. LNCS, vol. 6418, pp. 475–479. Springer, Heidelberg (2010)
5. Amato, G., Parton, M., Scozzari, F.: Discovering invariants via simple component analysis. Journal of Symbolic Computation (to appear, 2012), doi:10.1016/j.jsc.2011.12.052
6. Amato, G., Scozzari, F.: Optimality in goal-dependent analysis of sharing. Theory and Practice of Logic Programming 9(5), 617–689 (2009)

7. Anaya-Izquierdo, K., Critchley, F., Vines, K.: Orthogonal simple component analysis: a new, exploratory approach. Annals of Applied Statistics 5(1), 486–522 (2011)
8. Bagnara, R., Hill, P.M., Zaffanella, E.: The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. Science of Computer Programming 72(1–2), 3–21 (2008)
9. Bourdoncle, F.: Efficient Chaotic Iteration Strategies with Widenings. In: Pottosin, I.V., Bjorner, D., Broy, M. (eds.) FMP&TA 1993. LNCS, vol. 735, pp. 128–141. Springer, Heidelberg (1993)
10. Bruynooghe, M.: A practical framework for the abstract interpretation of logic programs. The Journal of Logic Programming 10(1/2/3 & 4), 91–124 (1991)
11. Canet, G., Cuoq, P., Monate, B.: A value analysis for C programs. In: SCAM 2009, Proceedings, pp. 123–124. IEEE Computer Society Press (2009)
12. Clarisó, R., Cortadella, J.: The octahedron abstract domain. Science of Computer Programming 64, 115–139 (2007)
13. Cousot, P., Cousot, R.: Static determination of dynamic properties of programs. In: Proc. Second Int'l Symposium on Programming, Dunod, pp. 106–130 (1976)
14. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: POPL 1979, Proc., pp. 269–282. ACM Press (1979)
15. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL 1978, Proc., pp. 84–97. ACM Press (1978)
16. Giacobazzi, R., Ranzato, F., Scozzari, F.: Making abstract domains condensing. ACM Transactions on Computational Logic 6(1), 33–60 (2005)
17. Hyvärinen, A., Karhunen, J., Oja, E.: Independent Component Analysis. John Wiley & Sons (2001)
18. Jeannet, B., Miné, A.: Apron: A Library of Numerical Abstract Domains for Static Analysis. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 661–667. Springer, Heidelberg (2009)
19. Marriott, K., Søndergaard, H., Jones, N.D.: Denotational abstract interpretation of logic programs. ACM Transactions on Programming Languages and Systems 16(3), 607–648 (1994)
20. Miné, A.: A New Numerical Abstract Domain Based on Difference-Bound Matrices. In: Danvy, O., Filinski, A. (eds.) PADO 2001. LNCS, vol. 2053, pp. 155–172. Springer, Heidelberg (2001)
21. Miné, A.: The octagon abstract domain. Higher-Order and Symbolic Computation 19(1), 31–100 (2006)
22. R Development Core Team. R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria (2011), http://www.R-project.org/
23. Sankaranarayanan, S., Colón, M., Sipma, H.B., Manna, Z.: Efficient Strongly Relational Polyhedral Analysis. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 111–125. Springer, Heidelberg (2005)
24. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Scalable Analysis of Linear Systems Using Mathematical Programming. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 25–41. Springer, Heidelberg (2005)
25. Scozzari, F.: Abstract Domains for Sharing Analysis by Optimal Semantics. In: SAS 2000. LNCS, vol. 1824, pp. 397–412. Springer, Heidelberg (2000)
26. Simon, A., King, A., Howe, J.M.: Two Variables Per Linear Inequality as an Abstract Domain. In: Leuschel, M. (ed.) LOPSTR 2002. LNCS, vol. 2664, pp. 71–89. Springer, Heidelberg (2003)

Solving Graded/Probabilistic Modal Logic via Linear Inequalities (System Description)

William Snell, Dirk Pattinson, and Florian Widmann

Dept. of Computing, Imperial College, London

Abstract. We present the experience gained from implementing a new decision procedure for both graded and probabilistic modal logic. While our approach uses standard tableaux for propositional connectives, modal rules are given by linear constraints on the arguments of operators. The implementation uses binary decision diagrams for propositional connectives and a linear programming library for the modal rules. We compare our implementation, for graded modal logic, with other tools, showing average performance. Due to lack of other implementations, no comparison is provided for probabilistic modal logic, the main new feature of our implementation.

Introduction

Both graded modal logic [9] and probabilistic modal logic [7] extend (classical) propositional logic by modal connectives that constrain the number of successor states in a Kripke frame or the probability of events in a Markov chain.

Graded modalities are used in description logic to represent number restrictions in (extensions of) the logic $\mathcal{ALCQ}$ [1], which is supported by a large number of tools such as Pellet [20], Fact++ [21], RACER [11] or Hermit [16]. Probabilities do not feature as prominently in description logic, where they do not describe Markov chains but model uncertainty in the form of a single distribution over non-probabilistic models [14].

Most formalisms for describing the behaviour of probabilistic systems [17,6] rely on probabilistic modal logic in some form or other. While dedicated model checkers such as Prism [13] support the analysis of probabilistic systems, there is currently no tool support for *reasoning* in probabilistic (modal) logics with Markov-chain semantics.

This paper is a first step towards filling this gap: we present the experience gained from implementing a satisfiability checker that supports both graded and probabilistic modal logic in the same framework. The calculus that we implement uses standard tableau rules together with linear inequalities to describe the arithmetic or probabilistic constraints on successor states/events. The distinguishing feature of this calculus (see [19] for the dual treatment of validity) is that it does *not* try to construct a model. Instead, it produces linear constraints whose (un-)solvability guarantees the *existence* of a model. The calculus proceeds in two steps: in the first step, propositional connectives are eliminated in a standard way, leading to conjunctive clauses over modal literals. The modal rules applied subsequently are given in terms of *linear inequalities* over the arguments of the operators in the premise. Every solution (together with a side condition) represents a modal rule. The structural similarity of the rules for graded and probabilistic modal logic allows us to treat both logics in the same framework.

N. Bjørner and A. Voronkov (Eds.): LPAR-18, LNCS 7180, pp. 383–390, 2012.

The implementation of this calculus naturally reflects both steps. We represent modal formulae using binary decision diagrams [2] where modal atoms are represented by (BDD) variables. Propositional tableau rules then correspond to analysing all satisfying assignments of the corresponding BDD. In a second step, we calculate the set of all modal rules applicable to a set of modal literals where solvability of an associated linear programming problem is equivalent to validity of the corresponding tableau rule. Compared to other tools, our implementation of graded modal logic shows average performance, but allows for the treatment of probabilistic modalities in the same framework which is not currently supported by other implementations.

To eliminate artefacts such as the particular choice of blocking or caching techniques used, we focus on the satisfiability problem of both logics over empty TBoxes. We describe syntax and semantics of both graded and probabilistic modal logic and briefly introduce the underlying calculus. We then discuss implementation details and present a brief comparison with other tools.

Related Work. Linear inequalities have been used for graded modal logic as an addition to a tableau calculus [8] where they are used (along with the usual completion rules) to detect inconsistency in ABoxes. Our approach, in contrast, uses linear inequalities directly to determine valid completion steps (tableau rules). There are also approaches reducing $\mathcal{ALCQ}$ to SMT(C) [12]. A proof-of-concept implementation demonstrating compositionality of modal logics, containing graded and probabilistic modal logic as building blocks, was presented in [4] using a naive brute-force approach that was orders of magnitude slower than other tools (including the system presented here).

1 Syntax, Semantics and Tableau Calculus

Both graded modal logic and probabilistic modal logic are extensions of propositional logic by unary modal operators, given by the grammar

$$\mathcal{L} \ni \phi, \psi ::= p \mid \phi \wedge \psi \mid \neg\phi \mid \langle x \rangle \phi$$

where $p \in \mathcal{V}$ is a propositional variable and where $x \in \mathbb{N}$ in the case of graded modal logic and $x \in [0,1] \cap \mathbb{Q}$ for probabilistic modal logic.

For graded modal logic, we read $\langle n \rangle \phi$ as 'ϕ holds in *more than n* successor worlds, and the probabilistic operator $\langle p \rangle \phi$ stipulates that 'ϕ holds with probability $\geq p$ in the next state'. We interpret graded modal logic over *multigraph frames* [5] (W, σ, π) where W is a set of worlds, $\sigma : W \to \mathcal{B}(W)$ assigns a finite multiset (a 'bag', formally a function $W \to \mathbb{N}$ with finite support) of elements of W to every world w and $\pi : W \to \mathcal{P}(\mathcal{V})$ is a valuation of the propositional variables. Truth of a formula at a point is defined in the standard way, with the clause on the left below for the modal operators.

$$w \models \langle n \rangle \phi \iff \sum_{w' \models \phi} \sigma(w)(w') > n \qquad w \models \langle p \rangle \phi \iff \sum_{w' \models \phi} \sigma(w)(w') \geq p$$

Similarly, probabilistic modal logic is interpreted over discrete Markov chains (W, σ, π) where W and π are as above, and σ assigns a finitely supported probability distribution (formally a function $\mu : W \to [0,1]$ with finite support such that $\sum_{w \in W} \mu(w) = 1$) to

every world w. For modal operators, truth is given by the clause on the right above, together with the standard clauses for the remaining propositional connectives.

While graded modal logic is usually interpreted over Kripke frames (rather than multigraphs), it is easy to see that both semantics induce the same satisfiability problem. We obtain truth-preserving translations by considering a Kripke frame as a multigraph where every edge has multiplicity one, and multigraphs can be converted to Kripke frames by inserting the appropriate number of copies of each successor state. Similarly, the probabilistic logics of non-discrete and discrete Markov chains are equivalent.

Our focus in this paper is the *satisfiability problem* for graded and probabilistic modal logic, and as usual, a formula $\phi \in \mathcal{L}$ is *satisfiable* iff there exists a model (W, σ, π) and a world $w \in W$ such that $w \models \phi$. Our implementation is based on the following characterisation of satisfiability in terms of a tableau calculus over *tableau sequents*, that is, finite sets of formulae that we read conjunctively. We have the standard rules

$$\frac{\Gamma, p, \neg p}{} \quad \frac{\Gamma, A \wedge B}{\Gamma, A, B} \quad \frac{\Gamma, \neg(A \wedge B)}{\Gamma, \neg A \quad \Gamma, \neg B} \quad \frac{\Gamma, \neg\neg A}{\Gamma, A} \quad \frac{\langle x_0\rangle p_0, \ldots, \langle x_n\rangle p_n, \neg\langle y_0\rangle q_0, \ldots, \neg\langle y_m\rangle q_m}{\sum_{i=0}^{n} r_i p_i - \sum_{j=0}^{m} s_j q_j > k}$$

together with all substitution instances of the rule on the right for graded and probabilistic modal logic, subject to (different) side conditions, depending on whether we deal with graded or probabilistic modal logic. For graded modal logic, we have $k = 0$ and require the side condition on the left below, while for probabilistic modal logic,

$$\sum_{1\le i\le n} r_i(x_i + 1) - \sum_{1\le j\le m} s_j y_j \ge 1 \qquad \sum_{1\le i\le n} r_i x_i - \sum_{1\le j\le m} s_j y_j \rhd k$$

we have $k \in \mathbb{Z}$ and require the side condition on the right above, where $\rhd$ equals $\ge$ if $m > 0$ and $\rhd$ equals $>$ if $m = 0$. For both logics, $r_i, s_j \in \mathbb{N} \setminus \{0\}$.

The linear inequality in the rule conclusions is a compact shorthand, and denotes a (possibly empty) set of conclusions. More precisely, we associate to each function $v : \{p_1, \ldots, p_n, q_1, \ldots, q_m\} \to \{0, 1\}$ a propositional sequent $s(v)$ containing p_i if $v(p_i) = 1$ and $\neg p_i$ if $v(p_i) = 0$, analogously for q_j. The inequality $\sum_{i=0}^{n} r_i p_i - \sum_{j=0}^{m} s_j q_j > k$ then stands for the set of conclusions containing all $s(v)$ for which $\sum_{i=0}^{n} r_i v(p_i) - \sum_{j=0}^{m} s_j v(q_j) > k$. That is, the conclusion encodes the set of binary valuations, seen as a tableau sequent, for which the inequality holds.

The calculus above is sound and complete for both graded and probabilistic modal logic [19,15]. That is, a sequent Γ is (conjunctively) unsatisfiable iff there exists a closed tableau with root Γ. In particular, *every* solution of the side condition in terms of weights r_j and s_j induces a modal rule, but one can establish a polynomial bound on the weights retaining completeness of the calculus [19, Lemma 6.16].

2 Implementation Details

We describe the representation of formulae in terms of binary decision diagrams, the representation of tableau rules and the reasoning algorithm.

Representation of Formulae. We represent formulae of both graded and probabilistic modal logic using binary decision diagrams [3], where propositional variables and modal atoms (that is, formulae of the form $\langle x\rangle\phi$) are represented using BDD variables. A tableau sequent is represented by the conjunction of the formulae it contains. This allows us to effectively delegate propositional reasoning to binary decision diagrams: every satisfying valuation of (the BDD representing) a tableau sequent Γ corresponds to a leaf of a propositional tableau with root Γ. We maintain a look-up table to ensure that multiple occurrences of the same propositional variable are represented by the same BDD variable. In particular, modal atoms that have propositionally equivalent arguments are presented by the same BDD variable. We note two consequences.

Deep Congruence. We call two formulae *congruence equivalent* if their equivalence can be established by propositional reasoning and the congruence rule $\phi \leftrightarrow \psi / \heartsuit\phi \leftrightarrow \heartsuit\psi$ where $\heartsuit$ is a modal operator. As modal atoms with propositionally equivalent arguments are represented by the *same* BDD variable, congruence-equivalent formulae are represented by the same BDD.

Semantic Branching. The implementation of propositional tableau rules by means of computing satisfying assignments of the associated BDD implies an implicit use of semantic branching. This is a consequence of computing satisfying assignments using BDDs, where the set of satisfying assignments corresponds to the set of paths from the root node to the leaf representing logical truth. For example, the set of satisfying assignments of the (BDD encoding of the) formula $p_0 \vee p_1 \vee p_2$ will be p_0, $\neg p_0 \wedge p_1$ and $\neg p_0 \wedge \neg p_1 \wedge p_2$ in the variable order $p_0 < p_1 < p_2$.

While BDDs conveniently relieve us of the task of implementing the rules for propositional reasoning, one obvious disadvantage of using BDDs as a 'black box' is the impossibility of implementing other optimisations, in particular backjumping.

Application of Modal Rules. Given a sequent Γ consisting of possibly negated modal atoms and propositional variables, we compute the set of modal rules applicable to Γ on the fly. While every solution of the side condition gives rise to an instance of the rule, we can compute a polynomial bound on the size of the search space: Lemma 17.1b in [18] guarantees that all possible rules are generated if we limit the weights to non-negative integers of (binary) size $\leq 6n^3w$ where w is the size of the side condition and n is the number of modal literals. As numbers are represented in binary, the search space to be explored is of size 2^{6n^4w} (as every of the n coefficients may vary between 0 and 2^{6n^3w}). This bound is exponential unlike the doubly exponential bound 2^{2^n} on the number of possible rules applicable to a sequent containing n modal literals. In practice, the doubly exponential bound 2^{2^n} will be below the size of the search space computed above for sequents containing less than approximately 34 modal literals, which appears to be enough to cover all practical applications. We therefore implement rule generation using a doubly exponential algorithm. Given a sequent Γ consisting of n positive modal atoms of the form $\langle x\rangle p_i$ with arguments $p_1, \ldots, p_n$ and m negated modal atoms $\neg\langle x\rangle q_j$ with arguments $q_1, \ldots, q_m$, we iteratively check for all propositional formulae ϕ in variables $p_1, \ldots, p_n, q_1, \ldots, q_m$ whether $\Gamma / \mathsf{dnf}(\phi)$ is a valid rule by encoding both the side condition and the conclusion ϕ into a system of linear inequalities. (Here $\mathsf{dnf}(\phi)$ denotes the disjunctive normal form of ϕ in the form of a set of tableau sequents.) In

more detail, $\Gamma/\mathsf{dnf}(\phi)$ is a rule of graded / probabilistic modal logic if the system of linear inequalities consisting of the side condition together with the inequalities

$$\begin{cases} \sum_{i=1}^{n} r_i v(p_i) - \sum_{j=1}^{m} s_j v(q_j) > k & \text{if } \phi \text{ evaluates to } \top \text{ under } v \\ \sum_{i=1}^{n} r_i v(p_i) - \sum_{j=1}^{m} s_j v(q_j) \leq k & \text{otherwise,} \end{cases}$$

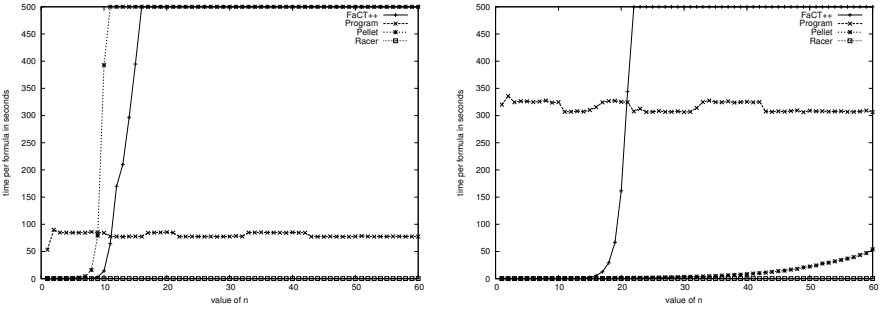
where v ranges over all valuations $\{p_1, \ldots, p_n, q_1, \ldots, q_m\} \to \{0, 1\}$, has a solution. We use an external library [10] to determine whether systems of linear inequalities have a solution. This (doubly exponential) search space can be pruned significantly.

Rule Subsumption. We say that a modal rule $\Gamma/\mathsf{dnf}(\phi)$ *subsumes* the rule $\Gamma/\mathsf{dnf}(\psi)$ if $\phi \to \psi$ is propositional tautology. Intuitively, the conclusion ϕ is 'harder' to satisfy, so that any application of $\Gamma/\mathsf{dnf}(\psi)$ in a closed tableau can be replaced by an application of $\Gamma/\mathsf{dnf}(\phi)$. In other words, omitting the rule $\Gamma/\mathsf{dnf}(\psi)$ does not jeopardise completeness of the calculus. Given a sequent Γ consisting of positive atoms $\langle x \rangle p_i$ with arguments $p_1, \ldots, p_n$ and negative modal atoms $\neg\langle x \rangle q_j$ with arguments $q_1, \ldots, q_m$, we arrange the set of propositional formulae in variables $\{p_1, \ldots, p_n, q_1, \ldots, q_m\}$ in a directed, rooted, acyclic graph where the edge relation represents subsumption. Let $\Theta = \{v_1, \ldots, v_{2^{n+m}}\}$ be the set of all possible valuations of the propositional variables present. A node in the subsumption graph contains a subset $N \subseteq \Theta$ of valuations that represent the propositional formula $\phi_N = \bigvee_{v \in \Theta} \bigwedge_{v(a)=1} a \wedge \bigwedge_{v(a)=0} \neg a$ where a ranges over the set $\{p_1, \ldots, p_n, q_1, \ldots, q_m\}$ of propositional variables. Two (different) nodes N and M are related by a direct edge $N \to M$ if $M = N \cup \{v_j\}$ and $j > \max\{1 \leq i \leq 2^{n+m} \mid v_i \in N\}$, and the root of the subsumption graph is the empty set. This arrangement guarantees that a rule with conclusion ϕ_N subsumes all rules with conclusions ϕ_M provided that N is a (direct or indirect) ancestor of M. We generate rule conclusions using *breadth first* search through the subsumption tree, pruning all children of a node N as soon as it has been established that $\Gamma/\mathsf{dnf}(N)$ is a valid rule instance.

Reasoning Algorithm. Completeness of the tableau calculus allows us to reduce satisfiability of the root formula to the non-existence of a closed tableau. We check for the existence of a closed tableau using standard depth-first search. At the moment, we have not implemented any optimisations such global caching or unsat caching.

3 Experimental Evaluation

As mentioned already, we are not aware of any other reasoner that supports probabilistic modal logic. We therefore restrict ourself to graded modal logic in this section. We do not claim to conducting a thorough system comparison, but merely want to gauge whether the approach described in the previous section is feasible. In particular, we only consider satisfiability over the empty TBox to factor out artefacts like the particular choice of caching or blocking techniques used and concentrate on formulae whose main "complexity" is due to graded modalities. All formulae, and our tool Program: *Pro*babilistic and *Gra*ded *M*odalities, are available on the web at `http://www.doc.ic.ac.uk/~dirk/Software/Program/`.



(a) $\phi_{\text{unsat}}(n)$ (not satisfiable) (b) $\phi_{\text{sat}}(n)$ (satisfiable)

Fig. 1. Increasing the cardinality

Table 1. Randomly generated formulae

No. of timeouts	n = 10	20	30	40	50	60	70	80	90	100
Fact	1	0	2	4	4	5	3	6	6	8
Program	0	1	5	20	23	40	33	55	61	72
Pellet	3	13	31	30	41	46	42	60	57	71
Racer	0	0	0	0	0	0	0	0	0	0

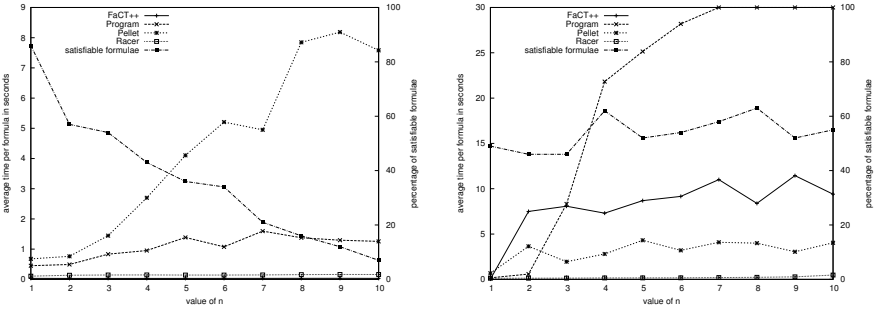
We compare our prover (Program) with Fact++ [21], Pellet [20], and RACER [11]. The benchmarks were run on a 64 bit Linux system with an Intel Core i5-650 CPU and a memory limit of 4 GB, and out of memory errors were treated as timeouts.

The first two sets of benchmarks show how the provers are affected by increasing numbers in cardinality constraints. The formulae are of the form

$$\phi_x(n) = \langle n-1\rangle\neg p_1 \wedge \langle n-1\rangle p_1 \wedge \neg\langle n\rangle p_0 \wedge \neg\langle m_x\rangle\neg p_0$$

where $x \in \{\text{unsat}, \text{sat}\}$ and $m_{\text{unsat}} = n-1$ and $m_{\text{sat}} = n$. Figure 1a shows the results for $\phi_{\text{unsat}}(n)$ for $1 \leq n \leq 60$ and a timeout of 500 seconds. Since Fact++ and Pellet appear to treat number restrictions naively, they fail for larger instances, whereas Program and Racer are not affected by the cardinality. In the satisfiable case, see Fig. 1b, Program takes longer because all modal rule applications have to be constructed, whereas Pellet behaves much better, presumably because it can find a model fast.

As a sanity check, we also used randomly generated formulae without any structure. Despite not being overly informative, it is useful for testing provers for discrepancies. (Indeed, one bug in Fact++ was discovered which has been fixed prior to our benchmarks.) The size of these formulae, denoted by n, is the number of symbols, counting 1 for cardinalities. The formulae were created by randomly choosing a connective with equal probability and recursively creating the subformulae. Cardinalities were chosen randomly between 0 and 99 inclusive. We tested 100 formulae for each size with a timeout of 30 seconds. Table 1 shows the number of timeouts relative to the size (n) of formulae, as this is more informative than average runtime.



(a) $\psi_{\text{unsat}}(n)$ (tree-shaped) (b) $\chi(1, m)$ (constraints)

Fig. 2. Tree-shaped models and simple constraint problems

Table 2. $\chi(2, 2)$ (constraints)

	Fact	Program	Pellet	Racer
Number of timeouts	15	0	7	0
Average time per formula	123s	79s	80s	<1s

The next benchmark formulae enforce tree shaped models and are of the form

$$\psi(0) = \top \qquad \text{and} \qquad \psi(n) = \langle c_1^n - 1\rangle\top \wedge \neg\langle c_2^n\rangle\top \wedge \neg\langle 0\rangle\neg\psi(n-1)$$

where $1 \leq c_1^n, c_2^n \leq 100$ are drawn randomly so that $c_1^n \leq c_2^n$ with probability 0.8. For each $1 \leq n \leq 10$ we generated 100 formulae and the average time (per n) is plotted in Fig. 2a. The timeout was 30 seconds which was only exceeded by Pellet occasionally. The figure also shows the percentage of satisfiable formulae which naturally drops as the depth of the trees increases.

The last two sets of benchmarks mimic simple constraint problems of the form

$$\chi(n, m) = \bigwedge_{i=1}^{n} \langle c_i\rangle(wish_i \wedge disj_i) \;\wedge\; \bigwedge_{j=1}^{m} \neg\langle d_j\rangle r_j$$

where $0 \leq c_i, d_j < 100$ are drawn randomly and $wish_i$ is a simple randomly generated propositional formula in the variables r_j and $disj_i$ is a (fixed) propositional formula not containing r_j such that $disj_i \wedge disj_{i'}$ is unsatisfiable for $i \neq i'$: if two children want ice cream (r_0) and three want ice cream or an apple (r_1), and you have two ice creams and three apples, we get the formula $\langle 1\rangle(r_0 \wedge q) \wedge \langle 2\rangle((r_0 \vee r_1) \wedge \neg q) \wedge \neg\langle 2\rangle r_0 \wedge \neg\langle 3\rangle r_1$. Figure 2b shows the results for $n = 1$ fixed. For each m we tested 100 formulae, again with a timeout of 30 seconds. Unfortunately, Program cannot compete for large m as the performance of the linear solver degrades when the number of modal formulae in a sequent becomes too big. We also tested 30 formulae of the form $\chi(2, 2)$ with a timeout of 300 seconds and the results are given in Table 2.

4 Conclusion

While our implementation of graded modal logic was not quantitatively better in comparison to other tools, our implementation also supports probabilistic modal logic. The

experimental results indicate that the method itself can be made competitive. The structural similarity between graded and probabilistic modal logic insinuates that this also applies to probabilistic modal logic, and our tests with probabilistic modal logic formulae show roughly equivalent performance (comparing formula size).

References

1. Baader, F., Nutt, W.: Basic description logics. In: Baader, F., et. al. (ed.) Description Logic Handbook, pp. 43–95. Cambridge University Press (2003)
2. Bryant, R.E.: Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams. ACM Computing Surveys 24(3), 293–317 (1992)
3. BuDDy, http://sourceforge.net/projects/buddy/
4. Calin, G., Myers, R., Pattinson, D., Schröder, L.: COLOSS: The coalgebraic logic satisfiability solver. In: Proc. M4M 5 (2007). ENTCS, vol. 231, pp. 41–54 (2009)
5. D'Agostino, G., Visser, A.: Finality regained: A coalgebraic study of Scott-sets and multisets. Arch. Math. Logic 41, 267–298 (2002)
6. Desharnais, J., Edalat, A., Panangaden, P.: Bisimulation for labelled markov processes. Inf. Comput. 179(2), 163–193 (2002)
7. Fagin, R., Halpern, J.: Reasoning about knowledge and probability. J. ACM 41, 340–367 (1994)
8. Farsiniamarj, N., Haarslev, V.: Practical reasoning with qualified number restrictions: a hybrid Abox calculus for the description logic SHQ. AI Comms. 23(2–3), 205–240 (2010)
9. Fine, K.: In so many possible worlds. Notre Dame J. Formal Logic 13, 516–520 (1972)
10. GNU linear programming kit (glpk), http://www.gnu.org/s/glpk/
11. Haarslev, V., Möller, R.: RACER System Description. In: Goré, R.P., Leitsch, A., Nipkow, T. (eds.) IJCAR 2001. LNCS (LNAI), vol. 2083, pp. 701–705. Springer, Heidelberg (2001)
12. Haarslev, V., Sebastiani, R., Vescovi, M.: Automated Reasoning in $\mathcal{ALCQ}$ via SMT. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS, vol. 6803, pp. 283–298. Springer, Heidelberg (2011)
13. Hinton, A., Kwiatkowska, M., Norman, G., Parker, D.: PRISM: A Tool for Automatic Verification of Probabilistic Systems. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 441–444. Springer, Heidelberg (2006)
14. Klinov, P.: Practical Reasoning in Probabilistic Description Logic. PhD thesis, University of Manchester, Manchester, UK (2011)
15. Kupke, C., Pattinson, D.: On modal logics of linear inequalities. In: Goranko, V., Shehtman, V. (eds.) Proc. AiML 2010. College Publications (2010)
16. Motik, B., Shearer, R., Horrocks, I.: Hypertableau Reasoning for Description Logics. Journal of Artificial Intelligence Research 36, 165–228 (2009)
17. Parma, A., Segala, R.: Logical Characterizations of Bisimulations for Discrete Probabilistic Systems. In: Seidl, H. (ed.) FOSSACS 2007. LNCS, vol. 4423, pp. 287–301. Springer, Heidelberg (2007)
18. Schrijver, A.: Theory of linear and integer programming. Wiley Interscience (1986)
19. Schröder, L., Pattinson, D.: PSPACE bounds for rank-1 modal logics. ACM Transactions on Computational Logics 10(2) (2009)
20. Sirin, E., Parsia, B., Cuenca Grau, B., Kalyanpur, A., Katz, Y.: Pellet: A practical OWL-DL reasoner. Journal of Web Semantics (2006)
21. Tsarkov, D., Horrocks, I.: FaCT++ Description Logic Reasoner: System Description. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 292–297. Springer, Heidelberg (2006)

Labelled Superposition for PLTL

Martin Suda[1,2,3,⋆] and Christoph Weidenbach[1,⋆⋆]

[1] Max-Planck-Institut für Informatik, Saarbrücken, Germany
[2] Saarland University, Saarbrücken, Germany
[3] Charles University, Prague, Czech Republic

Abstract. This paper introduces a new decision procedure for PLTL based on labelled superposition. Its main idea is to treat temporal formulas as infinite sets of purely propositional clauses over an extended signature. These infinite sets are then represented by finite sets of labelled propositional clauses. The new representation enables the replacement of the complex temporal resolution rule, suggested by existing resolution calculi for PLTL, by a fine grained repetition check of finitely saturated labelled clause sets followed by a simple inference. The completeness argument is based on the standard model building idea from superposition. It inherently justifies ordering restrictions, redundancy elimination and effective partial model building. The latter can be directly used to effectively generate counterexamples of non-valid PLTL conjectures out of saturated labelled clause sets in a straightforward way.

1 Introduction

Propositional linear temporal logic [15] is an extension of classical propositional logic for reasoning about time. It introduces temporal operators such as $\Diamond P$ meaning P holds *eventually* in the future, $\Box P$ meaning P holds *always* in the future, and $\bigcirc P$ meaning P holds at the *next* time point. Time is considered to be a linear discrete sequence of time points represented by propositional valuations, called *worlds*. Such a potentially infinite sequence forms a PLTL interpretation. A decision procedure for PLTL takes a PLTL formula P and checks whether it is valid, i.e., that all PLTL interpretations are actually models for P. For example, the PLTL formula $\Box P \rightarrow \bigcirc P$ is valid (a theorem) whereas the PLTL formula $\bigcirc P \rightarrow \Box P$ is not, but is satisfiable, i.e., there is a PLTL model for it.

Attempts to use clausal resolution to attack the decision problem for PLTL appeared first in [3, 21]. The most recent resolution-based approach is the one of [6]. It relies on a satisfiability preserving clausal translation of PLTL formulas, where, in particular, all nestings of temporal operators are reduced to formulas (and, eventually, clauses) of the form P, $\Box(P \rightarrow \bigcirc Q)$, and $\Box(P \rightarrow \Diamond Q)$, where P and Q do not contain temporal operators. Classical propositional resolution is extended to cope with "local" temporal reasoning within neighbouring worlds,

⋆ Supported by Microsoft Research through its PhD Scholarship Programme.
⋆⋆ Supported by the German Transregional Collaborative Research Center SFB/TR 14 AVACS.

N. Bjørner and A. Voronkov (Eds.): LPAR-18, LNCS 7180, pp. 391–405, 2012.

while an additional inference rule called *temporal resolution* is introduced to deal with *eventuality* ($\Box\Diamond$) clauses. The temporal resolution rule is quite complex. It requires a search for certain combinations of clauses that together form a *loop*, i.e. imply that certain sets of worlds must be discarded from consideration, because an eventuality clause would be unsatisfied forever within them. This is verified via an additional proof task. Finally, the conclusion of the rule needs to be transformed back into the clause form.

Our labelled superposition calculus builds on a refinement of the above clause normal form [4]. It introduces a notion of labelled clauses in the spirit of [12] and replaces the temporal resolution rule by saturation and a new *Leap* rule. Although in PLTL equality is not present, the principles of superposition are fundamental for our calculus. Our completeness result is based on a model generation approach with an inherent redundancy concept based on a total well-founded ordering on the propositional atoms.

The main contributions of our paper are: 1) we replace the temporal resolution rule by a much more streamlined saturation of certain labelled clauses followed by a simple Leap inference, 2) our inference rules are guided by an ordering restriction that is known to reduce the search space considerably, 3) the completeness proof justifies an abstract redundancy notion that enables strong reductions, 4) if a contradiction cannot be derived, a temporal model can be extracted from a saturated clause set.

The paper is organized as follows. We fix our notation and formalize the problem to be solved in Sect. 2. Then in Sect. 3 we show how to use labelled clauses as a tool to "lift" the standard propositional calculus to reason about PLTL-satisfiability. Our calculus is introduced in Sect. 4 and used as a basis for an effective decision procedure in Sect. 5. We deal with abstract redundancy, its relation to the completeness proof, and model building in Sect. 6. Discussion of previous work and an experimental comparison to existing resolution approaches appear in Sect. 7. Finally, Sect. 8 concludes. Detailed proofs and additional material are available in a technical report [20].

2 Preliminaries

In this section we fix our notation and briefly recall the standard notions we will be using. We assume the reader to be familiar with ordered resolution calculus for propositional logic and its completeness proof [2] including the concepts of redundancy, saturation, and model construction. In the following, the symbol $\mathbb{N}$ stands for the naturals, and $\mathbb{N}^+$ denotes the set $\mathbb{N} \setminus \{0\}$.

The language of propositional formulas and clauses over a given signature $\Sigma = \{p, q, \ldots\}$ of propositional variables is defined in the usual way. We denote propositional clauses by letters C or D, possibly with subscripts, and understand them as multisets of literals. By propositional *valuation*, or simply a *world*, we mean a mapping $V : \Sigma \to \{0, 1\}$. We write $V \models P$ if a propositional formula P is satisfied by V. The semantics of PLTL is based on a discrete linear model of time, where the structure of possible time points is isomorphic to $\mathbb{N}$. A *PLTL-interpretation* is a sequence $(V_i)_{i\in\mathbb{N}}$ of propositional valuations.

In order to be able to talk about several neighbouring worlds at once we introduce copies (i.e. pairwise disjoint, bijectively equivalent sets) of the basic signature Σ. We use priming to denote the shift from one signature to the next (thus Σ' is the set of symbols $\{p', q', \ldots\}$), and shorten repeated primes by parenthesised integers (e.g. p''' is the same thing as $p^{(3)}$). This notational convention can be extended from symbols and signatures to formulas, and also to valuations in a natural way. For example, if V is a valuation over $\Sigma^{(i)}$ we write V' for the valuation over $\Sigma^{(i+1)}$ such that $V'(p^{(i+1)}) = V(p^{(i)})$ for every $p \in \Sigma$. We also need to consider formulas over two consecutive joined signatures, e.g. over $\Sigma \cup \Sigma'$. Such formulas can be evaluated over the respective joined valuations. When both V_1 and V_2 are valuations over Σ, we write $[V_1, V_2]$ as a shorthand for the mapping $V_1 \cup (V_2)' : (\Sigma \cup \Sigma') \to \{0, 1\}$.

As usual in refutational theorem proving, our method starts with negating the input formula and translating the result into a clause normal form. Then the task is to show that the translated form is unsatisfiable, which implies validity of the original formula. We build on the *Separated Normal Form* to which any PLTL formula can be translated by a satisfiability preserving transformation with at most linear increase in size [6, 4]. We skip here the details of the transformation due to lack of space and instead directly present its final result, the starting point for our method:

Definition 1. *A* PLTL-specification S *is a quadruple* (Σ, I, T, G) *such that*

- Σ *is a finite propositional signature,*
- I *is a set of* initial *clauses* C_i *(over the signature* Σ*),*
- T *is a set of* step *clauses* $C_t \vee D'_t$ *(over the joint signature* $\Sigma \cup \Sigma'$*),*
- G *is a set of* goal *clauses* C_g *(over the signature* Σ*).*

The initial and step clauses match their counterparts from [6] in the obvious way. Our goal clauses are a generalization of a single unconditional *sometimes* clause that can be obtained using the transformations described in [4]. The whole specification represents the PLTL formula:

$$\left(\bigwedge C_i\right) \wedge \Box \left(\bigwedge (C_t \vee \bigcirc D_t)\right) \wedge \Box\Diamond \left(\bigwedge C_g\right).$$

Example 1. We will be using the valid PLTL formula $\Box((a \to b) \to \bigcirc b) \to \Diamond\Box(a \vee b)$ as a running example that will guide us through the whole theorem proving process presented in this paper. By negating the formula and performing standard transformations we obtain $\Box(a \vee \bigcirc b) \wedge \Box(\neg b \vee \bigcirc b) \wedge \Box\Diamond(\neg a \wedge \neg b)$, which gives us the following PLTL-specification $S = (\{a, b\}, \emptyset, \{a \vee b', \neg b \vee b'\}, \{\neg a, \neg b\})$.

It is a known fact that when considering satisfiability of PLTL formulas attention can be restricted to *ultimately periodic* [18] interpretations. These start with a finite sequence of worlds and then repeat another finite sequence of worlds forever. This observation, which is also one of the key ingredients of our approach, motivates the following definition.

Definition 2. *Let $K \in \mathbb{N}$, and $L \in \mathbb{N}^+$ be given. A PLTL-interpretation $(V_i)_{i\in\mathbb{N}}$ is a* (K, L)-model *of $S = (\Sigma, I, T, G)$ if*

1. *for every $C \in I$, $V_0 \models C$,*
2. *for every $i \in \mathbb{N}$ and every $C \in T$, $[V_i, V_{i+1}] \models C$,*
3. *for every $i \in \mathbb{N}$ and every $C \in G$, $V_{(K+i\cdot L)} \models C$.*

A PLTL-specification is satisfiable *if it has a (K, L)-model for some K and L.*

Note that the eventuality represented by the goal clauses of S is satisfied infinitely often as the standard PLTL semantics dictates. Moreover, we keep track of the worlds where this is bound to happen by requiring they form an arithmetic progression with K as the initial term and L the common difference. This additional requirement doesn't change the notion of satisfiability thanks to the observation mentioned above. We will call the pair (K, L) the *rank* of a model.

3 Labelled Clauses

Recall that we defined a PLTL-interpretation as an infinite sequence of propositional valuations over the finite signature Σ. Alternatively though, it can be viewed as a *single* propositional valuation over the *infinite* signature $\Sigma^* = \bigcup_{i\in\mathbb{N}} \Sigma^{(i)}$. We simply index the signature symbols by the time moments to obtain this isomorphic representation. If we now examine Definition 2 of (K, L)-models from this perspective, we can reveal a simple (though at first sight not very useful) reduction of satisfiability in a (K, L)-model to propositional satisfiability of a potentially infinite set of clauses over Σ^*. For a specification $S = (\Sigma, I, T, G)$ this clause set will consist of copies of the clauses from I, T, and G that are "shifted in time" to proper positions, such that the whole set is (propositionally) satisfiable if and only if S has a (K, L)-model. Formally, the set is the union of $\{C^{(0)} \mid C \in I\}$, $\{C^{(i)} \mid C \in T, i \in \mathbb{N}\}$, and $\{C^{(K+i\cdot L)} \mid C \in G, i \in \mathbb{N}\}$. See Fig. 1 for the intuition behind this idea.

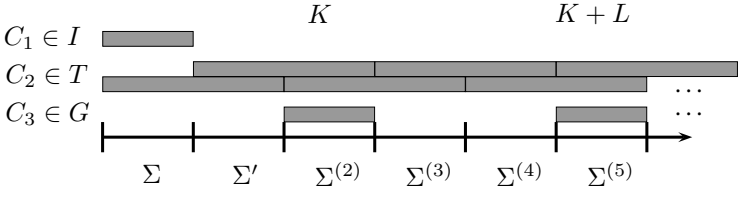


Fig. 1. Schematic presentation of the potentially infinite set of clauses that is satisfiable iff a PLTL-specification $S = (\Sigma, I, T, G)$ has a model of rank $(2, 3)$

In order to make use of the above described reduction we need to show how to solve for infinitely many values of K and L the propositional satisfiability problem consisting of infinitely many clauses. We do this by assigning *labels* to

the clauses of S such that a labelled clause represents up to infinitely many standard clauses over Σ^*. Then an inference performed between labelled clauses corresponds to infinitely many inferences on the level of Σ^*. This is not dissimilar to the idea of "lifting" from first-order theorem proving where clauses with variables represent up to infinitely many ground instances. Here, however, we deal with the additional dimension of performing infinitely many proof tasks "on the ground level" in parallel, one for each rank (K, L).

Formally, a *label* is a pair (b, k) where b is either $*$ or 0, and k is either $*$ or an element of $\mathbb{N}$. A *labelled clause* is a pair $(b, k) \,||\, C$ consisting of a label and a (standard) clause over $\Sigma \cup \Sigma'$. Given a PLTL-specification $S = (\Sigma, I, T, G)$, the *initial labelled clause set* N_S for S is defined to contain

- labelled clauses of the form $(0, *) \,||\, C$ for every $C \in I$,
- labelled clauses of the form $(*, *) \,||\, C$ for every $C \in T$, and
- labelled clauses of the form $(*, 0) \,||\, C$ for every $C \in G$.

We can think of the first label component b as relating the clause to the beginning of time, while the second component relates the clause to the indices of the form $K + i \cdot L$, where the goal should be satisfied. In both cases, $*$ stands for a "don't care" value, thus, e.g., the label $(*, *)$ marks clauses that occupy every possible index. It turns out that during inferences we also need to talk about clauses that reside k steps before indices of the goal. That is why the second label component may assume any value from $\mathbb{N}$. The semantics of labels is given via a map to world indices.

Let (K, L) be a rank. We define a set $R_{(K,L)}(b, k)$ of indices *represented* by the label (b, k) as the set of all $t \in \mathbb{N}$ such that

$$[b \neq * \to t = 0] \wedge [k \neq * \to \exists s \in \mathbb{N}. t + k = K + s \cdot L] \ .$$

Observe that while $R_{(K,L)}(0, k) \subseteq \{0\}$, the sets $R_{(K,L)}(*, k)$ are always infinite, and for $k \in \mathbb{N}$ constitute a range of an arithmetic progression with difference L. Now a standard clause of the form $C^{(t)}$ is said to be *represented by the labelled clause* $(b, k) \,||\, C$ *in* (K, L) if $t \in R_{(K,L)}(b, k)$. We denote the set of all standard clauses represented in (K, L) by the labelled clauses N by the symbol $N_{(K,L)}$.

$$N_{(K,L)} = \{C^{(t)} \mid \text{clause } (b, k) \,||\, C \in N \text{ and } t \in R_{(K,L)}(b, k)\} \ .$$

Example 2. Our example specification $S = (\{a, b\}, \emptyset, \{a \vee b', \neg b \vee b'\}, \{\neg a, \neg b\})$ contains among others the single literal goal clause $\neg a$. In the initial labelled clause set N_S this goal clause becomes $(*, 0) \,||\, \neg a$. If we now, for example, fix the same rank $(2, 3)$ as in Fig. 1, our labelled clause will in that rank represent all the standard clauses $(\neg a)^{(t)}$ with $t \in R_{(2,3)}(*, 0) = \{2, 5, 8, \dots\}$.

We summarize the main message of this section in the next lemma. Its proof follows from the definitions and ideas already given.

Lemma 1. *Let a rank (K, L) and a PLTL-specification S be given and let N_S be the initial labelled clause set for S. Then the set $(N_S)_{(K,L)}$ is satisfiable if and only if S has a (K, L)-model.*

4 The Labelled Superposition Calculus *LPSup*

In this section we present our calculus for labelled clauses *LPSup*. We lift the ordered resolution calculus of [2], which we call *PSup* for Propositional Superposition, and transfer to *LPSup* its valuable properties, including the ordering restrictions of inferences. For that purpose, we parameterize *LPSup* by a total ordering $<$ on the symbols of the signature Σ, which we implicitly extend to indexed signatures by first comparing the indices and only then the actual symbols. This means that $p^{(i)} < q^{(j)}$ if and only if $i < j$, or $i = j$ and $p < q$.[1] We then use the standard extension of this ordering to compare literals in clauses.

Before we proceed to the actual presentation of the calculus, we need to define how labels are updated by inferences. Two labelled clauses should only interact with each other when they actually represent standard clauses that interact on the ground level. Moreover, the resulting labelled clause should represent exactly all the possible results of the interactions on the ground level. We define the *merge* of two labels (b_1, k_1) and (b_2, k_2) as the label (b, k) such that

- if $b_1 = b_2 = *$ then $b = *$, otherwise $b = 0$,
- if $k_1 = *$ then $k = k_2$; if $k_2 = *$ then $k = k_1$; if $k_1 = k_2 \neq *$ then $k = k_1 = k_2$.

In the case when $k_1, k_2 \in \mathbb{N}$ and $k_1 \neq k_2$, the merge operation is *undefined*. The idea is that the merged label represents the intersection of the sets of indices represented by the arguments.

The calculus *LPSup* consists of the inference rules Ordered Resolution, Ordered Factoring, Temporal Shift, and Leap. They operate on a clause set N, an initial labelled clause set of a given PLTL-specification. While Ordered Resolution and Ordered Factoring constitute the labelled analogue of inferences of *PSup*, Temporal Shift and Leap are "structural" in nature, as they only modify the syntactic format, but the underlying represented set of standard clauses remains the same.

(i) *Ordered Resolution*

$$\mathcal{I}\frac{(b_1, k_1) \,||\, C \vee L \quad (b_2, k_2) \,||\, D \vee \bar{L}}{(b, k) \,||\, C \vee D}$$

where literal L is maximal in C, its complement $\bar{L}$ is maximal in D, and the merge of labels (b_1, k_1) and (b_2, k_2) is defined and equal to (b, k),

(ii) *Ordered Factoring*

$$\mathcal{I}\frac{(b, k) \,||\, C \vee A \vee A}{(b, k) \,||\, C \vee A}$$

where A is an atom maximal in C,

[1] In the case of labelled clauses this amounts to saying that the symbols of Σ' are considered larger than those of Σ. Our definition, however, also makes sense over the infinite signature Σ^* and it is this particular ordering that restricts the inferences on the level of standard clauses.

(iii) *Temporal Shift*

$$\mathcal{I}\,\frac{(*,k)\,||\,C}{(*,k')\,||\,(C)'}$$

where C is a clause over Σ only, and $k = k' = *$ or $k \in \mathbb{N}$ and $k' = k + 1$,

(iv) *Leap*

$$\mathcal{I}\,\frac{\{(b, u + i \cdot v)\,||\,C\}_{i\in\mathbb{N}} \text{ derivable from } N}{(b, u - v)\,||\,C}$$

where $u \geq v > 0$ are integers and C is an arbitrary standard clause.

Further explanation is needed for the inference rule Leap. In its present form it requires an infinite number of premises, one for each $i \in \mathbb{N}$, and thus cannot, strictly speaking, become applicable in any finite derivation. Here it is only a mathematical abstraction. In the next section we show how to effectively generate and finitely represent infinite sets of labelled clauses from which it will follow that Leap is, in fact, effective.

Going back to the other inferences note that the merge operation on labels ensures that the conclusion of Ordered Resolution represents exactly all the conclusions of the standard ordered resolution inferences between the standard clauses represented by the premises. Ordered Factoring carries over from *PSup* in a similar fashion.[2] The Temporal Shift operates only on clauses over the signature Σ. We will from now on call such clauses *simple*. Notice that the restriction to simple clauses is essential as it keeps the symbols of the conclusion to stay within $\Sigma \cup \Sigma'$.

Example 3. The initial labelled clause set N_S of our running example contains among others also clauses $(*, *)\,||\,a \vee b'$ and $(*, 0)\,||\,\neg b$. We can apply Temporal Shift to the second to obtain $(*, 1)\,||\,\neg b'$. Now b' is the only literal over Σ' in the first clauses and therefore maximal. So the first clause and the newly derived one can participate in Ordered Resolution inference with conclusion $(*, 1)\,||\,a$.

Although the rules Temporal Shift and Leap derive new labelled clauses, the represented sets of standard clauses remain the same in any rank (K, L). This is easy to see for Temporal Shift, but a little bit more involved for Leap, where it relies on the periodicity of (K, L)-models. The overall soundness of *LPSup* is established by relating it to the same property of the standard calculus *PSup*.

Theorem 1 (Soundness of *LPSup*). *Let N_S be the initial labelled clause set for a PLTL-specification S, and $(b, k)\,||\,C$ a labelled clause derivable from N_S by LPSup. Then for any rank (K, L) and any $t \in R_{(K,L)}(b, k)$ the standard clause $C^{(t)}$ is derivable from $(N_S)_{(K,L)}$ by PSup.*

If an empty labelled clause $(b, k)\,||\,\bot$ is derivable from N_S by LPSup, such that $R_{(K,L)}(b, k) \neq \emptyset$, then S doesn't have a (K, L)-model.

[2] Here we present the rule in a form as close as possible to the one in [2]. In practical implementation, however, it is reasonable to remove duplicate literals as soon as they occur without regard to ordering restrictions.

Notice that in *LPSup* the fact that an empty labelled clause $(b, k) \,||\, \bot$ is derived does not necessarily mean that the whole clause set is unsatisfiable. It only rules out those (K, L)-models for which $R_{(K,L)}(b, k)$ is non-empty. This motivates the following definition.

Definition 3. *An empty labelled clause* $(b, k) \,||\, \bot$ *is called* conditional *if* $b = 0$ *and* $k \in \mathbb{N}$*, and* unconditional *otherwise. We say that a set of labelled clauses* N *is* contradictory *if it contains an unconditional empty clause, or* $(0, k) \,||\, \bot$ *is in* N *for every* $k \in \mathbb{N}$*.*

In Sect. 6 we demonstrate that a (K, L)-model can be found for any non-contradictory set of labelled clauses that is saturated by *LPSup*.

To complete the picture of *LPSup* we move on to mention reduction rules. As we discuss in detail in Sect. 6, these are justified by the abstract redundancy notion [2] which our calculus inherits from *PSup*. Thus the following are only examples and other reductions can be developed and used as long as they satisfy the criteria of abstract redundancy.

Tautology Deletion allows us to remove from the search any labelled clause the standard part of which contains both a literal and its complement. Another useful reduction is *Subsumption*.[3]

$$\mathcal{R}\frac{(b_1, k_1) \,||\, C \quad (b_2, k_2) \,||\, D}{(b_1, k_1) \,||\, C}$$

where C is a sub-multiset of D and the merge of labels (b_1, k_1) and (b_2, k_2) is defined and equal to (b_2, k_2).

5 Decision Procedure

In this section we explain how to turn the calculus *LPSup* into an effective decision procedure for PLTL. First, we have a look at termination.

Example 4. We have already derived the labelled clause $(*, 1) \,||\, \neg b'$ from our set N_S of initial clauses for S by Temporal Shift. Ordered Resolution between this clause and the clause $(*, *) \,||\, \neg b \vee b'$ yields $(*, 1) \,||\, \neg b$ to which Temporal Shift is again applicable, giving us $(*, 2) \,||\, \neg b'$. We see that the clause we started with differs from the last one only in the label where the k-component got increased by one. The whole sequence of inferences can now be repeated, allowing us to eventually derive labelled clauses $(*, k) \,||\, \neg b$ and $(*, k) \,||\, \neg b'$ for any $k \in \mathbb{N}^+$.

The example demonstrates how the Temporal Shift inference may cause non-termination when the k-component of the generated labelled clauses increases one by one. It also suggests, however, that from a certain point the derived clauses don't add any new information and the inferences essentially repeat in

[3] We use the letter $\mathcal{I}$ and $\mathcal{R}$ to distinguish between *inference rules*, whose premises are kept after the conclusion has been added to the given set of clauses, and *reduction rules*, whose premises are replaced by the conclusion.

cycles. Detecting these repetitions and finitely representing the resulting infinite clause sets is the key idea for obtaining a termination result for our calculus.

Given a set of labelled clauses N, it is convenient to think of N as being separated into *layers*, sets of clauses with the same value of their labels' second component k. This way we obtain the $*$-layer of clauses with the label of the form $(b, *)$ for $b \in \{*, 0\}$, and similarly layers indexed by $k \in \mathbb{N}$. The following list of observations forms the basis of our strategy for saturating clause sets by *LPSup*.

(1) In an initial labelled clause set only the $*$-layer and 0-layer are non-empty.
(2) If all premises of Ordered Resolution, Factoring or Temporal Shift inference belong to the $*$-layer, so does the conclusion of the respective inference.
(3) If a premise of Ordered Resolution or Factoring inference belongs to the k-layer for $k \in \mathbb{N}$, so does the inference's conclusion.
(4) If a premise of Temporal Shift belongs to the k-layer for $k \in \mathbb{N}$, the inference's conclusion belongs to the layer with index $(k + 1)$.
(5) The number of clauses in each layer is bounded by a constant depending only on the size of the signature.

We are ready to describe what we call *layer-by-layer* saturation of an initial labelled clause set. During this process we don't yet consider the Leap inference, which will be incorporated later. It follows from our observations that the $*$-layer can always be finitely saturated. We then perform all the remaining Ordered Resolution and Factoring inferences (together with possible reductions) to saturate the 0-layer, again in a finite number of steps. After that we exhaustively apply the Temporal Shift rule to populate the 1-layer and again saturate this layer by Ordered Resolution and Factoring. This process can be repeated in the described fashion to saturate layers of increasing indices. It is important that the new clauses of the higher layers can never influence (by participating on inferences or reductions) clauses in the lower, already saturated, layers. Eventually, thanks to point (5) above, we will encounter a layer we have seen before and then we stop. More precisely, in a finite number of steps we are bound to obtain a set of labelled clauses N such that there are integers $o \in \mathbb{N}$ and $p \in \mathbb{N}^+$ and

- the o-layer of N is equal to the $(o + p)$-layer of N (up to reindexing[4]),
- the clause set is saturated by *LPSup* (without Leap), except, possibly, for Temporal Shift inferences with premise in layer $(o + p)$,
- the layers with index larger than $(o + p)$ are empty.

Now we need a final observation to finish the argument. The applicability of Ordered Resolution, Factoring and Temporal Shift (as well as that of the reductions of *LPSup*) is "invariant under the move from one layer to another". In other words, exactly the same (up to reindexing) inferences (and reductions) that have been performed to obtain, e.g., the saturated layer of index $(o + 1)$, can now be repeated to obtain the saturated layer of index $(o + p + 1)$. We can therefore stop the saturation process here and define:

[4] Meaning the first mentioned set would be identical to the second if we changed the second label component of all its clauses from o to $(o + p)$.

Definition 4. *Let N be a clause set obtained by layer-by-layer saturation as described above. We call the numbers o and p the* offset *and* period *of N, respectively. The* infinite extension *of such N is the only set of labelled clauses N^* for which $N \subseteq N^*$ and such that for every $i \in \mathbb{N}$ the $(o+i)$-layer of N^* is equal to the $(o+i \bmod p)$-layer of N (up to reindexing).*

The infinite extension of N is completely saturated by *LPSup* (without Leap).

Example 5. In our running example, the $*$-layer and 0-layer are already saturated. The next layers we obtain are

$$\{(*,1) \,||\, \neg a', (*,1) \,||\, \neg b', (*,1) \,||\, a, (*,1) \,||\, \neg b\} \ , \tag{1}$$

$$\{(*,2) \,||\, a', (*,2) \,||\, \neg b', (*,2) \,||\, a, (*,2) \,||\, \neg b\} \tag{2}$$

As the 3-layer is then equal to the previous (up to reindexing), layer-by-layer saturation terminates with offset 2 and period 1.

In layer-by-layer saturation we always give priority to Ordered Resolution and Factoring inferences, and only when these are no longer applicable in the current clause set, we perform all the pending Temporal Shift inferences, and possibly repeat. Similarly, the *overall saturation procedure* which we present next combines layer-by-layer saturation phases with an exhaustive application of the Leap inference:

1. Set N_1 to the initial labelled clause set N_S of a given PLTL-specification S.
2. Set N_2 to the layer-by-layer saturation on N_1.
3. If the clause set N_2^* is contradictory, stop and report UNSAT.
4. Set N_3 to be the set N_2 enriched by all the possible conclusions of Leap inference with premises in N_2^*, possibly reduced.
5. If $N_3 = N_2$ stop and report SAT, else go back to step 2 resetting $N_1 := N_3$.

Note that if we go to line 2 for the second time, N_1 is no longer an initial labelled clause set. Although we didn't discuss it previously, it is straightforward to perform layer-by-layer saturation of any finitely represented clause set.

On lines 3 and 4 we refer to the infinite extension N_2^*. It actually means that we operate with the layer-by-layer saturation N_2 together with offset o and period p. Now N_2^* is bound to be contradictory if and only if N_2 contains an unconditional empty clause or $(0,k) \,||\, \bot$ is in N_2 for every $0 \leq k < o+p$. Similarly, a labelled clause $(b,j) \,||\, C$ with $j < o$[5] can be derived by Leap inference with premises in N_2^* if and only if there is a clause $(b,i) \,||\, C$ in N_2 such that $o \leq i < o+p$ and p divides $i-j$.

Finally note that while the values of offset and period associated with N_2 may change from one repetition to another, their sum is each time bounded by the same constant depending only on the size of the signature, namely the number of different possible layers (up to reindexing). Moreover, thanks to the fact that we only work with a fixed finite signature, there is also a bound on the number of non-trivial additions to the individual layers on line 4. These together ensure that the procedure always terminates.

[5] Leap conclusion with $j \geq o$ is always redundant.

Example 6. In our example, the infinite extension of the layer-by-layer saturation contains the premises $\{(*, 1+i) \,||\, a\}_{i \in \mathbb{N}}$ of a Leap inference with conclusion $(*, 0) \,||\, a$. This clause together with the already present $(*, 0) \,||\, \neg a$ gives us the empty clause $(*, 0) \,||\, \bot$ by Ordered Resolution, which eventually terminates the overall procedure, because the empty clause is unconditional and therefore the overall set becomes contradictory.

6 Redundancy, Completeness and Model Building

The calculus *LPSup* comes with an abstract notion of redundancy in the spirit of [2]. Also here one can recognize the idea of "lifting", which relates the standard level of *PSup* to the level of labelled clauses. Recall that a standard clause C is called redundant with respect to a set of standard clauses N if there are clauses $C_1, \ldots, C_n \in N$ such that for every $i = 1, \ldots n$, $C_i < C$, and $C_1, \ldots, C_n \models C$. On the level of labelled clause we define:

Definition 5. *A labelled clause* $(b, k) \,||\, C$ *is* redundant *with respect to a set of labelled clauses* N*, if for any rank* (K, L) *every standard clause represented by* $(b, k) \,||\, C$ *in* (K, L) *is redundant w.r.t.* $N_{(K,L)}$*.*

A set of labelled clauses N *is* saturated up to redundancy *with respect to LPSup, if for every inference from* N *such that its premises are not redundant w.r.t.* N*, the conclusion is either redundant w.r.t.* N *or contained in* N*.*

Note that the reductions of *LPSup* described in Sect. 4 are instances of redundancy elimination. This is easy to see for Tautology deletion, and follows from the semantics of the merge operation on labels for the Subsumption reduction. It is important to note that these are just examples and further reductions can be developed and used. As long as they fit into the framework prescribed by Definition 5, they are guaranteed to preserve completeness and the underlying proof need not be changed.

Our main theorem relates completeness of *LPSup* to the same property of the underlying calculus *PSup* via the notion of redundancy.

Theorem 2 (Completeness of *LPSup*). *Let* N *be a labelled clause set saturated in a layer-by-layer fashion with offset* o *and period* p *and let* N^**, the infinite extension of* N*, be a non-contradictory set of labelled clause saturated up to redundancy w.r.t. LPSup. We set* K *to be the smallest number from* $\mathbb{N}$ *such that* $(0, K) \,||\, \bot$ *is not in* N^* *(note that* N^* *is non-contradictory), and further set* L *to the smallest positive multiple of* p *that is not smaller than* o*. Then the set* $N^*_{(K,L)}$ *does not contain the (standard) empty clause and is saturated up to redundancy w.r.t. PSup.*

Recall the overall saturation procedure of the previous section. Its input is a PLTL-specification which is immediately transformed into the initial labelled clause set. If the procedure reports UNSAT, we know the input is unsatisfiable, because we derived (using a sound calculus) a contradictory set of labelled clauses, which rules out any (K, L)-model. If, on the other hand, the procedure

reports SAT, we may apply Theorem 2 together with completeness of *PSup* to conclude that the set $N^*_{(K,L)}$ is satisfiable, and, therefore, the specification we started with has a (K, L)-model. Thus the overall saturation procedure decides satisfiability of PLTL-specifications.

We close this section by commenting on the possibility of using our method to provide counterexamples to non-valid PLTL formulas. Due to space restrictions, we cannot describe the method in full detail, but to those familiar with the model construction for classical logic based on *PSup* [2], it should be clear that with Theorem 2 proven, we are practically done.

Given a non-contradictory set of labelled clauses N^* that is saturated up to redundancy w.r.t. *LPSup*, we pick (K, L) as described in Theorem 2 and generate the standard clauses of $N^*_{(K,L)}$ one by one with increasing $<$. We apply classical model construction to these clauses to gradually build a (partial) valuation over $\Sigma^* = \bigcup_{i\in\mathbb{N}} \Sigma^{(i)}$, which, as we know, corresponds in the obvious way to a (K, L)-model $(V_i)_{i\in\mathbb{N}}$. We can stop the generation as soon as a particular (already completed) valuation repeats (i.e. $V_i = V_{i-j}$ for some $j \in \mathbb{N}^+$) and the goal has already been reached (i.e. $i > K$). An ultimately periodic model is then output as a result.

7 Final Discussion and Experiments

We now compare our calculus to Clausal Temporal Resolution [6]. Older resolution based approaches to PLTL are [3, 21], but they don't seem to be used or developed any further nowadays. Besides resolution there are approaches to PLTL satisfiability based on tableaux deduction [22, 17], and on automata theory [16]. These seem to be less related and we don't discuss them here further.

It can be shown that operationally there is a close connection between *LPSup* and the Clausal Temporal Resolution (CTR) of [6]. From this perspective, our formalism of labelled clauses can be seen as a new way to derive completeness of CTR that justifies the use of ordering restrictions and redundancy elimination. This has not been achieved yet in full by previous work: [9] contains a proof theoretic argument, but only for the use of ordering restrictions, [11] sketches the idea how to justify tautology removal and subsumption, but not the general redundancy notion in the style of [2] that we provide.

Moreover, there is also a correspondence between our layer-by-layer saturation followed by the application of the Leap inference and the BFS-Loop search of CTR as described in [7, 13]. Apart from being interesting in its own right, this view sheds new light on explaining BFS-Loop search, as it gives meaning to the intermediate clauses generated in the process, and we thus don't need to take the detour through the DNF representation of [5]. Even here, the idea of labels clearly separates logical content of the clauses from the meta-logical one (c.f. the ad hoc marker literal of [7]).

Despite these similarities between *LPSup* and CTR, the calculi are by no means identical. As discussed before, a temporal model can be extracted in a straightforward way from a satisfiable set of labelled clauses saturated by *LPSup*.

This doesn't hold for CTR, where a more complex approach that simulates the model construction of [2] only locally needs to be applied [14]. In particular, because saturation by CTR doesn't give the model building procedure any guidance as to where to look for the goal, in each considered world all the possible orderings on the signature (in the worse case) need to be tried out in a fair way to make sure a goal world is eventually reached. As each change of the ordering calls for a subsequent resaturation of the clause set in question (so that the local model construction still works), it obviously diminishes the positive effect orderings in general have on reducing the search space.

Finally note that since we eventually rely on propositional superposition, we can also take into account the explicit use of partial models to further guide the search for a proof or saturation. The idea is to build a partial model based on the ordering on propositional literals. Then it can be shown that resolution can be restricted to premises where one is false and the other true in the partial model [1]. This superposition approach on propositional clauses is closely related to the state of the art CDCL calculus (see, e.g. [23]) for propositional logic. The missing bit is to "lift" this setting to our labelled clauses. This will be one direction for future research.

We implemented a simple prototype of both *LPSup* and CTR (with BFS loop-search in the style of [7]), in order to compare the two calculi on non-trivial examples. In this section we briefly report on our experiment. The prototype, written in SWI-Prolog, is available along with the test examples at [19].

For the experiment we choose two formula families described in [10], which we call $\mathcal{C}_n^1$ and $\mathcal{C}_n^2$. In addition, we also tested the calculi on formulas from two families specifically constructed to highlight the respective weaknesses of *LPSup* and CTR. These we call the implicit and explicit cycles problems, respectively, and denote them by $\mathcal{I}_{(l_1+\cdots+l_k)}$ and $\mathcal{E}_{(l_1+\cdots+l_k)}$. The problems are parameterized by the sequence of numbers $l_1, \ldots, l_k$, which denote the cycles' lengths.

Table 1. Results of comparing our implementations of *LPSup* and CTR with TRP++

Problem	Size	*LPSup*-Prolog			CTR-Prolog			TRP++	
		Cl-gen	Lits-gen	Cl-subs	Cl-gen	Lits-gen	Cl-subs	Cl-gen	Cl-subs
$\mathcal{C}_{10}^1$	56	53	202	100	174	576	110	363	300
$\mathcal{C}_{15}^1$	81	78	377	145	334	1161	240	688	595
$\mathcal{C}_{20}^1$	106	103	602	190	544	1946	420	1113	990
$\mathcal{C}_3^2$	22	442	1376	324	984	3972	909	1146	968
$\mathcal{C}_4^2$	30	1937	7649	1612	5298	26086	5047	3560	3053
$\mathcal{C}_5^2$	38	6287	28576	5635	18724	102704	18134	7925	6922
$\mathcal{I}_{(3+5)}$	62	406	1563	368	203	1022	194	86	160
$\mathcal{I}_{(3+5+8)}$	253	8010	42024	7356	1087	7613	1145	390	745
$\mathcal{E}_{(2+3)}$	8	23	25	4	131	424	78	177	77
$\mathcal{E}_{(2+3+4)}$	13	52	55	6	1061	4490	595	1597	627

Table 7 summarizes the results of our experiments. For each problem and for both calculi we report the number of clauses in the input, the number of derived[6] clauses and literals, and the number of subsumed clauses. For comparison, we also include in the last two columns clause data obtained by running the temporal prover TRP++ [8][7], which also implements the CTR calculus, to provide evidence that our experimental results are not biased. We decided not to report on running times as our aim here is to compare the calculi rather than the implementations. The number of generated clauses (literals) should provide a good measure on the amount of data to be processed by any prover, which is, moreover, independent on the choice programming language or the use of particular data structures.

As we can see, *LPSup* needs to generate consistently less clauses to draw its conclusion for both $\mathcal{C}_n^1$ and $\mathcal{C}_n^2$. It only behaves worse on the implicit cycles examples $\mathcal{I}$, which are constructed in such a way that the number of iterations of the layer-by-layer saturation is much higher for *LPSup* than CTR. The examples $\mathcal{E}$, on the other hand, present much more work for CTR, where the equivalent of Temporal Shift rule causes the clause set to "blow-up". All in all, *LPSup* seems to come considerably better off out of our experiments.

8 Conclusion

We applied the ideas of labelled superposition to develop a new decision procedure for propositional linear temporal logic. On the presentation level, it replaces the complex temporal resolution rule from the previously proposed calculus by a simple check for repetition in the derived clause set and a subsequent inference. Its unique treatment of goal clauses enables straightforward partial model building of satisfiable clause sets which could potentially be used to further restrict inferences. Moreover, the experimental comparison to previous work suggests that the new calculus typically explores smaller search spaces to derive its conclusion. Development of an optimized implementation, to be tested on a set of representative benchmarks, will be part of our future work.

References

[1] Bachmair, L., Ganzinger, H.: On Restrictions of Ordered Paramodulation with Simplification. In: Stickel, M.E. (ed.) CADE 1990. LNCS, vol. 449, pp. 427–441. Springer, Heidelberg (1990)

[2] Bachmair, L., Ganzinger, H.: Resolution theorem proving. In: Robinson, J.A., Voronkov, A. (eds.) Handbook of Automated Reasoning, pp. 19–99. Elsevier and MIT Press (2001)

[6] This covers all the resolvents, plus the clauses derived by non-trivial Leap inference. (Leap conclusions subsumed by other clauses are not generated at all.)

[7] We used version 2.1 available at `http://www.csc.liv.ac.uk/~konev/software/trp++/`.

[3] Cavalli, A., del Cerro, L.: A Decision Method for Linear Temporal Logic. In: Shostak, R.E. (ed.) CADE 1984. LNCS, vol. 170, pp. 113–127. Springer, Heidelberg (1984)
[4] Degtyarev, A., Fisher, M., Konev, B.: A Simplified Clausal Resolution Procedure for Propositional Linear-Time Temporal Logic. In: Egly, U., Fermüller, C. (eds.) TABLEAUX 2002. LNCS (LNAI), vol. 2381, pp. 85–99. Springer, Heidelberg (2002)
[5] Dixon, C.: Search Strategies for Resolution in Temporal Logics. In: McRobbie, M.A., Slaney, J.K. (eds.) CADE 1996. LNCS, vol. 1104, pp. 673–687. Springer, Heidelberg (1996)
[6] Fisher, M., Dixon, C., Peim, M.: Clausal temporal resolution. ACM Trans. Comput. Logic 2, 12–56 (2001)
[7] Fernández Gago, M.C., Fisher, M., Dixon, C.: Algorithms for Guiding Clausal Temporal Resolution. In: Jarke, M., Koehler, J., Lakemeyer, G. (eds.) KI 2002. LNCS (LNAI), vol. 2479, pp. 235–252. Springer, Heidelberg (2002)
[8] Hustadt, U., Konev, B.: TRP++ 2.0: A Temporal Resolution Prover. In: Baader, F. (ed.) CADE-19. LNCS (LNAI), vol. 2741, pp. 274–278. Springer, Heidelberg (2003)
[9] Hustadt, U., Konev, B., Schmidt, R.A.: Deciding Monodic Fragments by Temporal Resolution. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 204–218. Springer, Heidelberg (2005)
[10] Hustadt, U., Schmidt, R.: Scientific benchmarking with temporal logic decision procedures. In: KR 2002, pp. 533–546. Morgan Kaufmann (2002)
[11] Konev, B., Degtyarev, A., Dixon, C., Fisher, M., Hustadt, U.: Mechanising first-order temporal resolution. Inf. Comput. 199, 55–86 (2005)
[12] Lev-Ami, T., Weidenbach, C., Reps, T., Sagiv, M.: Labelled Clauses. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 311–327. Springer, Heidelberg (2007)
[13] Ludwig, M., Hustadt, U.: Fair Derivations in Monodic Temporal Reasoning. In: Schmidt, R.A. (ed.) CADE-22. LNCS, vol. 5663, pp. 261–276. Springer, Heidelberg (2009)
[14] Ludwig, M., Hustadt, U.: Resolution-based model construction for PLTL. In: TIME 2009, pp. 73–80. IEEE Computer Society (2009)
[15] Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science, pp. 46–57. IEEE (1977)
[16] Rozier, K.Y., Vardi, M.Y.: LTL Satisfiability Checking. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 149–167. Springer, Heidelberg (2007)
[17] Schwendimann, S.: A New One-Pass Tableau Calculus for PLTL. In: de Swart, H. (ed.) TABLEAUX 1998. LNCS (LNAI), vol. 1397, pp. 277–291. Springer, Heidelberg (1998)
[18] Sistla, A.P., Clarke, E.M.: The complexity of propositional linear temporal logics. J. ACM 32, 733–749 (1985)
[19] Suda, M., Weidenbach, C.: Prototype implementation of LPSup. (2011), http://www.mpi-inf.mpg.de/~suda/supLTL.html
[20] Suda, M., Weidenbach, C.: Labelled Superposition for PLTL. Research Report MPI-I-2012-RG1-001, Max-Planck-Institut für Informatik, Saarbrücken (2012)
[21] Venkatesh, G.: A Decision Method for Temporal Logic Based on Resolution. In: Maheshwari, S.N. (ed.) FSTTCS 1985. LNCS, vol. 206, pp. 272–289. Springer, Heidelberg (1985)
[22] Wolper, P.: The tableau method for temporal logic: An overview. Logique et Analyse 28, 119–136 (1985)
[23] Zhang, L., Madigan, C.F., Moskewicz, M.W., Malik, S.: Efficient conflict driven learning in boolean satisfiability solver. In: ICCAD, pp. 279–285 (2001)

The TPTP Typed First-Order Form with Arithmetic

Geoff Sutcliffe[1], Stephan Schulz[2], Koen Claessen[3], and Peter Baumgartner[4]

[1] University of Miami, USA
[2] Technische Universität München, Germany
[3] Chalmers University, Sweden
[4] NICTA and ANU, Australia

Abstract. The TPTP World is a well established infrastructure supporting research, development, and deployment of Automated Theorem Proving systems. Recently, the TPTP World has been extended to include a typed first-order logic, which in turn has enabled the integration of arithmetic. This paper describes these developments.

1 Motivation and History

The TPTP World [32] is a well established infrastructure that supports research, development, and deployment of Automated Theorem Proving (ATP) systems. The TPTP World is based on the Thousands of Problems for Theorem Provers (TPTP) problem library [30], and includes the TPTP language, the SZS ontologies, the Thousands of Solutions from Theorem Provers (TSTP) solution library, various tools associated with the libraries, and the CADE ATP System Competition (CASC). This infrastructure has been central to the progress that has been made in the development of high performance first-order ATP systems – most state of the art systems natively read the TPTP language, many produce proofs or models in the TSTP format, much testing and development is done using the TPTP problem library, and CASC is an annual focal point where developers meet to discuss new ideas and advances in ATP techniques.

Originally the TPTP supported only first-order problems in clause normal form (CNF). Over the years support for the full first-order form (FOF) and typed higher-order form (THF) have been added. Recently the simply typed first-order form (TFF) has been added. TFF has in turn been used as the basis for supporting arithmetic. Problems that use these new features have been added to the TPTP problem library, and ATP systems that can solve these problems have been developed. This paper describes the key steps of these developments.

While the development of the TPTP World for typed first-order logic is new, several similar logics have been described previously, e.g., [36,26,13]. However, there are no contemporary ATP systems that implement those logics. There is active related research in the SMT community, which started in 2003 [24]. There are high performance systems for the various logics of the SMT-LIB[1], e.g., those

[1] See the SMT-LIB web page `http://combination.cs.uiowa.edu/smtlib/`

N. Bjørner and A. Voronkov (Eds.): LPAR-18, LNCS 7180, pp. 406–419, 2012.

that performed well in the SMT-COMP competitions.[2] While the TFF language was designed independently, there are inevitable parallels between the TFF and SMT languages. The TPTP and SMT languages both fully support a typed first-order logic, and both have specific features for arithmetic theories. Some features of the TPTP TFF language were adopted from SMT, and some differences were motivated by differences between the two communities' idioms (e.g., the TPTP arithmetic includes the Euclidean quotient used in the SMT-LIB `Ints` theory, but also other quotients requested by TPTP users). Salient commonalities and differences between the two languages are evident in this paper. At the level of the TPTP and SMT-LIB problem collections, the problems in SMT-LIB are categorized with respect to their underlying logics and theories (e.g., the admissible quantifier prefixes and the kind of arithmetic used). Those categories and the problems in them typically reflect the capabilities of the available SMT solvers. The TPTP library uses its "specialist problem class" categorization (e.g., the use of equality and the SZS status of problems [29]) only for the analysis of results, and in this way encourages the submission of problems and the development of tools without a specific reasoner or language fragment in mind. There is a growing linkage between the SMT and TPTP worlds, stimulated and made possible by the developments described in this paper. One example was the entry of the SMT-based systems CVC3 and Z3 in the TFA division of CASC-23 [34]. Tools for translating between the TPTP and SMT formats have (or should be by the time this paper is published) been developed – the TPTP2X utility distributed as part of the TPTP is used to translate TPTP TFF problems to SMT2 format for input by CVC3 (see Section 4).

2 The TPTP Typed First-Order Form Language

The design of the TPTP's TFF language was based on consideration of various features of type systems. These fall into four broad categories – the *sorts* (atomic types) that are available, the possibilities for *the types of terms*, the *syntax* of expressions, and the *semantics* of the logic. The decisions made for the first two categories, and the effects of the decisions on ATP systems, are discussed in Section 2.1. The syntax is described in Section 2.2, and semantic issues are discussed in Section 2.3. There were many possibilities for each issue, and the decisions aimed to impose a low initial entry barrier for ATP system developers and users, and to allow for future additions to the language. The initial language is thus known as "TFF0" (much like THF0 [8]).

2.1 Decisions About Types and Terms

The decisions made for TFF0 provide a useful and simple extension of the existing untyped FOF logic. The decisions are:

– TFF0 is a simple many-sorted logic. Sorts are interpreted by non-empty, pairwise disjoint, domains.

[2] `http://www.smtcomp.org/`

- All uninterpreted functions and predicates are monomorphic. Although ad-hoc polymorphism over sorts is conceptually simple, allowing it would require extending the TPTP syntax to provide sort annotations on symbols (as supported in the SMT language version 2).
- Equality is ad-hoc polymorphic over the sorts. An equation between terms that have different sorts is ill-typed.
- Subtyping is not employed. (Subtyping may be added in the future.)

This simple many-sorted type system is an extension of untyped first-order logic. As all symbols are monomorphic, all terms except for variables are automatically typed. For problems without equality, satisfiability of well-typed formulae is not affected by ignoring types. For problems with equality, variables need to carry explicit type information because context does not provide the type information, e.g., the variables in an equation $x = y$.

Many proof calculi generalize directly to the typed case, so that existing techniques and implementations can be carried over without prohibitive effort. In particular, unique most general unifiers and matches still exist, and can be computed by straightforward generalizations of existing algorithms. Standard inferences and simplifications remain correct as long as variable instantiations are type-preserving. Systems based on direct instantiation need to check every variable instantiation. Systems based on unification need to check variable instantiations in equational inferences, e.g., paramodulation and superposition, when paramodulating from (or into, but that is not required in most calculi) a variable term. These checks reject inferences that are allowed when using sort predicates (see Section 2.3), and proof search can be simpler. For finite model finders type information is valuable because it restricts the space of possible models that need to be explored. The domains of some of the sorts can be smaller than the domains of other sorts, leading to possibly more efficient algorithms. While some systems, e.g., Paradox [11], try to derive type information to exploit this, user specified type information can be more precise.

For ATP users, typing leads to much simpler encodings than using type predicates, and the requirement of well-typedness helps to correctly encode problems. A typed language is also necessary for correct encoding of problems with arithmetic.

2.2 Syntax

The TFF0 syntax implements the decisions described in Section 2.1. A new TPTP language variant has been introduced, using `tff` as the language symbol of annotated formulae.[3]

- The sorts `$i` (individuals) and `$o` (booleans) are defined. (Note, as is explained below, `$o` is used only as the result sort in predicate type declarations, i.e., there is no built-in theory of boolean terms.) Other defined sorts are associated with specific theories. In particular, `$int`, `$rat`, and `$real` are defined for interpreted arithmetic – see Section 3.

[3] The BNF is available at `http://www.tptp.org/TPTP/SyntaxBNF.html`

– `$tType` is used to introduce users' sorts, by declaring them to be of the psuedo-sort `$tType`. For example

```
tff(fruit_type,type,                tff(list_type,type,
    fruit: $tType ).                    list: $tType ).
```

  This is the only use of `$tType`. Declaration of users' sorts is not required, i.e., sorts can be introduced on the fly. TFF problems in the TPTP problem library have all sorts declared, so as to provide a typo check (pun intended).
– Every function and predicate symbol has at most one declared *type* that specifies the argument and result sorts. For example

```
tff(cons_type,type,                 tff(is_empty_type,type,
    cons: (fruit * list) > list ).      isEmpty: list > $o ).
```

  The argument sorts cannot be `$o`. The result sort of a function cannot be `$o`, and the result sort of a predicate must be `$o`. Note that symbols of arity greater than one use the `*` for a cross-product type – currying is not possible. If a symbol's type is declared more than once, and the types are not the same, that is an error. Multiple identical type declarations for a symbol are allowed (to support, e.g., the merging of specifications from multiple different input files).
– Defined functions and predicates have preassigned types.
  - `$true` is of type `$o`
  - `$false` is of type `$o`
  - `=` is ad-hoc polymorphic over the sorts except `$o`. The two arguments must be of the same sort, and the result sort is `$o`. The equality symbol thus represents distinct predicate symbols for each sort.
  - The types of numbers and the arithmetic functions and predicates are defined in Section 3.
– Every variable can be given a sort at quantification time. For example

```
tff(list_not_empty,axiom,
    ! [X: fruit,Xs: list] : ~isEmpty(cons(X,Xs)) ).
```

– If a symbol is used and its type has not been declared, then default types are assumed:
  - All untyped predicates get the type `($i * ... * $i) > $o`.
  - All untyped functions get the type `($i * ... * $i) > $i`.
  - All untyped variables are of the sort `$i`.

  If a symbol's type is declared later to be different from an assumed type, that is an error.

TPTP file names for TFF problems use a `_` separator (in the way that `^` is used for THF, `+` is used for FOF, and `-` is used for CNF). Use of the TFF0 language is demonstrated in the following example. The formulae are given in Figure 1.

Every student is enrolled in at least one course. Every professor teaches at least one course. Every course has at least one student enrolled. Every course has at least one professor teaching. The coordinator of a course teaches the course. If a student is enrolled in a course then the student is taught by every professor who teaches the course. Michael is enrolled in CSC410. Victor is the coordinator of CSC410. Therefore, Michael is taught by Victor.

```
%----------------------------------------------------------------------
tff(student_type,type,   student:    $tType ).
tff(professor_type,type, professor:  $tType ).
tff(course_type,type,    course:     $tType ).
tff(michael_type,type,   michael:    student ).
tff(victor_type,type,    victor:     professor ).
tff(csc410_type,type,    csc410:     course ).
tff(enrolled_type,type,  enrolled: ( student * course ) > $o ).
tff(teaches_type,type,   teaches:  ( professor * course ) > $o ).
tff(taught_by_type,type, taughtby: ( student * professor ) > $o ).
tff(coordinator_of_type,type, coordinatorof: course > professor ).

tff(student_enrolled_axiom,axiom,
    ! [X: student] : ? [Y: course] : enrolled(X,Y) ).
tff(professor_teaches,axiom,
    ! [X: professor] : ? [Y: course] : teaches(X,Y) ).
tff(course_enrolled,axiom,
    ! [X: course] : ? [Y: student] : enrolled(Y,X) ).
tff(course_teaches,axiom,
    ! [X: course] : ? [Y: professor] : teaches(Y,X) ).
tff(coordinator_teaches,axiom,
    ! [X: course] : teaches(coordinatorof(X),X) ).
tff(student_enrolled_taught,axiom,
    ! [X: student,Y: course] :
      ( enrolled(X,Y)
     => ! [Z: professor] : ( teaches(Z,Y) => taughtby(X,Z) ) ) ).
tff(michael_enrolled_csc410_axiom,axiom,
    enrolled(michael,csc410) ).
tff(victor_coordinator_csc410_axiom,axiom,
    coordinatorof(csc410) = victor ).

tff(teaching_conjecture,conjecture,
    taughtby(michael,victor) ).
%----------------------------------------------------------------------
```

Fig. 1. Example TFF problem

2.3 Type Checking and Semantics

A formula is well-typed iff all the atoms in the formula are well-typed. A non-equality atom is well-typed iff all the terms in the atom are well-typed, and the sorts of the arguments of the atom conform to the predicate symbol's type. An equality atom is well-typed iff both the terms of the equation are well-typed and are of the same sort. A term is well-typed iff all the subterms in the term are well-typed, and the sorts of the arguments of the term conform to the function symbol's type.

The semantics of TFF0 (without arithmetic) is a standard and straightforward generalization of the standard semantics of untyped first-order logic. A semantics consistent with the one below has been given, e.g., in [15].

Assume a TFF0 language with sorts $s_1, \ldots, s_n$ and variables $V = V_{s_1} \uplus \ldots \uplus V_{s_n}$, where variables from V_{s_i} have the sort s_i. Further, assume a formula (or set of formulae) build over V, function symbols from F and predicate symbols from P. An *interpretation* I consists of a *domain* $D = D_{s_1} \uplus \ldots \uplus D_{s_n}$ with disjoint, non-empty sub-domains for each sort, and a sort and arity-respecting mapping of function symbols to functions and predicate symbols to relations (representing the tuples of which the predicate holds true). In other words, if the function symbol $\mathtt{f} \in F$ is declared as $f : (s_1 * \ldots * s_n) > s$, then its interpretation is a function $I(\mathtt{f}) : D_{s_1} \times \ldots \times D_{s_n} \to D_s$. If the predicate symbol $\mathtt{p} \in P$ is declared as $p : (s_1 * \ldots * s_n) > \$\mathtt{o}$, then its interpretation is a relation $I(\mathtt{p}) \subseteq (D_{s_1} \times \ldots \times D_{s_n})$. A *typed valuation* is a function $\phi : V \to D$ with the property that $\phi(V_s) \subseteq D_s$ for all sorts s. $\phi_{x \leftarrow d}$ denotes a valuation that is equal to ϕ for all variables but X, and maps X to d.

The value of a term under an interpretation I and valuation ϕ is: $eval_{I,\phi}(X) = \phi(X)$ for $X \in V$, and $eval_{I,\phi}(\mathtt{f}(t_1, \ldots, t_n) = I(\mathtt{f})(eval_{I,\phi}(t_1), \ldots, eval_{I,\phi}(t_n))$ for $\mathtt{f} \in F$. Let $\{T, F\}$ be the truth values. For atoms $eval_{I,\phi}(\mathtt{p}(t_1, \ldots, t_n)) = T$ iff $(eval_{I,\phi}(t_1), \ldots, eval_{I,\phi}(t_n)) \in I(\mathtt{p})$. Formulae with connectives are interpreted as usual. Quantifiers, on the other hand, respect the type of the bound variable: $eval_{I,\phi}(\forall X : s \;.\; G) = T$ iff $eval_{I,\phi_{X \leftarrow d}}(G) = T$ for all $d \in D_t$ and $eval_{I,\phi}(\exists X : s \;.\; G) = T$ iff $eval_{I,\phi_{X \leftarrow d}}(G) = T$ for at least one $d \in D_t$. Note that for closed formulae the valuation of the variables is determined by the quantifiers, and the value of a closed formula depends only on the interpretation.

Recall that the equality symbol represents distinct predicate symbols for each sort, each written here as as $\mathtt{=}_s$ for a sort s. An interpretation I is an *E-interpretation*, if $I(\mathtt{=}_{s_i})$ is the equality relation on D_{s_i}, i.e., $I(\mathtt{=}_{s_i}) = \{(d, d) \mid d \in D_{s_i}\}$, for $i = 1, \ldots, n$.[4] An E-interpretation I is a *TFF model* of a formula F if $eval_I(F) = T$. As usual, a formula is *TFF satisfiable* if it has at least one TFF model, *TFF unsatisfiable* otherwise. A formula is a *TFF tautology* if every E-interpretation is a TFF model.

The semantics is alternatively given by the following standard (e.g., [36]) translation into untyped first-order logic with equality. A well-typed formula F has a typed model iff its translated untyped counterpart F' has an (untyped) model. Each sort becomes a new unary predicate in the untyped world. Then

- A TFF sort declaration a_sort : $\mathtt{\$tType}$ produces a FOF axiom $\exists X \;.\; a\_sort(X)$. This ensures that sorts are inhabited.
- Pairs of TFF sort declarations one_sort : $\mathtt{\$tType}$ and two_sort : $\mathtt{\$tType}$ produce a FOF axiom $\forall X, Y \;.\; one\_sort(X) \wedge two\_sort(Y) \Rightarrow X \neq Y$. This

[4] In refutational theorem proving it is customary to work with Herbrand interpretations and congruence relations on them to provide semantics for the equality symbol. This approach can still be used in the context of (in particular) clause logic and theory reasoning, see e.g., [19]. However, it cannot be used when arbitrary quantification is allowed, as Herbrand's theorem no longer holds, even without theories.

ensures that sorts are pairwise disjoint. These axioms are not logically necessary, because a model of the FOF formulae without these axioms can be used to construct a model of the TFF formulae [12], i.e., a formula has a model with disjoint domains iff it has a model with one domain. However, for model generation these axioms are useful because they force terms with different types to be interpreted as different domain elements, i.e., the domain of the FOF model can be divided into subdomains for the different sorts.

- A TFF function type declaration $f : (s_1 * \ldots * s_n) > s_f$ produces a FOF axiom $\forall X_1, \ldots, X_n \,.\, s_f(X_1, \ldots, X_n)$. It is unnecessary to have an implication with the antecedent checking the sorts of the arguments `X1,...,Xn`, because it is impossible to use incorrectly sorted arguments in a well-typed formula.
- Predicate type declarations are ignored.
- A TFF universally quantified formula $\forall X_1 : s_1, \ldots, X_n : s_n \,.\, p(X_1, \ldots, X_n)$ produces a FOF formula $\forall X_1, \ldots, X_n \,.\, s_1(X_1) \wedge \ldots \wedge s_n(X_n) \Rightarrow p(X_1, \ldots, X_n)$.
- A TFF existentially quantified formula $\exists X_1 : s_1, \ldots, X_n : s_n \,.\, p(X_1, \ldots, X_n)$ produces a FOF formula $\exists X_1, \ldots, X_n \,.\, s_1(X_1) \wedge \ldots \wedge s_n(X_n) \wedge p(X_1, \ldots, X_n)$.

3 TPTP Arithmetic

The TFF0 language has features that facilitate the addition of interpreted functions and predicates for integer, rational, and real arithmetic. Arithmetic requires a separate name space for numeric constants (i.e., numbers) and operators. Separate structures are assumed for integer, rational, and real arithmetic, each comprised of denumerably many numeric constants, and certain defined function and predicate symbols.

3.1 Syntax

The TPTP syntax for numeric constants[5] and the defined function and predicate symbols are given in Table 1. Each function and predicate symbol is ad-hoc polymorphic over the numeric sorts (with one exception – `$quotient` is not defined for `$int`). All arguments must have the same numeric sort. All the functions, except for the coercion functions `$to_int` and `$to_rat`, have the same result sort as their arguments. For example, `$sum` can be used with the types `($int * $int) > $int`, `($rat * $rat) > $rat`, and `($real * $real) > $real`. The coercion functions `$to_???` always have a `$???` result. All the predicates have a `$o` result. For example, `$less` can be used with the types `($int * $int) > $o`, `($rat * $rat) > $o`, and `($real * $real) > $o`.

TPTP file names for TFF problems with arithmetic use a `=` separator. Use of the TFF0 language with integer arithmetic is demonstrated in the following example. The formulae are given in Figure 2.

[5] See `http://www.tptp.org/TPTP/SyntaxBNF.html` for the precise syntax in BNF.

Table 1. The TPTP arithmetic syntax

Symbol	Usage, comments, examples
`$int`	The type of integers. Examples: 123, -123
`$rat`	The type of rationals. Examples: 123/456, -123/456, +123/456 The denominator must be unsigned and positive.
`$real`	The type of reals. Examples: 123.456, -123.456, 123.456E789.
= (infix)	See Section 2.2
`$less`/2	Less-than comparison of two numbers.
`$lesseq`/2	Less-than-or-equal-to comparison of two numbers.
`$greater`/2	Greater-than comparison of two numbers.
`$greatereq`/2	Greater-than-or-equal-to comparison of two numbers.
`$uminus`/1	Unary minus of a number.
`$sum`/2	Sum of two numbers.
`$difference`/2	Difference between two numbers.
`$product`/2	Product of two numbers.
`$quotient`/2	Exact quotient of two `$rat` or `$real` numbers. For zero divisors the result is not specified.
`$quotient_?`/2	Integral quotient of two numbers, `?` is one of `e`, `t`, or `f`. `$quotient_e` is the Euclidean quotient. `$quotient_t` and `$quotient_f` are respectively the truncation and floor of the real division of the arguments. For zero divisors the result is not specified.
`$remainder_?`/2	Remainder after integral division of two numbers using `$quotient_?`. For zero divisors the result is not specified.
`$floor`/1	Floor of a number.
`$ceiling`/1	Ceiling of a number.
`$truncate`/1	Truncation of a number.
`$round`/1	Rounding of a number.
`$is_int`/1	Test for coincidence with an integer.
`$is_rat`/1	Test for coincidence with a rational.
`$to_int`/1	Coercion of a number to `$int`, using `$floor`.
`$to_rat`/1	Coercion of a number to `$rat`. For reals that are not (known to be) rational the result is not specified.
`$to_real`/1	Coercion of a number to `$real`.

Lists of integers are constructed from a head element and a tail list, with the empty tail being represented by `nil`. A list is *Fibonacci sorted* if it is sorted, and every element is greater or equal to the sum of its two predecessors (from the third element onwards). Therefore the list $[1, 2, 4]$ is Fibonacci sorted.

The TFF arithmetic language aims to provide a comprehensive basis for automated reasoning with arithmetic. There are some minor differences between the TFF arithmetic and SMT-LIB's `Ints`, `Reals`, and `Reals_Ints` theories, e.g., rationals are not explicitly available in SMT, negative numbers are available in TFF, and the available defined predicates and functions are different. Some of the decisions regarding the TFF defined predicates and functions warrant justification: The decision to support the three integral quotients (and hence the

```
%----------------------------------------------------------------------
tff(list_type,type,   list: $tType ).
tff(nil_type,type,    nil: list ).
tff(mycons_type,type, mycons: ( $int * list ) > list ).
tff(sorted_type,type, fib_sorted: list > $o ).

tff(empty_fib_sorted,axiom,
    fib_sorted(nil) ).
tff(single_is_fib_sorted,axiom,
    ! [X: $int] : fib_sorted(mycons(X,nil)) ).
tff(double_is_fib_sorted_if_ordered,axiom,
    ! [X: $int,Y: $int] :
      ( $less(X,Y)
     => fib_sorted(mycons(X,mycons(Y,nil))) ) ).
tff(recursive_fib_sort,axiom,
    ! [X: $int,Y: $int,Z: $int,R: list] :
      ( ( $less(X,Y)
        & $greatereq(Z,$sum(X,Y))
        & fib_sorted(mycons(Y,mycons(Z,R))) )
     => fib_sorted(mycons(X,mycons(Y,mycons(Z,R)))) ) ).

tff(check_list,conjecture,
    fib_sorted(mycons(1,mycons(2,mycons(4,nil)))) ).
%----------------------------------------------------------------------
```

Fig. 2. Example TFF problem with arithmetic

corresponding remainder functions) came from John Harrison's observations [17] that most programming languages and hardware uses the "t" definition, most interactive theorem provers use the "f" definition, Boute's [9] arguments for the "e" definition are quite sound, and the "e" definition fits better with the generalization to other Euclidean rings. The decision to separate the floor, ceiling, and truncation functions from the `$to_???` type coercion functions allows the production of integral numbers from non-integral numbers without changing their type. The type coercion functions can be used separately to change the type of a number. The decision to overload the type coercion functions for all three numeric types provides the flexibility to change the types of variables in formulae without having to change the formula structure, e.g., `! [X:$real] : p($to_int(X))` can be changed to `! [X:$int] : p($to_int(X))`. Feedback on the TFF arithmetic language is welcome.

3.2 Semantics

The semantics of TFF formulae with arithmetic is defined as a refinement of the semantics of TFF0 formulae in Section 2.3. An E-interpretation I *extends arithmetic* iff (i) the domains of the numeric sorts `$int`, `$rat` and `$real` are $\mathbb{Z}$, $\mathbb{Q}$ and $\mathbb{R}$, respectively, and, (ii), the numeric constants and operators are interpreted as described in Table 1. Note that in the case of `$quotient`, `$quotient_?` and `$remainder_?` the result is not specified for zero divisors. An interpretation may assign any value to a quotient term whose divisor evaluates to zero. This way, for instance, `$quotient(5,0) = 4` is true in some interpretations and false in others. With these provisions, the semantics of TFF in Section 2.3 carries over

to TFA in the expected way. A *TFA model* of a formula F is a TFF model of F that extends arithmetics. A formula is *TFA satisfiable* if it has at least one TFA model, *TFA unsatisfiable* otherwise. A formula is a *TFA tautology* if every E-interpretation that extends arithmetic is a TFA model.

The above definitions are intended to encompass existing theorem proving approaches, such as [3,19,6,25,2], in the sense that the TFA tautologies are the same on the common logical languages and theories. For example, the approaches in [19,6,25,2] all assume a single arithmetic background theory and linear arithmetic expressions. Restricting TFA correspondingly then is intended to provide a reference semantics for these fragments.

The translation from typed to untyped logic (Section 2.3) can still be used in presence of arithmetic, to "translate away" uninterpreted sorts. Variables of a numeric sort lead to new sort predicates that recognize numeric constants. Additionally, for the overloaded arithmetic functions, e.g., `$sum`, `$difference`, etc., the translations need to have an implication with the antecedent checking the sorts of the arguments. ATP systems must build in these numeric sort predicates in order to completely and correctly process translated problems with arithmetic.

3.3 Solvability and Decidability

The extent to which ATP systems are able to work with the arithmetic predicates and functions is expected to vary, from a simple ability to do arithmetic by evaluating ground numerical terms, e.g., `$sum(2,3)` might be evaluated to `5`, through an ability to instantiate variables in equations involving such functions, e.g., `? [X:$int] : $product(2,$uminus(X)) = $uminus($sum(X,2))` might instantiate `X` to `2`, to extensive algebraic manipulation capability and ability to prove general arithmetic statements, e.g., `! [X: $int] : ? [Y: $int] : $greater(Y,X)`.

The TFA language is rich enough to accommodate virtually any interesting formula class, and asking whether a formula is TFA valid just requires stating that formula as a `conjecture`. Unfortunately, decision procedures or even semi-decision procedures for that validity problem can exist for only rather restricted fragments of TFA. For example, it is well-known that linear arithmetic (over all three numeric domains) is decidable.[6] However, as soon as free predicate symbols are allowed, semi-decidability is lost. Just adding one unary predicate symbol to linear integer arithmetic gives a validity problem that is Π^1_1-hard [16], and hence no complete calculus can exist. Whether function symbols with result sort `$int` are allowed or not does not make a difference, as they can be encoded using predicate symbols (recall that full quantification is available).

Most theorem proving calculi are based on clause logic. Without full quantification, it makes a significant difference whether free function symbols with result sort `$int` are allowed or not.[7] Without free function symbols, but with free

[6] See [21] for a recent study of decision methods based on quantifier elimination, for linear integer and for linear real arithmetic.

[7] Free function symbols with the result type `$i` are less problematic.

predicate symbols, (refutationally) complete calculi still exist (e.g., [3,6,25]). Allowing free function symbols with result sort `$int` leads again to a Π^1_1-hard unsatisfiability problem, even for formulas without `$int`-sorted variables [19]. This applies to all three numeric domains, as the integers can be encoded in the rationals (and the real numbers) [19,18]. However, completeness can be achieved under certain assumptions – see [3,19,6] for (different) approaches.

4 TFF Problems, ATP Systems, TPTP Software

Prior to the development of the TFF part of the TPTP World, ATP users and developers had long expressed support for extending the TPTP language to include the typed first-order form and arithmetic. However, there had not been a corresponding production of TPTP problems that use typing or arithmetic, or the development of ATP systems that could solve TPTP problems that use typing or arithmetic. This was a chicken-and-egg situation – without such problems in the TPTP problem library there was little infrastructure support for developing the systems, and without the systems there was little motivation for ATP users to produce such problems. It is hoped that the TFF0 developments have broken the cycle: TFF0 problems have been added to the TPTP problem library, systems that can solve TFF0 problems have been developed (with great potential for further work!), and the TPTP World infrastructure has been extended to process TFF0 problems and solutions.

TFF0 problems without and with arithmetic were added to the TPTP in release v5.0.0. The problems came from various sources. Firstly, problems were found in the many papers that describe type systems, e.g., [36,13]. Not all the problems were suitable, mainly because they employ subtyping, but others were translated to the TFF0 syntax. Secondly, existing TPTP CNF problems were analyzed for implicit type information. The CNF problems were converted in an obvious way to FOF, and then combined with the type information to produce TFF0 problems. Thirdly, TPTP users were asked for such problems, and several replied. Finally, a suite of purely arithmetic conjectures was produced, aimed at testing the basic arithmetic capabilities of ATP systems (these are in the ARI domain of the TPTP problem library). Since then some users have contributed TFF0 problems with and without arithmetic, and TPTP v5.3.0 contains 970 TFF0 problems, of which 846 include arithmetic.

Twelve ATP systems have been written for or adapted to problems written in TFF0, eleven of which have some arithmetic capability. They are CVC3 [5] H2WO4 [33], leanCoP-Ω [31], Otter [20], MELIA [7], MetiTarski [1], SNARK [28], SPASS+T [22], SPASS-XDB [35], ToFoF, Vampire, Z3 [14]. ToFoF is the system that has no arithmetic capability – it is simply the TPTP2X implementation of the translation described in Section 2.3, combined with either the E prover [27] for theorem proving or Paradox [11] for model finding. For input, H2WO4, leanCoP-Ω, MELIA, SNARK, Vampire, and Z3 read TFF0 natively. For CVC3, TPTP2X is used to translate the formulae to SMT2 syntax [4]. For the other five systems, TPTP2X is used to translate the formulae to FOF. SPASS-XDB

and the ToFoF backends read FOF natively. For leanCoP-Ω, Otter, and MetiTarski, TPTP2X is further used to export the formulae in their input syntaxes. Six of the systems rely, to a greater or lesser extent, on external procedures for dealing with the arithmetic aspects of problems. H2WO4 and SPASS-XDB use Mathematica, leanCoP-Ω uses the Omega test system [23], SPASS+T uses the Yices or CVC3 SMT solver, and MetiTarski uses the QEPCAD-B decision procedure for the theory of real closed fields [10]. All the systems are available in the SystemOnTPTP interface.[8] Seven of the systems entered the TFA division of CASC-23, which was won by SPASS+T [34].

The TPTP World infrastructure includes various tools to support ATP users and developers. This infrastructure has been extended to process TFF0 formulae. The Prolog, Java, lex/yacc, and C parsers, which are available as part of the TPTP World, have been updated to support TFF0. These developments make it possible to extend other TPTP World tools, e.g., the GDV derivation verifier and the IDV derivation viewer, to TFF0 data. A utility for checking that all symbols have declared types has been implemented, and a full type checker is being developed. This is ongoing work.

5 Conclusion

This paper has described the TPTP World infrastructure for typed first-order form logic, and its use for expressing arithmetic. The aim of developing the infrastructure is to support research, development, and deployment of ATP for the TFF logic, as a step towards satisfying a long-standing demand from ATP users. Propagation of the TFF language is partially reliant on contributions of TFF problems to the TPTP, and the automated reasoning community is encouraged to make contributions.

Current work includes the addition of conditional terms and formulae, let-binders, and a `$distinct` predicate to implement unique names. Other TPTP users are extending TFF0 with polymorphic types.[9] Future work includes developing a general framework for specifying further theories, e.g., booleans, arrays, bit-vectors, in a machine readable way, along the lines of the SMT-LIB theory specifications.

Acknowledgments. Alexandre Riazanov did the analysis of TPTP CNF problems for implicit type information. Michael Schick and Peter Watson produced many of the TFF and arithmetic problems. Mark Stickel provided a lot of useful feedback on the arithmetic syntax, and the selection of defined arithmetic functions and predicates. Uwe Waldmann provided valuable feedback on precise formulation of parts of the specification. John Harrison helped with insights on computability issues. Andrei Voronkov made some helpful suggestions.

[8] `http://www.tptp.org/cgi-bin/SystemOnTPTP`
[9] `https://sites.google.com/site/polymorphictptptff/home`

References

1. Akbarpour, B., Paulson, L.: MetiTarski: An Automatic Theorem Prover for Real-Valued Special Functions. Journal of Automated Reasoning 44(3), 175–205 (2010)
2. Althaus, E., Kruglov, E., Weidenbach, C.: Superposition Modulo Linear Arithmetic SUP(LA). In: Ghilardi, S., Sebastiani, R. (eds.) FroCoS 2009. LNCS, vol. 5749, pp. 84–99. Springer, Heidelberg (2009)
3. Bachmair, L., Ganzinger, H., Waldmann, U.: Refutational Theorem Proving for Hierachic First-Order Theories. Applicable Algebra in Engineering, Communication and Computing 5(3/4), 193–212 (1994)
4. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: Version 2.0. In: Gupta, A., Kroening, D. (eds.) Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (2010)
5. Barrett, C.W., Tinelli, C.: CVC3. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 298–302. Springer, Heidelberg (2007)
6. Baumgartner, P., Fuchs, A., Tinelli, C.: ME(LIA) - Model Evolution with Linear Integer Arithmetic Constraints. In: Cervesato, I., Veith, H., Voronkov, A. (eds.) LPAR 2008. LNCS (LNAI), vol. 5330, pp. 258–273. Springer, Heidelberg (2008)
7. Baumgartner, P., Pelzer, B., Tinelli, C.: Model Evolution with Equality - Revised and Implemented. Journal of Symbolic Computation (2011) (page to appear)
8. Benzmüller, C.E., Rabe, F., Sutcliffe, G.: THF0 – The Core of the TPTP Language for Higher-Order Logic. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 491–506. Springer, Heidelberg (2008)
9. Boute, R.: The Euclidean Definition of the Functions div and mod. ACM Transactions on Programming Languages and Systems 14(2), 127–144 (1992)
10. Brown, C.E.: QEPCAD B - A Program for Computing with Semi-algebraic sets using CADs. ACM SIGSAM Bulletin 37(4), 97–108 (2003)
11. 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)
12. Cohn, A.G.: Many Sorted Logic = Unsorted Logic + Control? In: Bramer, M. (ed.) Proceedings of Expert Systems 1986, The 6th Annual Technical Conference on Research and Development in Expert Systems, pp. 184–194. Cambridge University Press (1986)
13. Cohn, A.G.: A More Expressive Formulation of Many Sorted Logic. Journal of Automated Reasoning 3(2), 113–200 (1987)
14. de Moura, L., Bjørner, N.S.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
15. Gallier, J.: Logic for Computer Science: Foundations of Automated Theorem Proving. Computer Science and Technology Series. Wiley (1986)
16. Halpern, J.: Presburger Arithmetic With Unary Predicates is Π_1^1-Complete. Journal of Symbolic Logic 56(2), 637–642 (1991)
17. Harrison, J.: Email to Cesare Tinelli
18. Harrison, J.: Handbook of Practical Logic and Automated Reasoning. Cambridge University Press (2009)
19. Korovin, K., Voronkov, A.: Integrating Linear Arithmetic into Superposition Calculus. In: Duparc, J., Henzinger, T.A. (eds.) CSL 2007. LNCS (LNAI), vol. 4646, pp. 223–237. Springer, Heidelberg (2007)

20. McCune, W.W.: Otter 3.3 Reference Manual. Technical Report ANL/MSC-TM-263, Argonne National Laboratory, Argonne, USA (2003)
21. Nipkow, T.: Linear Quantifier Elimination. Journal of Automated Reasoning 45(2), 189–212 (2010)
22. Prevosto, V., Waldmann, U.: SPASS+T. In: Sutcliffe, G., Schmidt, R., Schulz, S. (eds.) Proceedings of the FLoC 2006 Workshop on Empirically Successful Computerized Reasoning, 3rd International Joint Conference on Automated Reasoning. CEUR Workshop Proceedings, vol. 192, pp. 19–33 (2006)
23. Pugh, W.: The Omega Test: A Fast and Practical Integer Programming Algorithm for Dependence Analysis. Communications of the ACM 31(8), 4–13 (1992)
24. Ranise, S., Tinelli, C.: The SMT-LIB Format: An Initial Proposal. In: Nebel, B., Swartout, W. (eds.) Proceedings of the Workshop on Pragmatics of Decision Procedures in Automated Reasoning (2003)
25. Rümmer, P.: A Constraint Sequent Calculus for First-Order Logic with Linear Integer Arithmetic. In: Cervesato, I., Veith, H., Voronkov, A. (eds.) LPAR 2008. LNCS (LNAI), vol. 5330, pp. 274–289. Springer, Heidelberg (2008)
26. Schmidt-Schauss, M.: A Many-Sorted Calculus with Polymorphic Functions Based on Resolution and Paramodulation. In: Joshi, A. (ed.) Proceedings of the 9th International Joint Conference on Artificial Intelligence, pp. 1162–1168 (1985)
27. Schulz, S.: E: A Brainiac Theorem Prover. AI Communications 15(2-3), 111–126 (2002)
28. Stickel, M.E.: SNARK - SRI's New Automated Reasoning Kit, `http://www.ai.sri.com/ stickel/snark.html`
29. 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)
30. 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)
31. Sutcliffe, G.: Proceedings of the 5th IJCAR ATP System Competition. Edinburgh, United Kingdom (2010)
32. Sutcliffe, G.: The TPTP World – Infrastructure for Automated Reasoning. In: Clarke, E.M., Voronkov, A. (eds.) LPAR-16 2010. LNCS, vol. 6355, pp. 1–12. Springer, Heidelberg (2010)
33. Sutcliffe, G.: Proceedings of the CADE-23 ATP System Competition. Wroclaw, Poland (2011)
34. Sutcliffe, G.: The CADE-23 Automated Theorem Proving System Competition - CASC-23. AI Communications (page to appear, 2012)
35. 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, pp. 110–115. AAAI Press (2010)
36. Walther, C.: A Many-Sorted Calculus Based on Resolution and Paramodulation. In: Bundy, A. (ed.) Proceedings of the 8th International Joint Conference on Artificial Intelligence, pp. 882–891 (1983)

Ordinals and Knuth-Bendix Orders

Sarah Winkler*, Harald Zankl, and Aart Middeldorp

Institute of Computer Science, University of Innsbruck, 6020 Innsbruck, Austria

Abstract In this paper we consider a hierarchy of three versions of Knuth-Bendix orders. (1) We show that the standard definition can be (slightly) simplified without affecting the ordering relation. (2) For the extension of *transfinite Knuth-Bendix orders* we show that transfinite ordinals are not needed as weights, as far as termination of finite rewrite systems is concerned. (3) Nevertheless termination proving benefits from transfinite ordinals when used in the setting of *general Knuth-Bendix orders* defined over a weakly monotone algebra. We investigate the relationship to polynomial interpretations and present experimental results for both termination analysis and ordered completion. For the latter it is essential that the order is totalizable on ground terms.

Keywords: Knuth-Bendix order, termination, ordered completion.

1 Introduction

The Knuth-Bendix order (KBO) [10] is a popular criterion for automated termination analysis and theorem proving. Consequently many extensions and generalizations of this order have been proposed and investigated [2–4,6,12,13,15,18,19]. Despite the fact that this order is so well-studied we show that the definition of KBO can be simplified without affecting the ordering relation.

Concerning generalizations of this ordering, Dershowitz [2,3] suggested to extend the semantic component beyond weight functions and Middeldorp and Zantema [15] presented the *generalized Knuth-Bendix order* (GKBO) using weakly monotone algebras. Independently in the theorem proving community, McCune [14] suggested linear functions for computing weights of terms. Recently Ludwig and Waldmann [13] introduced the *transfinite Knuth-Bendix order* (TKBO), which allows linear functions over the ordinals and Kovács *et al.* [12] show that for finite signatures one can restrict to ordinals below ω^{ω^ω} without losing power. However, for finite rewrite systems (which is the typical case for proving termination) we show that finite weights suffice. This is in sharp contrast to GKBO where transfinite ordinals are beneficial. We also show how a restricted version of this ordering can be implemented. To this end (a fragment of) ordinal arithmetic is encoded as a constraint satisfaction problem. The usefulness of the different versions of KBO is illustrated by experimental results for both termination analysis and theorem proving.

* Supported by a DOC-fFORTE fellowship of the Austrian Academy of Sciences.

N. Bjørner and A. Voronkov (Eds.): LPAR-18, LNCS 7180, pp. 420–434, 2012.

The remainder of this paper is organized as follows. In the next section we recall preliminaries. In Section 3 we prove that the weight of variables can be fixed a priori without affecting the power of KBO. Section 4 shows that for finite rewrite systems no transfinite ordinals are needed for TKBO. Section 5 shows the benefit of transfinite ordinals for GKBO and studies the relationship to polynomial interpretations. Implementation issues and experimental results are discussed in Section 6. Section 7 concludes.

2 Preliminaries

Term Rewriting: We assume familiarity with term rewriting and termination [21]. Let $\mathcal{F}$ be a signature and $\mathcal{V}$ a set of variables. By $\mathcal{T}(\mathcal{F},\mathcal{V})$ we denote the terms over $\mathcal{F}$ and $\mathcal{V}$. For a term t let $\mathcal{P}\mathsf{os}(t)$ be the set of positions in t and $\mathcal{P}\mathsf{os}_x(t)$ the set of positions of the variable x in t. A (well-founded $\mathcal{F}$-)algebra $(\mathcal{A},>)$ consists of a non-empty carrier A, a well-founded relation $>$ on A, and an interpretation function $f_{\mathcal{A}}$ for every $f \in \mathcal{F}$. By $[\alpha]_{\mathcal{A}}(\cdot)$ we denote the usual evaluation function of $\mathcal{A}$ according to an assignment α. An algebra $(\mathcal{A},>)$ is called (weakly) monotone if every $f_{\mathcal{A}}$ is (weakly) monotone with respect to $>$. Any algebra $(\mathcal{A},>)$ induces an order on terms, as follows: $s >_{\mathcal{A}} t$ if for any assignment α the condition $[\alpha]_{\mathcal{A}}(s) > [\alpha]_{\mathcal{A}}(t)$ holds. The order $\geqslant_{\mathcal{A}}$ is defined similarly based on the reflexive closure of $>$. We say that a TRS $\mathcal{R}$ is compatible with relation $>$ (an algebra $\mathcal{A}$) if $\ell \to r \in \mathcal{R}$ implies $\ell > r$ ($\ell >_{\mathcal{A}} r$). It is well-known that every TRS that is compatible with a monotone algebra is terminating.

An interpretation $f_{\mathcal{A}}$ is simple if $f_{\mathcal{A}}(a_1,\dots,a_n) > a_i$ for all $a_1,\dots,a_n \in A$ and $1 \leqslant i \leqslant n$. An algebra $\mathcal{A}$ is simple if all its interpretation functions are simple. A polynomial interpretation $\mathcal{N}$ is a monotone algebra over the carrier $\mathbb{N} = \{0,1,2,\dots\}$, using $>_{\mathbb{N}}$ as ordering and where every $f_{\mathcal{N}}$ is a polynomial. A polynomial interpretation where every $f_{\mathcal{N}}$ is linear is called a linear interpretation.

Ordinals: We assume basic knowledge of ordinals [9]. Let $+$ and $\cdot$ denote standard addition and multiplication on ordinals (and hence also on natural numbers). Likewise let $\oplus$ and $\odot$ denote natural addition and multiplication of ordinals. Let $\mathcal{O}$ be the set of ordinals strictly less than ϵ_0. Recall that every ordinal $\alpha \in \mathcal{O}$ can be uniquely represented in Cantor Normal Form (CNF):

$$\alpha = \sum_{1 \leqslant i \leqslant n} \omega^{\alpha_i} \cdot a_i$$

where $a_1,\dots,a_n \in \mathbb{N} \setminus \{0\}$ and $\alpha_1,\dots,\alpha_n \in \mathcal{O}$ are also in CNF, with $\alpha_1 > \cdots > \alpha_n$. Ordinals below ω—the natural numbers—are called finite.

3 KBO

A precedence is a strict order on a signature. A weight function for a signature $\mathcal{F}$ is a pair (w, w_0) consisting of a mapping $w\colon \mathcal{F} \to \mathbb{N}$ and a positive constant $w_0 \in \mathbb{N}$ such that $w(c) \geqslant w_0$ for every constant $c \in \mathcal{F}$. A weight function (w, w_0)

is admissible for a precedence $\succ$ if for every unary $f \in \mathcal{F}$ with $w(f) = 0$ we have $f \succ g$ for all $g \in \mathcal{F} \setminus \{f\}$. The weight of a term is computed as follows: $w(x) = w_0$ for $x \in \mathcal{V}$ and $w(f(t_1, \ldots, t_n)) = w(f) + w(t_1) + \cdots + w(t_n)$. By $|t|_x$ we denote how often a variable x occurs in a term t.

Definition 1. *Let $\succ$ be a precedence and (w, w_0) a weight function. We define the* Knuth-Bendix order $\succ_{\mathrm{kbo}}$ *inductively as follows: $s \succ_{\mathrm{kbo}} t$ if $|s|_x \geqslant |t|_x$ for all variables $x \in \mathcal{V}$ and either $w(s) > w(t)$, or $w(s) = w(t)$ and one of the following alternatives holds:*

(1) $s = f^k(x)$ and $t = x$ for some $k > 0$, or
(2) $s = f(s_1, \ldots, s_n)$, $t = g(t_1, \ldots, t_m)$, and $f \succ g$, or
(3) $s = f(s_1, \ldots, s_n)$, $t = f(t_1, \ldots, t_n)$, $s_1 = t_1, \ldots, s_{k-1} = t_{k-1}$, and $s_k \succ_{\mathrm{kbo}} t_k$ with $1 \leqslant k \leqslant n$.

To indicate the weight function (w, w_0) used for $\succ_{\mathrm{kbo}}$ we write $\succ_{\mathrm{kbo}}^{(w,w_0)}$. A TRS $\mathcal{R}$ is called compatible with KBO if there exists a weight function (w, w_0) admissible for a precedence $\succ$ such that $\mathcal{R}$ is compatible with $\succ_{\mathrm{kbo}}^{(w,w_0)}$.

Theorem 2 ([4,10]). *A TRS is terminating if it is compatible with KBO.* □

Below we examine if restricting w_0 to one decreases the power of KBO. We are not aware of any earlier investigations in this direction. A bit surprisingly indeed w_0 can be chosen one, which simplifies the definition of KBO. Note that it does not suffice to just replace w_0 by one. Consider the rule $\mathsf{h}(x, x) \to \mathsf{f}(x)$ and $w(\mathsf{h}) = 0$, $w(\mathsf{f}) = 2$. Using $w_0 = 3$ the constraints on the weight give $6 > 5$ but $w_0 = 1$ yields $2 \not> 3$. But a KBO proof with $w_0 > 1$ can be transformed into a KBO proof with $w_0 = 1$ by adapting (according to their arity) the weights of function symbols. Formally, we define a new weight function with

$$(w_0)^1 := 1 \qquad w^1(f) := w(f) + (n-1) \cdot (w_0 - 1) \tag{1}$$

for every n-ary function symbol f. Obviously $w^1(f) \geqslant 0$ for all $f \in \mathcal{F}$ and in particular $w^1(c) \geqslant (w_0)^1$ for constants $c \in \mathcal{F}$ since $w(c) \geqslant w_0$. Note that this transformation is not invertible since one could get negative weights for function symbols of higher arity. To cope with the reverse direction, i.e., transforming a KBO proof with arbitrary w_0 into a KBO proof with $w_0 = k$ for some $k \in \mathbb{N} \setminus \{0\}$ we define a weight function with

$$(w_0)_k := k \qquad w_k(f) := w^1(f) \cdot k \tag{2}$$

Lemma 3. *For terms s and t we have $w(s) > w(t)$ if and only if $w^1(s) > w^1(t)$ if and only if $w_k(s) > w_k(t)$ for any $k \in \mathbb{N} \setminus \{0\}$.*

Proof. The first statement follows from $w(s) = w^1(s) + w_0 - 1$, which we show by induction on the term s. In the base case $s \in \mathcal{V}$ and $w(s) = w_0 = w^1(s) + w_0 - 1$ since $w^1(s) = 1$. In the step case $s = f(s_1, \ldots, s_n)$ and we have

$$\begin{aligned} w(s) &= w(f) + w(s_1) + \cdots + w(s_n) \\ &= w(f) + w^1(s_1) + w_0 - 1 + \cdots + w^1(s_n) + w_0 - 1 \\ &= w^1(f) - (n-1) \cdot (w_0 - 1) + w^1(s_1) + w_0 - 1 + \cdots + w^1(s_n) + w_0 - 1 \\ &= w^1(f) + w^1(s_1) + \cdots + w^1(s_n) + w_0 - 1 = w^1(s) + w_0 - 1 \end{aligned}$$

where the induction hypothesis is used in the second step. The reasoning for the second statement is similar and based on $w_k(s) = w^1(s) \cdot k$. □

Note that Lemma 3 implies that for terms s and t we have $w(s) = w(t)$ if and only if $w^1(s) = w^1(t)$ if and only if $w_k(s) = w_k(t)$ for any $k \in \mathbb{N} \setminus \{0\}$.

Corollary 4. *We have* $\succ_{\mathrm{kbo}}^{(w,w_0)} = \succ_{\mathrm{kbo}}^{(w^1,1)} = \succ_{\mathrm{kbo}}^{(w_k,k)}$ *for all* $k \in \mathbb{N} \setminus \{0\}$. □

Lemma 5. *The weight function* (w, w_0) *is admissible for a precedence* $\succ$ *if and only if the weight functions* $(w^1, 1)$ *and* (w_k, k) *are admissible for* $\succ$.

Proof. The result follows from the fact that $w^1(f) = w(f)$ for unary $f \in \mathcal{F}$ and $w(f) \neq 0$ if and only if $w_k(f) = w^1(f) \cdot k \neq 0$. □

From the above results we immediately obtain that fixing the value of w_0 does not affect the power of KBO. Our implementation (see Section 6) benefited slightly from fixing $w_0 = 1$.

By now we have the machinery to show that KBO cannot demand lower bounds on weights (apart from unary function symbols to have weight zero). This can be supported as follows. Consider $\succ_{\mathrm{kbo}}^{(w_2,2)}$. By Corollary 4

$$\succ_{\mathrm{kbo}}^{(w_2,2)} = \succ_{\mathrm{kbo}}^{((w_2)^1,1)} = \succ_{\mathrm{kbo}}^{(((w_2)^1)_k,k)}$$

for any $k \in \mathbb{N} \setminus \{0\}$. Note that $(w_2)^1(f) \geqslant 1$ for all non-unary $f \in \mathcal{F}$ due to (1). Now choosing an appropriate k, all $f \in \mathcal{F}$ (with the possible exception of a unary function symbol of weight zero) satisfy $((w_2)^1)_k(f) \geqslant k$ by (2). This does not contradict [23, Theorem 1], which allows to compute an a priori upper bound on the weights.

A KBO with $w_0 = 0$ is not well-founded. The nonterminating TRS consisting of the rule $\mathsf{h}(\mathsf{h}(\mathsf{a},\mathsf{a}),\mathsf{a}) \to \mathsf{h}(\mathsf{a},\mathsf{h}(\mathsf{h}(\mathsf{a},\mathsf{a}),\mathsf{a}))$ is compatible with $\succ_{\mathrm{kbo}}^{(w,0)}$ where $w(\mathsf{a}) = w(\mathsf{h}) = 0$ and $\mathsf{h} \succ \mathsf{a}$.

4 Transfinite KBO

We will now consider the *transfinite Knuth-Bendix order* (TKBO) [13]. In this setting a weight function for a signature $\mathcal{F}$ is a pair (w, w_0) consisting of a mapping $w\colon \mathcal{F} \to \mathcal{O}$ and a positive constant $w_0 \in \mathbb{N}$ such that $w(c) \geqslant w_0$ for every constant $c \in \mathcal{F}$. A subterm coefficient function is a mapping $s\colon \mathcal{F} \times \mathbb{N} \to \mathcal{O}$ such that for a function symbol f of arity n we have $s(f,i) > 0$ for all $1 \leqslant i \leqslant n$. A TKBO where w_0,[1] all weights and subterm coefficients are finite will be

[1] Although $w_0 \in \mathbb{N}$ by definition, Theorem 13 is valid also for transfinite w_0.

called finite. Let (w, w_0) be a weight function and s a subterm coefficient function. We define the weight of a term inductively as follows: $w(t) = w_0$ for $t \in \mathcal{V}$ and $w(t) = w(t_1) \odot s(f,1) \oplus \cdots \oplus w(t_n) \odot s(f,n) \oplus w(f)$ if $t = f(t_1, \ldots, t_n)$. Given a term t and a subterm coefficient function s, the *coefficient* of a position $p \in \mathcal{P}\mathsf{os}(t)$ is inductively defined by $\mathit{coeff}(p,t) = 1$ if $p = \epsilon$ and $s(f,i) \odot \mathit{coeff}(q,t_i)$ if $t = f(t_1, \ldots, t_n)$ and $p = iq$. The *variable coefficient* of $x \in \mathcal{V}$ is $\mathit{vcoeff}(x,t) = \bigoplus_{p \in \mathcal{P}\mathsf{os}_x(t)} \mathit{coeff}(p,t)$.

Definition 6. *Let $\succ$ be a precedence, (w, w_0) a weight function and s a subterm coefficient function. We define the* transfinite Knuth-Bendix order $\succ_{\mathrm{tkbo}}$ *inductively as follows: $s \succ_{\mathrm{tkbo}} t$ if $\mathit{vcoeff}(x,s) \geqslant \mathit{vcoeff}(x,t)$ for all variables $x \in \mathcal{V}$ and either $w(s) > w(t)$, or $w(s) = w(t)$ and one of the following alternatives holds:*

(1) $s = f^k(x)$ and $t = x$ for some $k > 0$, or
(2) $s = f(s_1, \ldots, s_n)$, $t = g(t_1, \ldots, t_m)$, and $f \succ g$, or
(3) $s = f(s_1, \ldots, s_n)$, $t = f(t_1, \ldots, t_n)$, $s_1 = t_1, \ldots, s_{k-1} = t_{k-1}$, and $s_k \succ_{\mathrm{tkbo}} t_k$ with $1 \leqslant k \leqslant n$.

A TRS $\mathcal{R}$ is called compatible with TKBO if there is a weight function (w, w_0) admissible for a precedence $\succ$ and a subterm coefficient function s such that $\mathcal{R}$ is compatible with $\succ_{\mathrm{tkbo}}$.

Theorem 7 ([12,13]). *A TRS is terminating if it is compatible with TKBO.* □

In fact $\succ_{\mathrm{tkbo}}$ still satisfies the subterm property if admissibility is relaxed to requiring that a unary function symbol f with $w(f) = 0$ and $s(f,1) = 1$ satisfies $f \succ g$ for all $g \in \mathcal{F} \setminus \{f\}$. Hence Theorem 7 remains valid. Thus in the sequel we will use this less restrictive definition of admissibility.

One motivation in [13] for allowing subterm coefficients is to cope with duplicating rules such as $\mathsf{f}(x) \to \mathsf{h}(x,x)$. We agree with this observation but show that the benefit is not limited to this case, i.e., subterm coefficients are even useful for string rewrite systems (where all function symbols are unary).

Example 8. Consider the SRS consisting of the following rule

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

For KBO this rule demands $w(\mathsf{g}) = 0$ and the admissibility condition induces $\mathsf{g} \succ \mathsf{f}$ which does not orient the rule from left to right. On the other hand the SRS is compatible with the TKBO using weights $w(\mathsf{f}) = 1$ and $w(\mathsf{g}) = 1$ and a subterm coefficient function satisfying $s(\mathsf{f},1) = 3$ and $s(\mathsf{g},1) = 1$.

In the remainder of this section we show that if a *finite* TRS is compatible with a TKBO then it also is compatible with a finite TKBO. Assume termination of a finite rewrite system $\mathcal{R}$ is shown by a TKBO with weight function (w, w_0) and subterm coefficient function s. If some weights or subterm coefficients are infinite then $\mathcal{R}$ is compatible with the finite TKBO which is obtained if one "substitutes" a large enough natural number for ω in the given weight and subterm coefficient functions. The next example demonstrates the proof idea while the proof of Theorem 13 provides a suitable choice for this natural number.

Example 9. The TRS $\mathcal{R}$ consisting of the following three rules

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

is compatible with the TKBO using the weight function

$$w_0 = 1 \qquad w(\mathsf{g}) = 0 \qquad w(\mathsf{f}) = 5 \qquad w(\mathsf{h}) = \omega \qquad w(\mathsf{k}) = \omega^2 + 1$$

and g greatest in the precedence. (All subterm coefficients are set to one.) Substituting 13 for ω makes $\mathcal{R}$ also compatible with the finite TKBO using

$$w_0 = 1 \qquad w(\mathsf{g}) = 0 \qquad w(\mathsf{f}) = 5 \qquad w(\mathsf{h}) = 13 \qquad w(\mathsf{k}) = 13^2 + 1 = 170$$

and g greatest in the precedence.

Definition 10. *For an ordinal $\alpha = \sum_{1 \leqslant i \leqslant n} \omega^{\alpha_i} \cdot a_i$ in CNF, let $M(\alpha)$ denote the maximal natural number occurring as coefficient, i.e., the maximum of $a_1, \ldots, a_n$ and all coefficients occurring in $\alpha_1, \ldots, \alpha_n$. (If $\alpha = 0$ then $M(\alpha) = 0$.) We denote by $\alpha(k)$ the natural number obtained when α—considered as a function in ω—is evaluated for the natural number k, i.e., $\alpha(k) := \sum_{1 \leqslant i \leqslant n} k^{\alpha_i(k)} \cdot a_i$.*

Let $\alpha = \omega^{\omega \cdot 3} + \omega^2 \cdot 2 + 1$. Then $M(\alpha) = 3$ and $\alpha(3) = 3^{3 \cdot 3} + 3^2 \cdot 2 + 1 = 19,702$. The next result follows from a statement in [7, p. 34].

Lemma 11. *Let $\alpha, \beta \in \mathcal{O}$ and $k \in \mathbb{N}$ with $k > M(\alpha), M(\beta)$. If $\alpha = \beta$ then $\alpha(k) = \beta(k)$ and if $\alpha > \beta$ then $\alpha(k) > \beta(k)$.* □

Note that the restriction on k in Lemma 11 is essential: if $k = 2$, $\alpha = \omega$, and $\beta = 2$ then $\alpha > \beta$ but $\alpha(k) = \beta(k) = 2$.

Lemma 12. *For $\alpha, \beta \in \mathcal{O}$ we have*

(1) $(\alpha \oplus \beta)(k) = \alpha(k) + \beta(k)$
(2) $(\alpha \odot \beta)(k) = \alpha(k) \cdot \beta(k)$

Proof. Let $\alpha = \sum_{1 \leqslant i \leqslant n} \omega^{\alpha_i} \cdot a_i$ and $\beta = \sum_{1 \leqslant j \leqslant m} \omega^{\beta_j} \cdot b_j$ be in CNF.

(1) We may also write $\alpha = \sum_{1 \leqslant i \leqslant l} \omega^{\gamma_i} \cdot a'_i$ and $\beta = \sum_{1 \leqslant i \leqslant l} \omega^{\gamma_i} \cdot b'_i$ such that $\{\gamma_1, \ldots, \gamma_l\} = \{\alpha_1, \ldots, \alpha_n\} \cup \{\beta_1, \ldots, \beta_m\}$ where some a'_i and b'_i may be zero. We then have

$$\begin{aligned}
(\alpha \oplus \beta)(k) &= \left(\sum_{1 \leqslant i \leqslant l} \omega^{\gamma_i} \cdot (a'_i + b'_i) \right)(k) && \text{(definition of } \oplus) \\
&= \sum_{1 \leqslant i \leqslant l} k^{\gamma_i(k)} \cdot (a'_i + b'_i) \\
&= \sum_{1 \leqslant i \leqslant l} k^{\gamma_i(k)} \cdot a'_i + \sum_{1 \leqslant i \leqslant l} k^{\gamma_i(k)} \cdot b'_i \\
&= \alpha(k) + \beta(k)
\end{aligned}$$

(2) We have

$$\begin{aligned}(\alpha \odot \beta)(k) &= \left(\bigoplus_{i=1}^{n}\bigoplus_{j=1}^{m} \omega^{\alpha_i \oplus \beta_j} \cdot a_i \cdot b_j\right)(k) && \text{(definition of } \odot\text{)}\\ &= \sum_{i=1}^{n}\sum_{j=1}^{m} k^{\alpha_i(k)+\beta_j(k)} \cdot a_i \cdot b_j && (\star)\\ &= \left(\sum_{i=1}^{n} k^{\alpha_i(k)} \cdot a_i\right) \cdot \left(\sum_{j=1}^{m} k^{\beta_j(k)} \cdot b_j\right)\\ &= \alpha(k) \cdot \beta(k)\end{aligned}$$

where in step $(\star)$ we used part (1). □

Using Lemmata 11 and 12 we can prove the following result.

Theorem 13. *If a finite TRS is compatible with TKBO then it is compatible with a finite TKBO.*

Proof. Let $\mathcal{R}$ be a finite TRS compatible with a TKBO using weight function (w, w_0) and subterm coefficient function s. Since $\mathcal{R}$ is finite the natural number

$$k := \max\{M(w(\ell)), M(w(r)) \mid \ell \to r \in \mathcal{R}\} + 1$$

is well-defined (the maximum of the empty set is zero). We have $k > M(w(t))$ for all terms t occurring in $\mathcal{R}$. Consider the weight function given by $w'(f) := \alpha(k)$ whenever $w(f) = \alpha$ and $w'_0 := \beta(k)$ if $w_0 = \beta$ together with the subterm coefficient function $s'(f,i) := \alpha(k)$ whenever $s(f,i) = \alpha$, which assigns only natural numbers as weights and subterm coefficients. For any term t we then have $w'(t) = (w(t))(k)$, as is easily verified by induction: if $t \in \mathcal{V}$ then $w'(t) = w'_0 = (w_0)(k) = (w(t))(k)$ and

$$\begin{aligned}w'(f(t_1,\ldots,t_n)) &= \left(\sum_{i=1}^{n} w'(t_i) \cdot s'(f,i)\right) + w'(f)\\ &= \left(\sum_{i=1}^{n} (w(t_i))(k) \cdot (s(f,i))(k)\right) + (w(f))(k)\\ &= \left(\left(\bigoplus_{i=1}^{n} w(t_i) \odot s(f,i)\right) \oplus w(f)\right)(k)\\ &= (w(f(t_1,\ldots,t_n)))(k)\end{aligned}$$

where in the second step we used the induction hypothesis and the definition of $w'(f)$ and $s'(f,i)$, respectively, and the last but one step applies Lemma 12. Thus by Lemma 11 $w(\ell) > w(r)$ implies $w'(\ell) = (w(\ell))(k) > (w(r))(k) = w'(r)$ and $w(\ell) = w(r)$ implies $w'(\ell) = (w(\ell))(k) = (w(r))(k) = w'(r)$ for each $\ell \to r \in \mathcal{R}$. Note that admissibility is not affected. Hence $\mathcal{R}$ is compatible with the TKBO having weight function (w'_0, w') and subterm coefficient function s'. □

We remark that Theorem 13 does not make transfinite ordinal weights in TKBO superfluous since they are beneficial for hierarchic theorem proving [13]. Furthermore, Theorem 13 only applies to finite TRSs as [12, Theorem 5.8] shows that there are TRSs over finite signatures that need transfinite ordinals (larger than ω^{ω^k} for any $k \in \mathbb{N}$) for weights and subterm coefficients in TKBO. However, due to Theorem 13 the TRS showing the need for transfinite weights is necessarily infinite.

5 Generalized KBO

In this section we elaborate on the question if transfinite ordinals increase the power of KBO. As long as natural addition and multiplication of ordinals are considered, Theorem 13 shows that (for finite TRSs) coefficients beyond $\mathbb{N}$ can be ignored. Although standard addition and multiplication of ordinals are only weakly monotone, they can still be employed for KBO, as we recall in this section. To this end we consider the *generalized Knuth-Bendix order* (GKBO) [15] which computes weights of terms according to a weakly monotone simple algebra.[2] Similar extensions have been presented in [2,3,6,19].

Definition 14. *Let $(\mathcal{A}, >)$ be a weakly monotone simple algebra and $\succ$ a precedence on $\mathcal{F}$. We define the general Knuth-Bendix order $\succ_{\text{gkbo}}$ inductively as follows: $s \succ_{\text{gkbo}} t$ if $s >_{\mathcal{A}} t$, or $s \geqslant_{\mathcal{A}} t$ and either*

(1) $s = f(s_1, \ldots, s_n)$, $t = g(t_1, \ldots, t_m)$, and $f \succ g$, or
(2) $s = f(s_1, \ldots, s_n)$, $t = f(t_1, \ldots, t_n)$, $s_1 = t_1, \ldots, s_{k-1} = t_{k-1}$, and $s_k \succ_{\text{gkbo}} t_k$ with $1 \leqslant k \leqslant n$.

Theorem 15 ([15]). *A TRS $\mathcal{R}$ is terminating if it is compatible with $\succ_{\text{gkbo}}$.* □

The condition that $\mathcal{A}$ is simple ensures admissibility with $\succ$ since it e.g. rules out interpretations of the form $f_{\mathcal{A}}(x) = x$. Next we elaborate on the use of standard addition and multiplication of ordinals for GKBO.

Example 16. Consider the SRS $\mathcal{R}$ containing the rules

$$1\colon \mathsf{a}(x) \to \mathsf{b}(x) \qquad 2\colon \mathsf{a}(\mathsf{b}(x)) \to \mathsf{b}(\mathsf{c}(\mathsf{a}(x))) \qquad 3\colon \mathsf{c}(\mathsf{b}(x)) \to \mathsf{a}(x)$$

The following weakly monotone interpretation

$$\mathsf{a}_{\mathcal{O}}(x) = x + \omega + 1 \qquad \mathsf{b}_{\mathcal{O}}(x) = x + \omega \qquad \mathsf{c}_{\mathcal{O}}(x) = x + 2$$

is simple and induces a strict decrease between left- and right-hand sides:

$$x + \omega + 1 >_{\mathcal{O}} x + \omega \quad x + \omega \cdot 2 + 1 >_{\mathcal{O}} x + \omega \cdot 2 \quad x + \omega + 2 >_{\mathcal{O}} x + \omega + 1$$

Hence $\mathcal{R}$ can be oriented by GKBO.

[2] Note that in contrast to [15] we restrict to the case where all function symbols have lexicographic status and arguments are compared from left to right.

Below we show that $\mathcal{R}$ cannot be shown terminating by GKBO using a linear interpretation over $\mathbb{N}$. To this end we assume abstract interpretations $f_{\mathcal{N}}(x) = f_1 x + f_0$. Since $\mathcal{N}$ must be simple we need $f_1, f_0 \geqslant 1$. For rule (2) the constraint on variable coefficients induces $a_1 b_1 \geqslant b_1 c_1 a_1$, which requires $c_1 = 1$. Rules (1) and (3) demand $a_1 \geqslant b_1$ and $b_1 \geqslant a_1$, so $b_1 = a_1$. Rule (2) further requires $a_1 b_0 + a_0 \geqslant b_1 a_0 + b_1 c_0 + b_0$. Because $a_0 \geqslant b_0$ due to rule (1), this demands $a_0 \geqslant a_1 c_0 + b_0 = (a_1 - 1)c_0 + (c_0 + b_0)$. Rule (3) demands $c_0 + b_0 \geqslant a_0$, and hence $(a_1 - 1)c_0 = 0$. Since $\mathcal{N}$ must be simple, $c_0 > 0$ which requires $a_1 = 1$. But then rule (2) implies the constraint $b_0 + a_0 \geqslant a_0 + c_0 + b_0$, which again contradicts that c_0 is positive.

Hence one might conclude that standard addition and multiplication of ordinals is more useful for termination proving (where one usually deals with finite TRSs) than their natural counterparts. But standard addition of ordinals might cause problems for (at least) binary function symbols since the absolute positiveness approach [8] to compare polynomials no longer applies. To see this note that $f_1 \geqslant g_1$ and $f_2 \geqslant g_2$ does not imply $x \cdot f_1 + y \cdot f_2 \geqslant y \cdot g_2 + x \cdot g_1$ for all values of x and y if $f_1, f_2, g_1, g_2 \in \mathbb{N}$ and $x, y \in \mathcal{O}$. Next we show that a combination of standard and natural addition is helpful.

Example 17. Consider the TRS $\mathcal{R}$ consisting of the single rule

$$\mathsf{s}(\mathsf{f}(x, y)) \to \mathsf{f}(\mathsf{s}(y), \mathsf{s}(\mathsf{s}(x)))$$

The weakly monotone interpretation $\mathsf{f}_{\mathcal{O}}(x, y) = (x \oplus y) + \omega$ and $\mathsf{s}_{\mathcal{O}}(x) = x + 1$ is simple and induces a strict decrease between left- and right-hand side:

$$(x \oplus y) + \omega + 1 >_{\mathcal{O}} ((y + 1) \oplus (x + 2)) + \omega = (x \oplus y) + 3 + \omega = (x \oplus y) + \omega$$

Hence $\mathcal{R}$ can be oriented by GKBO. Again, linear interpretations with coefficients in $\mathbb{N}$ are not sufficient: Assuming abstract interpretations $\mathsf{f}_{\mathcal{N}}(x, y) = f_1 x + f_2 y + f_0$ and $\mathsf{s}_{\mathcal{N}}(x) = s_1 x + s_0$, we get the constraints

$$s_1 f_1 \geqslant f_2 s_1 s_1 \qquad s_1 f_2 \geqslant f_1 s_1 \qquad s_1 f_0 + s_0 \geqslant f_1 s_0 + f_2(s_0 + s_1 s_0) + f_0$$

Since $\mathsf{s}_{\mathcal{N}}$ and $\mathsf{f}_{\mathcal{N}}$ must be simple $s_1, f_1, f_2 \geqslant 1$. From the first two constraints we conclude $s_1 = 1$, such that the third simplifies to $f_0 + s_0 \geqslant f_0 + (f_1 + 2f_2)s_0$. This contradicts f_1, f_2, and s_0 being positive, which is needed for $\mathcal{N}$ being simple.

GKBO vs. Polynomial Interpretations

Finally we investigate the relationship of polynomial interpretations and GKBO using a weakly monotone simple algebra $(\mathcal{N}, >_{\mathbb{N}})$ over $\mathbb{N}$ assigning polynomials $f_{\mathcal{N}}$ to every $f \in \mathcal{F}$. In the sequel we refer to this restricted version of GKBO by PKBO. We first show that there are TRSs that can be shown terminating by PKBO but not by polynomial interpretations.

Example 18. Consider the SRS $\mathcal{R}$ consisting of the rules

$$\mathsf{a}(\mathsf{b}(x)) \to \mathsf{b}(\mathsf{b}(\mathsf{a}(x))) \qquad \mathsf{a}(\mathsf{c}(x)) \to \mathsf{c}(\mathsf{c}(\mathsf{a}(x))) \qquad \mathsf{b}(\mathsf{c}(x)) \to \mathsf{c}(\mathsf{b}(x))$$

The GKBO with $\mathsf{a}_{\mathcal{N}}(x) = 3x{+}1$, $\mathsf{b}_{\mathcal{N}}(x) = \mathsf{c}_{\mathcal{N}}(x) = x{+}1$ and $\mathsf{b} \succ \mathsf{c}$ is compatible with $\mathcal{R}$. To orient the first two rules by a polynomial interpretation $\mathcal{N}$, $\mathsf{b}_{\mathcal{N}}$ and $\mathsf{c}_{\mathcal{N}}$ must be linear and monic. But then the last rule is not orientable.

The open question deals with the reverse direction, i.e., can any TRS that admits a compatible polynomial interpretation also be shown terminating by PKBO? Polynomial interpretations are monotone and hence also weakly monotone. Consequently polynomials can only exceed PKBO with respect to power if a *non-simple* interpretation can be enforced. Below we show that linear interpretations cannot enforce such an interpretation, in contrast to (non-linear) polynomial interpretations.

Linear Interpretations: In this subsection we refer to a PKBO where all interpretation functions are linear by LKBO. In the sequel we show that any linear interpretation can be transformed into a linear interpretation where all interpretation functions are simple.

Let $\mathcal{N}$ be a linear interpretation. For each $f_{\mathcal{N}}(x_1, \dots, x_n) = f_1x_1 + \cdots + f_nx_n + f_0$ and $m \in \mathbb{N}$ let $f_{\mathcal{N}_m}(x_1, \dots, x_n) = f_1x_1 + \cdots + f_nx_n + m \cdot f_0$. Let α_0 be the assignment such that $\alpha_0(x) = 0$ for all variables x.

Lemma 19. *If $s >_{\mathcal{N}} t$ then $[\alpha]_{\mathcal{N}_m}(s) > [\alpha]_{\mathcal{N}_m}(t) + (m-1)$ holds for all α.*

Proof. We first prove

$$[\alpha]_{\mathcal{N}_m}(t) = [\alpha]_{\mathcal{N}}(t) + (m-1)[\alpha_0]_{\mathcal{N}}(t) \tag{3}$$

by induction on t. In the base case $t \in \mathcal{V}$ and

$$[\alpha]_{\mathcal{N}_m}(t) = \alpha(t) = \alpha(t) + (m-1) \cdot 0 = [\alpha]_{\mathcal{N}}(t) + (m-1)[\alpha_0]_{\mathcal{N}}(t)$$

In the step case let $t = f(t_1, \dots, t_n)$. Then

$$\begin{aligned}
[\alpha]_{\mathcal{N}_m}(f(t_1, \dots, t_n)) &= m \cdot f_0 + \sum_{1 \leqslant i \leqslant n} f_i \cdot [\alpha]_{\mathcal{N}_m}(t_i) \\
&= m \cdot f_0 + \sum_{1 \leqslant i \leqslant n} f_i \cdot ([\alpha]_{\mathcal{N}}(t_i) + (m-1)[\alpha_0]_{\mathcal{N}}(t_i)) \\
&= [\alpha]_{\mathcal{N}}(t) + (m-1)[\alpha_0]_{\mathcal{N}}(t)
\end{aligned}$$

where the induction hypothesis is applied in the second step.

From the assumption $s >_{\mathcal{N}} t$ we obtain $[\alpha]_{\mathcal{N}}(s) > [\alpha]_{\mathcal{N}}(t)$ and in particular $[\alpha_0]_{\mathcal{N}}(s) \geqslant [\alpha_0]_{\mathcal{N}}(t) + 1$. Hence

$$\begin{aligned}
[\alpha]_{\mathcal{N}_m}(s) &= [\alpha]_{\mathcal{N}}(s) + (m-1)[\alpha_0]_{\mathcal{N}}(s) > [\alpha]_{\mathcal{N}}(t) + (m-1)[\alpha_0]_{\mathcal{N}}(s) \\
&\geqslant [\alpha]_{\mathcal{N}}(t) + (m-1)([\alpha_0]_{\mathcal{N}}(t) + 1) = [\alpha]_{\mathcal{N}_m}(t) + (m-1)
\end{aligned}$$

where the two equality steps follow from (3). □

Definition 20. *Let $\mathcal{A}$ be an algebra. We define the algebra $\mathcal{A}'$ to be as $\mathcal{A}$ but for each function symbol g with $[\alpha_0]_{\mathcal{A}}(g(x_1,\dots,x_n)) = 0$ we define $g_{\mathcal{A}'}(x_1,\dots,x_n) = g_{\mathcal{A}}(x_1,\dots,x_n)+1$. For a finite TRS $\mathcal{R}$ let $M := \max\{[\alpha_0]_{\mathcal{A}'}(r) \mid \ell \to r \in \mathcal{R}\}+1$.*

Lemma 21. *If a finite TRS is compatible with $\mathcal{N}$ then it is compatible with $(\mathcal{N}_M)'$.*

Proof. A straightforward induction proof shows that any term t satisfies

$$[\alpha]_{\mathcal{N}_m}(t) + [\alpha_0]_{\mathcal{N}'}(t) \geqslant [\alpha]_{(\mathcal{N}_m)'}(t) \tag{4}$$

Compatibility yields $\ell >_{\mathcal{N}} r$ for every $\ell \to r \in \mathcal{R}$. The claim follows from $[\alpha]_{(\mathcal{N}_M)'}(\ell) \geqslant [\alpha]_{\mathcal{N}_M}(\ell) > [\alpha]_{\mathcal{N}_M}(r)+M-1 \geqslant [\alpha]_{\mathcal{N}_M}(r)+[\alpha_0]_{\mathcal{N}'}(r) \geqslant [\alpha]_{(\mathcal{N}_M)'}(r)$ where Lemma 19 is used in the second step, $M > [\alpha_0]_{\mathcal{N}'}(r)$ in the third step, and the last step is an application of (4). □

As $(\mathcal{N}_M)'$ is simple we obtain the following result from Lemma 21.

Theorem 22. *If a finite TRS is compatible with a linear interpretation then it is compatible with an LKBO.*

Proof. Let $\mathcal{R}$ be a finite TRS that is compatible with a linear interpretation $\mathcal{N}$ and by Lemma 21 also with $(\mathcal{N}_M)'$. By construction $(\mathcal{N}_M)'$ is simple and all its interpretation functions are linear. Since $(\mathcal{N}_M)'$ is monotone it is also weakly monotone. Hence based on $(\mathcal{N}_M)'$ we have $\ell \succ_{\text{gkbo}} r$ for every $\ell \to r \in \mathcal{R}$ and empty precedence $\succ$. □

Non-linear Interpretations: To show that PKBO does not subsume polynomial interpretations we give a TRS that can be shown terminating by a polynomial interpretation but not by PKBO. The reason for failure is that compatibility with PKBO can enforce interpretation functions that are not simple.

Theorem 23. *The TRS $\mathcal{R}$ consisting of $\mathsf{s}(x) \to \mathsf{t}(\mathsf{t}(\mathsf{t}(x)))$ and the rules*

$$\begin{array}{lccc}
\mathcal{R}_1 & \mathsf{f}(0) \to 0 & \mathsf{f}(\mathsf{s}(0)) \to \mathsf{s}(0) & \mathsf{f}(\mathsf{s}^2(0)) \to \mathsf{s}^6(0) \\
\mathcal{R}_1 & \mathsf{s}^2(0) \to \mathsf{f}(0) & \mathsf{s}^3(0) \to \mathsf{f}(\mathsf{s}(0)) & \mathsf{s}^8(0) \to \mathsf{f}(\mathsf{s}^2(0)) \\
\mathcal{R}_2 & \mathsf{g}(x) \to \mathsf{h}(x,x) & \mathsf{s}(x) \to \mathsf{h}(x,0) & \mathsf{s}(x) \to \mathsf{h}(0,x) \\
\mathcal{R}_3 & \mathsf{f}(\mathsf{g}(x)) \to \mathsf{g}(\mathsf{g}(\mathsf{f}(x))) & \mathsf{g}(\mathsf{s}(x)) \to \mathsf{s}(\mathsf{s}(\mathsf{g}(x))) & \mathsf{h}(\mathsf{f}(x),\mathsf{g}(x)) \to \mathsf{f}(\mathsf{s}(x))
\end{array}$$

can be shown terminating by a polynomial interpretation but not by PKBO.

Proof. The TRS $\mathcal{R}$ is compatible with the following polynomial interpretation:

$$\begin{array}{ccc}
\mathsf{f}_{\mathcal{N}}(x) = 2x^2 - x + 1 & \mathsf{s}_{\mathcal{N}}(x) = x + 1 & \mathsf{g}_{\mathcal{N}}(x) = 4x + 5 \\
\mathsf{h}_{\mathcal{N}}(x,y) = x + y & \mathsf{t}_{\mathcal{N}}(x) = x & 0_{\mathcal{N}} = 0
\end{array}$$

To see that $\mathcal{R}$ cannot be shown terminating by PKBO we adopt the idea from [16] where the shape and the coefficients of a compatible polynomial interpretation can be determined by the rewrite rules in $\mathcal{R}_1 \cup \mathcal{R}_2 \cup \mathcal{R}_3$. However, in our setting

we have to re-inspect the results from [16] since PKBO allows $\ell \geqslant_{\mathcal{A}} r$ in contrast to a polynomial interpretation which requires $\ell >_{\mathcal{A}} r$ for all rules. Furthermore monotonicity has to be replaced by weak monotonicity and the algebra has to be simple.

Next we investigate which interpretation functions are enforced by the rules in $\mathcal{R}$. Inspecting [16, Lemma 16] using the first two rules from $\mathcal{R}_3$ we obtain that $\mathsf{s}_{\mathcal{N}}$ and $\mathsf{g}_{\mathcal{N}}$ must be linear. Moreover $\mathsf{s}_{\mathcal{N}}(x) = x + d$ for some $d \in \mathbb{N}$ and $\mathsf{f}_{\mathcal{N}}$ is not linear. Since $\mathcal{N}$ must be simple we have $d > 0$. From [16, Lemma 21] and $\mathcal{R}_2$ we obtain $\mathsf{h}_{\mathcal{N}}(x, y) = x + y + p$. Now [16, Lemma 20] and the last rule of $\mathcal{R}_3$ limit the degree of $\mathsf{f}_{\mathcal{N}}$ to at most two.

Next we focus on [16, Lemma 18]. Let $z = \mathsf{0}_{\mathcal{N}}$. The last rule of $\mathcal{R}_1$ yields

$$z + 8d \geqslant f_{\mathcal{N}}(z + 2d)$$

Note that $\mathsf{f}(x) >_{\mathcal{N}} x$ since $\mathsf{f}_{\mathcal{N}}$ must be simple. Since the degree of $\mathsf{f}_{\mathcal{N}}$ is two we have $\mathsf{f}_{\mathcal{N}}(x) = ax^2 + bx + c$ with $a \geqslant 1$. For well-definedness of $\mathsf{f}_{\mathcal{N}}$ we need $a \geqslant -b$ and $c \geqslant 0$. Next we show that $d \leqslant 2$. To this end we assume $d \geqslant 3$ and arrive at a contradiction. Now

$$\begin{aligned}\mathsf{f}_{\mathcal{N}}(z + 2d) &= a(z^2 + 4zd + 4d^2) + b(z + 2d) + c = f(z) + a(4zd + 4d^2) + 2bd \\ &> z + a(4zd + 4d^2) + 2bd \geqslant z + 4ad^2 + 2bd = z + d(4ad + 2b) \\ &= z + d((4d - 2)a + 2(a + b)) \geqslant z + 10d\end{aligned}$$

which is a contradiction to the constraint $z + 8d \geqslant \mathsf{f}_{\mathcal{N}}(z + 2d)$ from above. Hence $d \leqslant 2$. But then $\mathsf{s}(x) \to \mathsf{t}(\mathsf{t}(\mathsf{t}(x)))$ requires $\mathsf{t}_{\mathcal{N}}(x) = x$. Hence PKBO cannot prove termination of the TRS $\mathcal{R}$. □

6 Implementation and Evaluation

To establish a termination proof by KBO the task is to search for suitable weights and a precedence. For efficiently finding a compatible KBO by linear programming we refer to [23]. TKBO with finite weights can easily be encoded in non-linear integer arithmetic (similarly to [23]) for which powerful but (necessarily) incomplete tools exist [24]. Since these tools are typically overflow-safe the problems sketched in [12,13] do not appear in our setting (termination proving).

In the remainder of this section we sketch how one can implement a version of GKBO using transfinite ordinal weights (below ω^ω) with the standard addition and multiplication of ordinals. This is sound by Theorem 15, provided the interpretation functions are weakly monotone and simple. We restrict ourselves to string rewrite systems and interpretations of the (canonical) form

$$f_{\mathcal{O}}(x) = x \cdot f' + \omega^d \cdot f_d + \cdots + \omega^1 \cdot f_1 + f_0 \tag{5}$$

where $f', f_d, \ldots, f_0 \in \mathbb{N}$. As illustration, we abstractly encode the rule

$$\mathsf{a}(\mathsf{b}(x)) \to \mathsf{b}(\mathsf{a}(\mathsf{a}(x)))$$

with $d = 1$. For the left-hand side we get

$$x \cdot b' \cdot a' + \omega^1 \cdot b_1 \cdot a' + b_0 \cdot a' + \omega^1 \cdot a_1 + a_0$$

which can be written in the canonical form

$$x \cdot b' \cdot a' + \omega^1 \cdot (b_1 \cdot a' + a_1) + (a_1 > 0 \,?\, 0 : b_0 \cdot a') + a_0$$

where the $(\cdot\,?\,\cdot:\cdot)$ operator implements if-then-else, i.e., if a_1 is greater than zero then the summand $b_0 \cdot a'$ vanishes. To determine whether

$$x \cdot l' + \omega^1 \cdot l_1 + l_0 \geqslant x \cdot r' + \omega^1 \cdot r_1 + r_0$$

for all values of x, we use the criterion $l' \geqslant r' \wedge (l_1 > r_1 \vee (l_1 = r_1 \wedge l_0 \geqslant r_0))$. Finally, $f' \geqslant 1 \wedge (f_1 \geqslant 1 \vee f_0 \geqslant 1)$ ensures that the interpretation $f_{\mathcal{O}}$ is simple while the interpretation functions are then weakly monotone for free. Hence the search for suitable weights, subterm coefficients, and the precedence can be encoded in non-linear integer arithmetic, similar as in [24].

Termination: For termination analysis we considered the 1416 TRSs and 720 SRSs from TPDB 7.0.2.[3] The experiments[4] have been performed single-threaded with T$_{\text{T}}$T$_2$ [11]. The leftmost part of Table 1 shows how many TRSs can be proved terminating (column yes) and the average duration of finding a termination proof (column time) in seconds.[5] Coefficients for polynomials and weights/subterm coefficients for (T)KBO have been represented by at most six bits (to get a maximum number of termination proofs). As termination criteria we considered Theorem 2 (row KBO) and Theorem 7 using finite weights and subterm coefficients (row TKBO). We list the data for linear interpretations (POLY) and the lexicographic path order (LPO) for reference. For SRSs the entry TKBO$_\omega$ corresponds to an implementation of Theorem 15 using interpretation functions as in (5) with $d = 1$ while for KBO$_\omega$ in addition we fixed $f' = 1$ in (5). Different values for d did not increase the number of systems proved terminating in this setting. However, it is possible to construct systems that need an arbitrarily large d (based on the derivational complexity of the system). In both categories (TRS and SRS) TKBO subsumes KBO and POLY (which is no surprise in light of Theorem 22). Hence the additional systems stem from LPO.

Ordered Completion: Ordered completion [1] is one of the most frequently used calculi in equational theorem proving. Classical ordered completion tools require a reduction order as input that can be extended to a total order on ground terms. This parameter is critical for the success of a run, but a suitable choice is hardly predictable in advance. *Ordered multi-completion with termination tools* [22] addresses this challenge by employing automatic termination tools and exploring different orientations in parallel, instead of sticking to one fixed ordering. This

[3] http://termcomp.uibk.ac.at/status/downloads/tpdb-7.0.2.tar.gz
[4] Details are available from http://colo6-c703.uibk.ac.at/ttt2/tkbo/
[5] This includes the "start-up time" of T$_{\text{T}}$T$_2$ which is around 0.3 seconds.

Table 1. Termination and Ordered Completion

	Termination				Ordered Completion		
	1416 TRSs		720 SRSs		42 TPTP theories		
method	yes	time	yes	time	yes	time	tc
KBO/KBO$_\omega$	107/-	0.5	33/34	0.5/0.7	31/-	16	19%
TKBO/TKBO$_\omega$	192/-	0.9	43/44	1.7/2.9	34/-	56	35%
POLY	149	0.9	22	1.6	15	4	70%
LPO	159	0.5	5	0.5	26	37	7%
$\sum$	262	-	45	-	34		

approach is implemented in the tool OMKBTT [22]. However, only termination techniques that guarantee *total termination* [5] are applicable in order to obtain a TRS that is indeed ground-complete. In practice, applicable termination techniques are thus restricted to classical reduction orders such as LPO, KBO or polynomial interpretations. We thus compared the power of OMKBTT using TKBO besides other reduction orders on a test set of 42 theories underlying TPTP [20]. Indeed TKBO is able to produce ground-complete systems for more problems than any of the other reduction orders. The right part of Table 1 shows that TKBO significantly extends the class of orientable TRSs, although more time is required (column time). The column tc indicates the percentage of the execution time spent on termination checks.

7 Conclusion

In this paper we considered three variants of the Knuth-Bendix order and showed that some extensions do not add power (as far as termination proving of finite TRSs is considered) while others do. We have implemented the finite version of TKBO [12,13] as an SMT problem in non-linear integer arithmetic. Since our solver uses arbitrary precision arithmetic, overflows (as reported in [12,13]) are not an issue. However, since already standard KBO can demand arbitrarily large weights (see [23, Example 2]) overflows are not specific to TKBO (as the discussions in [12,13] convey). We have also implemented a KBO using ordinal weights, which has been identified as one challenge in [12]. Also Vampire [17] uses ordinal numbers (see [12, Section 7]), but only for weights of predicate symbols. Since they occur at the root only no ordinal arithmetic is needed but only comparison. Hence the same effect could be achieved by allowing a (quasi-)precedence on predicate symbols.

Acknowledgments. We thank the anonymous reviewers, Bertram Felgenhauer, and Georg Moser for helpful comments.

References

1. Bachmair, L., Dershowitz, N., Plaisted, D.: Completion without failure. In: Kaci, H.A., Nivat, M. (eds.) Resolution of Equations in Algebraic Structures 1989. Rewriting Techniques of Progress in Theoretical Computer Science, vol. 2, pp. 1–30. Academic Press (1989)

2. Dershowitz, N.: Orderings for term-rewriting systems. TCS 17, 279–301 (1982)
3. Dershowitz, N.: Termination of rewriting. J. Symb. Comp. 3(1-2), 69–116 (1987)
4. Dick, J., Kalmus, J., Martin, U.: Automating the Knuth Bendix ordering. AI 28, 95–119 (1990)
5. Ferreira, M., Zantema, H.: Total termination of term rewriting. AAECC 7(2), 133–162 (1996)
6. Geser, A.: An improved general path order. AAECC 7(6), 469–511 (1996)
7. Goodstein, R.L.: On the restricted ordinal theorem. J. Symb. Log. 9(2), 33–41 (1944)
8. Hong, H., Jakuš, D.: Testing positiveness of polynomials. JAR 21(1), 23–38 (1998)
9. Jech, T.: Set Theory. Springer, Heidelberg (2002)
10. Knuth, D., Bendix, P.: Simple word problems in universal algebras. In: Leech, J. (ed.) Computational Problems in Abstract Algebra, pp. 263–297. Pergamon Press, New York (1970)
11. 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)
12. Kovács, L., Moser, G., Voronkov, A.: On Transfinite Knuth-Bendix Orders. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 23. LNCS (LNAI), vol. 6803, pp. 384–399. Springer, Heidelberg (2011)
13. Ludwig, M., Waldmann, U.: An Extension of the Knuth-Bendix Ordering with LPO-Like Properties. In: Dershowitz, N., Voronkov, A. (eds.) LPAR 2007. LNCS (LNAI), vol. 4790, pp. 348–362. Springer, Heidelberg (2007)
14. McCune, B.: Otter 3.0 reference manual and guide. Technical Report ANL-94/6, Argonne National Laboratory (1994)
15. Middeldorp, A., Zantema, H.: Simple termination of rewrite systems. TCS 175(1), 127–158 (1997)
16. Neurauter, F., Middeldorp, A., Zankl, H.: Monotonicity Criteria for Polynomial Interpretations over the Naturals. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS (LNAI), vol. 6173, pp. 502–517. Springer, Heidelberg (2010)
17. Riazanov, A., Voronkov, A.: The design and implementation of VAMPIRE. AI Commun. 15(2-3), 91–110 (2002)
18. Steinbach, J.: Extensions and Comparison of Simplification Orders. In: Dershowitz, N. (ed.) RTA 1989. LNCS, vol. 355, pp. 434–448. Springer, Heidelberg (1989)
19. Steinbach, J., Zehnter, M.: Vademecum of polynomial orderings. Technical Report SR-90-03, Universität Kaiserslautern (1990)
20. Sutcliffe, G.: The TPTP problem library and associated infrastructure: The FOF and CNF parts, v3.5.0. JAR 43(4), 337–362 (2009)
21. TeReSe: Term Rewriting Systems. Cambridge Tracts in Theoretical Computer Science, vol. 55. Cambridge University Press (2003)
22. Winkler, S., Middeldorp, A.: Termination Tools in Ordered Completion. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS (LNAI), vol. 6173, pp. 518–532. Springer, Heidelberg (2010)
23. Zankl, H., Hirokawa, N., Middeldorp, A.: KBO orientability. JAR 43(2), 173–201 (2009)
24. Zankl, H., Middeldorp, A.: Satisfiability of Non-linear (Ir)rational Arithmetic. In: Clarke, E.M., Voronkov, A. (eds.) LPAR-16. LNCS (LNAI), vol. 6355, pp. 481–500. Springer, Heidelberg (2010)

r-TuBound: Loop Bounds for WCET Analysis (Tool Paper)

Jens Knoop, Laura Kovács, and Jakob Zwirchmayr*

TU Vienna

Abstract. We describe the structure and the usage of a new software tool, called r-TuBound, for deriving symbolic loop iteration bounds in the worst-case execution time (WCET) analysis of programs. r-TuBound implements algorithms for pattern-based recurrence solving and program flow refinement, and it was successfully tested on a wide range of examples. The purpose of this article is to illustrate what r-TuBound can do and how it can be used to derive the WCET of programs.

1 Introduction

One of the most challenging tasks in the worst-case execution time (WCET) analysis of programs with loops comes with the task of providing precise bounds, called loop bounds, over the number of loop iterations.

In this article we describe the r-TuBound tool for deriving automatically loop bounds in the WCET analysis of programs. Several software packages for this purpose have already been developed in the past and can be classified within two categories. One line of research uses powerful symbolic computation algorithms to derive loop bounds (see e.g. [1]), but makes very little, if any, progress in integrating these loop bounds in the program analysis environment of WCET. Another line of research makes use of abstract interpretation based static analysis techniques to provide good WCET estimates; however, often loop bounds are assumed to be a priori given, in part, by the user (see e.g. [10,4,7]).

The philosophy of our tool is somewhat in the middle of these two research trends. Rather than a package integrating powerful symbolic computation algorithms, r-TuBound uses pattern-based recurrence solving (Section 2.4) for a restricted, yet in practice quite general class of programs. Loop bounds are inferred to be satisfiable instances of a system of arithmetic constraints over the loop iteration variable. r-TuBound can thus derive non-trivial loop bounds, but not only that. The inferred loop bounds are further used in the WCET analysis of programs. To make the loop bound computation techniques scale for the WCET analysis, r-TuBound translates loops with nested conditionals into loops without conditionals (Section 2.3) using SMT reasoning in conjunction with program flow refinement. When evaluated on a large class of benchmarks, our experiments indicate the applicability of r-TuBound in the WCET analysis of programs (Section 3).

* This research is supported by the CeTAT project of TU Vienna. The second author is supported by an FWF Hertha Firnberg Research grant (T425-N23). This research is partly supported by the FWF National Research Network RiSE (S11410-N23) and the WWTF PROSEED grant (ICT C-050).

N. Bjørner and A. Voronkov (Eds.): LPAR-18, LNCS 7180, pp. 435–444, 2012.

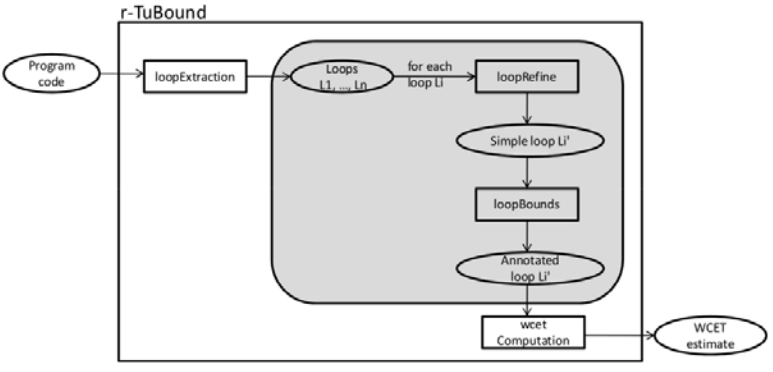


Fig. 1. The r-TuBound tool

The goal of this article is to describe *what* r-TuBound can do, explain how to *use it*, and give *implementation details* about the structure of r-TuBound. We describe only briefly *how* r-TuBound obtains its results and refer to [6] for more details.

Implementation and Availability. r-TuBound is implemented in C++ and the Termite library [9] for Prolog. It is available at `www.complang.tuwien.ac.at/jakob/tubound/`. All results presented in this article were obtained on a machine with a 2.53GHZ Intel Core i5 CPU and 4GB of RAM. To improve readability, we employed minor simplifications over the input and output format of the examples discussed in the article.

2 r-TuBound: Tool Description and Usage

2.1 Workflow

The overall workflow of r-TuBound is given in Figure 1.

Inputs to r-TuBound are arbitrary C/C++ programs. In the first part of r-TuBound, the input program is parsed and analysed. As a result all loops and unstructured goto statements of the input code are extracted. To this end, various static analysis techniques from [10] are applied, such as parsing by the EDG C/C++ frontend, building the abstract syntax trees and the control-flow graph of the program, interval analysis over program variables, and points-to analysis. Further, the extracted loops and goto statements are rewritten, whenever possible, into the format given in equation (M). Doing so requires, among others, the following steps: rewriting while-loops into equivalent for-loops, rewriting if-statements into if-else statements, translating multi-path loops with abrupt termination into loops without abrupt termination, and approximating non-deterministic variable assignments. The aforementioned steps for parsing, analysing and preprocessing C/C++ programs are summarized in the `loopExtraction` part of Figure 1. If a program loop cannot be converted into equation (M) by the `loopExtraction` part of r-TuBound, r-TuBound does not compute a loop bound and hence the WCET computation step of r-TuBound will fail.

Next, the loops extracted in the format given in equation (M) are analysed and translated into equation (S), required by the loop bound computation engine of r-TuBound

```
1 void test() {
2  int i, a[16];
3  i = 0;
4  while (i < 100) {
5   if (a[i] > 0) i = i * 2 + 2;
6   else i = i * 2 + 1;
7   i = i + 1; }}
```

Fig. 2. Input C program `test.c`

```
1 void test() {
2  int i, a[16];
3  for (i = 0; i < 100; i = i + 1)
4   if (a[i] > 0) i = i * 2 + 2;
5   else i = i * 2 + 1;
6 }
```

Fig. 3. C program `test.c` satisfying the format requirements of equation (M)

(see Section 2.3). This step is performed in the `loopRefine` part of Figure 1. As a result of `loopRefine`, the multi-path loops of equation (M) are translated into the simple loops presented in equation (S).

For deriving loop bounds in the `loopBounds` step of Figure 1, each loop is analysed separately and bounds are inferred using recurrence solving. The computed loop bounds are added as annotations to the loops and are further used to calculate the WCET of the input program, as illustrated in the `wcetComputation` engine of Figure 1. Outputs of r-TuBound are thus the WCET of the input programs. For simplicity, in the current version of r-TuBound, the execution time of a processor cycle is assumed to be of 20 nanoseconds per cycle.

Let us note that in the `wcetComputation` and `loopExtraction` steps of Figure 1, r-TuBound makes use of the static analysis framework of [10]. The distinctive features of r-TuBound, and the main contributions of this article, come with the *automatic inference of loop bounds* (steps `loopRefine` and `loopBounds` of Figure 1). To this end, r-TuBound implements pattern-based recurrence solving and program flow refinement techniques, and integrates these techniques in the WCET analysis of programs.

To invoke r-TuBound, one uses the following command.

Command 2.1 : `rTuBound program.c`

Input: C/C++ program
Output: WCET of `program.c`
Assumption: Loops of `program.c` are or can be transformed in equation (M)

EXAMPLE 2.1 Consider the `test.c` program given in Figure 2. The WCET returned by r-TuBound is listed below.

```
Input: rTuBound test.c
Output: 26
```

The WCET of `test.c` is thus inferred to be of 26 time units. For doing so, the while-loop of Figure 2 is first translated into equation (M), as given in Figure 3. Next, in the WCET computation of Figure 3, we assume for simplicity that all program expressions take one time unit to execute. Therefore, the execution of one iteration of the loop between lines 3-5 of Figure 3 takes 4 time units: 1 unit each to check the boolean conditions of the loop and of the if-statement, 1 unit to execute the assignment statement $i = i+1$ from the loop header, and 1 unit to execute the assignment from one branch of the if-statement, depending on the boolean test $a[i] > 0$. Further, r-TuBound computes

```
for (i = 0; i < 100; i = i + 1) {
  if (a[i] > 0) i = i * 2 + 2;
  else i = i * 2 + 1;
}
```

Fig. 4. Multi-path loop from `test.c` (`mpath1.c`)

```
for (i = 0; i < 100;
     i = 2 * i + 3) {
}
```

Fig. 5. Over-approximation (`simple1.c`)

```
for (i = 0; i < 100;
     i = 2 * i + 3) {
  #wcet_loopbound(6)
}
```

Fig. 6. Annotated with loop bound (`annot1.c`)

6 to be the loop bound, from which it infers that the execution of all loop iterations takes altogether 24 time units. As the initialisation of the loop counter i, as well as the last test of the loop condition takes one unit each, the WCET for `test.c` is hence computed to be 26 time units.

2.2 Loop Restrictions

The syntax of program loops that can be handled by the loop bound computation part of r-TuBound is given below.

$$
\begin{array}{l}
\texttt{for}\,(\mathtt{i} = \mathtt{a};\ \mathtt{i} \diamond \mathtt{b};\ \mathtt{i} = \mathtt{c} * \mathtt{i} + \mathtt{d})\ \{ \\
\quad \mathtt{i} = \mathtt{f}_0(\mathtt{i}); \\
\quad \texttt{if}\ \ (\mathtt{g}_1)\ \ \mathtt{i} = \mathtt{f}_1(\mathtt{i});\ \ \texttt{else}\ \ \mathtt{i} = \mathtt{f}_2(\mathtt{i}); \\
\quad \texttt{if}\ \ (\mathtt{g}_2)\ \ \mathtt{i} = \mathtt{f}_3(\mathtt{i});\ \ \texttt{else}\ \ \mathtt{i} = \mathtt{f}_4(\mathtt{i}); \\
\quad \ldots; \\
\quad \texttt{if}\ \ (\mathtt{g}_m)\ \ \mathtt{i} = \mathtt{f}_{2\mathtt{m}-1}(\mathtt{i});\ \ \texttt{else}\ \ \mathtt{i} = \mathtt{f}_{2\mathtt{m}}(\mathtt{i}); \\
\}
\end{array}
\qquad \text{(M)}
$$

where $\diamond \in \{<, >\}$, $g_1, \ldots, g_m$ are boolean expressions, and a, b, c, d are symbolic integer-valued constants such that a, b, c, d do not depend on i and $c > 0$. The expressions f_s, with $s = 0, \ldots, 2m$, are non-constant linear integer arithmetic functions over i; that is, $f_s(i) = c_s * i + d_s$ where c_s, d_s are symbolic integer-valued constants that do not depend on i and $c_s > 0$. Moreover, either $i < c * i + d$ and $i \leq f_s(i)$ hold for $i \geq a$, or $i > c * i + d$ and $i \geq f_s(i)$ are valid for $i \leq a$.

Let us make the following observation over the restrictions of (M). As $c > 0$ and $c_s > 0$ in equation (M), the functions $i \mapsto c * i + d$ and $f_0(i), \ldots, f_{2m}(i)$ are all monotonically increasing. Therefore, when translating arbitrary loops into the format of equation (M) in the `loopExtract` part of r-TuBound, we proceed as follows. To ensure that $i < c * i + d$ and $i \leq f_s(i)$ hold it suffices to check whether $i < c * i + d$ and $i \leq f_s(i)$ are valid for $i = a$.

In what follows, we fix some terminology used in the rest of the article. In the sequel whenever we write *loop* (M) or (M) we refer to a multi-path loop as given in equation (M). We refer to the variable i in equation (M) as the *loop counter* or the *loop iteration variable*, whereas the assignment $i = c * i + d$ in the loop header is called the *update expression* of the loop. The constant a is called the *initial value* of i. We consider a loop a *simple loop* if there is only one execution path through the body, i.e. if $m = 0$ in equation (M). Otherwise, if $m > 0$, the loop (M) is said to be a *multi-path loop*. Finally, the assignments $i = f_s(i)$, with $s = 1, \ldots, 2m$, are called the *conditional updates* of the loop.

2.3 Program Flow Refinement

Given a multi-path loop (M), the `loopRefine` part of r-TuBound translates (M) into a simple loop, such that the loop bound of the simple loop is also a loop bound of the multi-path loop (M).

To this end, the multi-path behavior of (M) is safely over-approximated, as follows.

The boolean conditions $g_1, \ldots, g_m$ are first ignored, yielding thus a loop body with non-deterministic conditional statements.

Next, for each $s = 1, \ldots, m$, we are left with choosing $k_s \in \{2s-1, 2s\}$ such that

$$f_{k_s}(i) \leq f_{2s-1}(i) \text{ and } f_{k_s}(i) \leq f_{2s}(i) \text{ for every } i \triangleright a, \tag{1}$$

where $\triangleright \in \{\leq, \geq\}$ is defined as follows:
- $\triangleright$ is $\geq$ if $i < c * i + d$ in (M);
- $\triangleright$ is $\leq$ if $i > c * i + d$ in (M).

The conditional update $i = f_{k_s}(i)$ determined by (1) yields thus the minimal increase, respectively the minimal decrease, over i after an arbitrary execution of the if-statement with ignored test condition g_s. Therefore, by replacing each if-statement with the corresponding $i = f_{k_s}(i)$ at every iteration of (M), a safe loop bound for (M) can be derived.

However, a $k_s \in \{2s-1, 2s\}$ might not always be computed from (1), as (1) needs to hold for *every* $i \geq a$ (respectively, for *every* $i \leq a$). That is, the existence of $k_s \in \{2s-1, 2s\}$ such that (1) is valid depends crucially on the initial value of a. To overcome this limitation, we proceed as follows. Whenever $k_s \in \{2s-1, 2s\}$ cannot be computed from (1), we take $f_{k_s}(i) = i$. Based on the restrictions of equation (M), we clearly have $f_{k_s}(i) = i \leq f_{2s-1}(i)$ and $f_{k_s}(i) = i \leq f_{2s}(i)$ for every $i \geq a$ (respectively, $f_{k_s}(i) = i \geq f_{2s-1}(i)$ and $f_{k_s}(i) = i \geq f_{2s}(i)$ for every $i \leq a$). That is, $i = f_{k_s}(i)$ yields a smaller increase (respectively, decrease) over i than any branch of the if-statement with ignored test condition g_s. Therefore, if k_s cannot be computed from (1), we define $f_{k_s}(i) = i$ and replace the if-statement with the ignored test condition g_s by $i = f_{k_s}(i)$. The loop bound for (M) is thus safely over-approximated.

Based on the above observations, equation (M) is translated into the simple loop (T), given below.

$$\begin{array}{l} \texttt{for}\ (\texttt{i} = \texttt{a};\ \texttt{i} \diamond \texttt{b};\ \texttt{i} = \texttt{c} * \texttt{i} + \texttt{d})\ \{ \\ \qquad \texttt{i} = \texttt{f}_0(\texttt{i}); \texttt{i} = \texttt{f}_{\texttt{k}_1}(\texttt{i}); \ldots; \texttt{i} = \texttt{f}_{\texttt{k}_\texttt{m}}(\texttt{i})\} \end{array} \tag{T}$$

Let us write $\phi(i) = c * i + d$, and let $\circ$ denote the standard operation of function composition. Using this notation, (T) is further rewritten into the simple loop:

$$\texttt{for}\ (\texttt{i} = \texttt{a};\ \texttt{i} \diamond \texttt{b};\ \texttt{i} = (\texttt{f}_{\texttt{k}_\texttt{m}} \circ \cdots \circ \texttt{f}_{\texttt{k}_1} \circ \texttt{f}_0 \circ \phi)(\texttt{i}))\ \{\} \tag{S}$$

In the sequel whenever we write *loop* (S) or (S) we refer to a simple loop as given in equation (S).

Note that as linear functions are closed under composition, $(f_{k_m} \circ \cdots \circ f_{k_1} \circ f_0 \circ \phi)(i)$ yields a non-constant linear integer arithmetic function over i in (S).

The behavior of `loopRefine` is summarised below.

```
for (i = 0; i < 100; i = i + 1) {
  if (a[i] > 0) i = i * 3 + 2;
  else i = i * 2 + 10;
}
```

Fig. 7. A multi-path loop (`mpath2.c`)

```
for (i = 0; i < 100;
     i = i + 1) {
}
```

Fig. 8. Over-approximation (`simple2.c`)

```
for (i = 0; i < 100;
     i = i + 1) {
  #wcet_loopbound(100)
}
```

Fig. 9. Annotated loop (`annot2.c`)

Command 2.2 : `loopRefine loop.c`

Input: Loop
Output: a simple loop as in (S)
Assumption: `loop.c` as in (M)

EXAMPLE 2.2 For translating the multi-path loop given in Figure 4, `loopRefine` infers that the conditional update corresponding to the else-branch of the conditional statement of `mpath1.c` yields the minimal update over i. The multi-path loop of Figure 4 is thus rewritten into the simple loop given in Figure 5, as listed below.

```
Input: loopRefine mathp1.c

Output: simple1.c
```

EXAMPLE 2.3 Consider now the multi-path loop given in Figure 7. Note that $3*i+2 \leq 2*i+10$ does not hold for every $i \geq 0$. Similarly, $2*i+10 \leq 3*i+2$ does not hold for every $i \geq 0$. Hence, no conditional update can be chosen by `loopRefine` as the update yielding the minimal increase over i. Therefore, the conditional statement of Figure 7 is over-approximated by the update $i = i$, and Figure 7 is rewritten into Figure 8, as shown below.

```
Input: loopRefine mpath2.c

Output: simple2.c
```

The task of choosing k_s in (1) such that $f_{k_s}(i)$ yields the minimal increase over i, is encoded in r-TuBound as a set of SMT queries. For doing so, we interfaced `loopRefine` with the Boolector SMT solver [2]. To this end, for each variable in the program, a bit vector variable is introduced by `loopRefine`. An array is used to model the values of the variables involved. This representation allows `loopRefine` to capture the loop behavior in a symbolic manner, by using symbolic values to model the updates to the loop counter.

2.4 Pattern-Based Recurrence Solving

Loop bounds of simple loops (S) are derived by the `loopBounds` part of r-TuBound. For doing so, `loopBounds` implements a pattern-based recurrence solving algorithm, as follows.

An additional variable, denoted by n, is introduced to speak about the value $i(n)$ of the variable i at the nth iteration of the loop. Using this notation, the update expression of (S) can be modeled by a linear recurrence equation with constant coefficient.

Such recurrences can always be solved [3]. Hence, $i(n)$ is expressed as a function, i.e. the closed form, over n and the initial value of i. However, `loopBounds` does not implement the general algorithm for solving linear recurrences of arbitrary orders, but it makes use of the restrictions imposed over (S). Since i is modified in (S) by a non-constant linear expression over i, the resulting recurrence equation of $i(n)$ is a (homogeneous) linear recurrence of order 2. Using the generic closed form pattern of such recurrences, `loopBounds` derives the closed form of $i(n)$ by instantiating the symbolic constants in the generic closed form with expressions over the initial value of i.

The loop bound of (S) is then inferred by computing the smallest n such that the loop condition holds at the nth loop iteration but it is violated at the $n+1$th iteration (see [6] for more details). That is, the loop bound is obtained as a satisfying assignment over n such that the formula below holds:

$$n \geq 0 \ \wedge \ i(n) < b \ \wedge \ i(n+1) \geq b, \tag{2}$$

where the constant b is as given in (S).

The usage of the loop bound computation part of r-TuBound is listed below.

Command 2.3 : `loopBounds simple.c`

Input: Simple loop as in (S)
Output: Loop annotated with its loop bound

EXAMPLE 2.4 For the `simple1.c` loop given in Figure 5 we obtain:

```
Input:  loopBounds simple1.c
Output: annot1.c
```

where `annot1.c` is listed in Figure 6. The annotation `#wcet_loopbound(6)` specifies that `loopBounds` computed 6 as the loop bound of `simple1.c`.

Similarly, the annotated loop derived by `loopBounds` for Figure 8 is given in Figure 9.

The pattern-based recurrence solving algorithm and the satisfiability checking of (2) are implemented in `loopBounds` on top of Prolog. `loopBounds` operates on the TERM representation offered by the Termite library [9]. Let us note that the closed form representation of $i(n)$ in equation (2) involves, in general, exponential sequences in n. Therefore, to compute the value of n such that (2) holds, `loopBounds` makes use of the logarithm, floor and ceiling built-in functions of Prolog.

3 r-TuBound: Experimental Results

The overall flow of r-TuBound is given in Figure 1. The program analysis framework `loopExtraction`, the loop refinement step `loopRefine` and the WCET computation part `wcetComputation` of r-TuBound are written in C++. The loop bound

computation engine `loopBounds` of r-TuBound is implemented on top of the Termite library of Prolog. The `loopExtraction` and `wcetComputation` components of r-TuBound are based on the work presented in [10]. Our contribution in r-TuBound comes with extending [10] with an automatic inference of symbolic loop bounds. r-TuBound offers thus software support for the timing analysis of programs by **r**ecurrence solving and SMT based flow **r**efinement in conjunction with WCET techniques (TuBound).

The `loopRefine` part of r-TuBound comprises about 1000 lines of C++ code, whereas the `loopBounds` engine of r-TuBound contains about 350 lines of Prolog code. The `loopRefine` and `loopBounds` parts of r-TuBound are glued together using a 50 lines shellscript code.

Experimental Evaluation and Comparison. We evaluated r-TuBound on a number of benchmarks coming from the WCET community, as well as on some industrial examples coming from Dassault Aviation. The results are summarized in Table 1. The first column of Table 1 contains the name of the analysed benchmark suite. The second and third columns give respectively the lines of code and the total number of loops in the benchmark suite. The fourth column presents the number of loops for which r-TuBound inferred loop bounds. For a detailed evaluation of r-TuBound we refer to [5].

The Debie-1d and Mälardalen examples come from the WCET community and were used in the WCET tool challenges [11]. These examples are fine tuned for the WCET analysis of programs. Loop bounds need to be either inferred or assumed to be a priori given as program annotations. r-TuBound inferred loop bounds for 180 loops out of the 227 loops coming from the WCET community; some of the 180 loops could not yet be treated by other WCET tools, such as [10,8]. The remaining 47 loops could not be handled by r-TuBound as various restrictions of equation (M) were violated. Namely, the loops had a nested-loop structure, loop updates contained operations over arrays and pointers, and non-linear and/or floating point arithmetic was used in the loop body,

We also run r-TuBound on 77 loops coming from Dassault Aviation. These examples have not yet been optimised for the WCET analysis of programs. When compared to [10], r-TuBounds infers non-trivial loop bounds for 46 loops. Out of these 46 loops, the approach of [10] can only handle 39 loops. The 7 loops that can only be treated by r-TuBound involved nested loops and multi-path reasoning with non-trivial linear arithmetic updates over the loop counter. r-TuBound failed on 31 loops coming from Dassault Aviation, as these loops required the analysis of nested loops with floating point arithmetic.

The current version of r-TuBound has successfully participated in the WCET 2011 tool challenge [11]. When compared to other WCET tools, such as Sweet [4] and OTAWA+oRange [7], we observed that the annotation language of r-TuBound has very little support for specifying variable input ranges or program execution frequencies. Moreover, r-TuBound was the only WCET tool whose results were obtained on the C16x microcontroller; the other WCET tools target the ARM7 or the Freescale MPC555x microcontrollers. Extending the annotation language and microcontroller support of r-TuBound is left for further work.

Experiments and Runtime. We also analysed the runtime of the flow refinement and recurrence solving parts of r-TuBound. The pattern-based recurrence solving approach

Table 1. Experimental results with r-TuBound

Benchmark Suite	# LoC	# Loops	r-TuBound
Mälardalen	$\sim$ 7500	152	121
Debie-1d	$\sim$ 6100	75	59
Dassault	$\sim$ 1000	77	46
Total	$\sim$ 14700	304	226

of `loopBounds` essentially takes no time: for every loop we tried, a loop bound is inferred in less than 0.5 seconds. The runtime performance of `loopRefine` is also relatively good; the flow refinement (i.e. parsing the code, executing the required SMT queries and writing back the simplified loop) of multi-paths loops with 1000 lines of code takes on average 5 - 20 seconds.

Our experiments thus suggest that r-TuBound is quite fast in practical application. We believe that improving the SMT based reasoning engine of `loopRefine`, for example by applying program slicing before monotonicity analysis, would yield overall better execution times for r-TuBound. We leave this task for further investigation.

4 Conclusion

r-TuBound offers software support for generating loop bounds in the WCET analysis of programs. The distinctive features of r-TuBound come with a pattern-based recurrence solving algorithm and over-approximating loop bounds of multi-path loops. For doing so, multi-path loops are translated into simple loops by using SMT encodings and deriving minimal updates over the loop counter. We presented the workflow of r-TuBound, illustrated how r-TuBound can be used on some example problems, and gave an overview on experimental results.

References

1. Blanc, R., Henzinger, T.A., Hottelier, T., Kovács, L.: ABC: Algebraic Bound Computation for Loops. In: Clarke, E.M., Voronkov, A. (eds.) LPAR-16 2010. LNCS, vol. 6355, pp. 103–118. Springer, Heidelberg (2010)
2. Brummayer, R., Biere, A.: Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 174–177. Springer, Heidelberg (2009)
3. Everest, G., van der Poorten, A., Shparlinski, I., Ward, T.: Recurrence Sequences. Mathematical Surveys and Monographs, vol. 104. American Mathematical Society (2003)
4. 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)
5. Knoop, J., Kovacs, L., Zwirchmayr, J.: An Evaluation of WCET Analysis using Symbolic Loop Bounds. In: Proc. of WCET (2011)
6. Knoop, J., Kovacs, L., Zwirchmayr, J.: Symbolic Loop Bound Computation for WCET Analysis. In: Proc. of PSI, p. 116 (2011)
7. De Michiel, M., Bonenfant, A., Cassé, H., Sainrat, P.: Static Loop Bound Analysis of C Programs Based on Flow Analysis and Abstract Interpretation. In: RTCSA, pp. 161–166 (2008)

8. Schoeberl, M., Puffitsch, W., Pedersen, R.U., Huber, B.: Worst-case Execution Time Analysis for a Java Processor. Software: Practice and Experience 40/6, 507–542 (2010)
9. Prantl, A.: The Termite Library, http://www.complang.tuwien.ac.at/adrian/termite/Manual/
10. Prantl, A., Schordan, M., Knoop, J.: TuBound - A Conceptually New Tool for WCET Analysis. In: Proc. of WCET, pp. 141–148 (2008)
11. von Hanxleden, R., et al.: The WCET Tool Challenge 2011: Report. In: Proc. of WCET (2011) (under journal submission)

Author Index