Cesare Tinelli
Viorica Sofronie-Stokkermans (Eds.)

LNAI 6989

Frontiers of Combining Systems

8th International Symposium, FroCoS 2011
Saarbrücken, Germany, October 2011
Proceedings

Springer

Lecture Notes in Artificial Intelligence 6989

Subseries of Lecture Notes in Computer Science

LNAI Series Editors

LNAI Founding Series Editor

Cesare Tinelli Viorica Sofronie-Stokkermans (Eds.)

Frontiers of Combining Systems

8th International Symposium, FroCoS 2011
Saarbrücken, Germany, October 5-7, 2011
Proceedings

Springer

Volume Editors

Cesare Tinelli
The University of Iowa
Iowa City, IA 52242, USA
E-mail: cesare-tinelli@uiowa.edu

Viorica Sofronie-Stokkermans
Max-Planck-Institut für Informatik
66123 Saarbrücken, Germany
E-mail: sofronie@mpi-inf.mpg.de

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-24363-9 e-ISBN 978-3-642-24364-6
DOI 10.1007/978-3-642-24364-6
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2011936707

CR Subject Classification (1998): I.2, I.2.3, D.3.1, F.4, I.1, F.2

LNCS Sublibrary: SL 7 – Artificial Intelligence

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 collects papers presented at the 8th International Symposium on Frontiers of Combining Systems (FroCoS 2011), held October 5–7, 2011 in Saarbrücken, Germany. Previous FroCoS meetings were organized in Munich (1996), Amsterdam (1998), Nancy (2000), Santa Margherita Ligure (2002), Vienna (2005), Liverpool (2007) and Trento (2009). In 2004, 2006, 2008 and 2010 FroCoS joined IJCAR, the International Joint Conference on Automated Reasoning. Like its predecessors, FroCoS 2011 offered a common forum for the presentation and discussion of research in the general area of combination, modularization and integration of systems, with emphasis on logic-based systems and their applications. This research touches on many areas of computer science such as computational logic, program development and verification, artificial intelligence, automated reasoning, constraint solving, declarative programming, and symbolic computation.

The Program Committee accepted 15 papers out of a total of 22 submissions. Each submission was reviewed by at least three Program Committee members or external reviewers. We thank all the reviewers for their work and all the members of the Program Committee for their careful and thoughtful deliberations.

In addition to the contributed papers, the program included three invited lectures, by Alessandro Artale (Free University of Bozen-Bolzano), Martin Lange (University of Kassel) and Tobias Nipkow (Technical University Munich), and an invited tutorial by André Platzer (Carnegie-Mellon University). We are grateful to the invited speakers not only for their interesting presentations, but also for contributing extended abstracts or full papers to the proceedings.

Many people and institutions contributed to the success of FroCoS 2011. We are indebted to the members of the Program Committee and to the additional referees for the thorough reviewing work, to the members of the FroCoS Steering Committee for their support, and to Andrei Voronkov for his indispensable EasyChair conference management system. We would also like to thank the Max Planck Institute for financial support and for providing the infrastructure. We are very grateful to Uwe Brahm, Manuel Lamotte, Jennifer Müller, Roxane Wetzel and Anja Zimmer for their help with the organization of the conference, and to Christoph Weidenbach for his support. Last, but not least, we thank all authors who submitted papers to FroCoS 2011 and all the symposium participants.

July 2011

Viorica Sofronie-Stokkermans
Cesare Tinelli

Organization

Program Committee

Franz Baader	TU Dresden, Germany
Clark Barrett	New York University, USA
Peter Baumgartner	NICTA, Australia
Torben Braüner	Roskilde University, Denmark
Thom Fruehwirth	University of Ulm, Germany
Silvio Ghilardi	University of Milan, Italy
Jürgen Giesl	RWTH Aachen, Germany
Valentin Goranko	Technical University of Denmark
Bernhard Gramlich	TU Wien, Austria
Sava Krstić	Intel Corporation, USA
Carsten Lutz	University of Bremen, Germany
Till Mossakowski	DFKI GmbH Bremen, Germany
Silvio Ranise	FBK-IRST, Italy
Christophe Ringeissen	LORIA-INRIA, France
Philipp Rümmer	Uppsala University, Sweden
Renate A. Schmidt	The University of Manchester, UK
Roberto Sebastiani	DISI, University of Trento, Italy
Viorica Sofronie-Stokkermans	MPI Informatik and University of Koblenz-Landau, Germany
Volker Sorge	University of Birmingham, UK
Cesare Tinelli	The University of Iowa, USA
Wolfgang Windsteiger	RISC, Johannes Kepler University, Austria
Frank Wolter	University of Liverpool, UK

Additional Reviewers

Blanchette, Jasmin Christian	Frigeri, Achille	Küsters, Ralf
Blanqui, Frederic	Genet, Thomas	Lohrey, Markus
Bruttomesso, Roberto	Goel, Amit	Maratea, Marco
Bursuc, Sergiu	Göller, Stefan	Marx, Maarten
Cortier, Veronique	Hadarean, Liana	Nipkow, Tobias
Deters, Morgan	Kahrs, Stefan	Papacchini, Fabio
Eggersglüß, Stephan	King, Tim	Saubion, Frédéric
	Kutsia, Temur	Tonetta, Stefano

Table of Contents

Invited Papers

Contributed Papers

Tailoring Temporal Description Logics for Reasoning over Temporal Conceptual Models

Alessandro Artale[1], Roman Kontchakov[2], Vladislav Ryzhikov[1], and Michael Zakharyaschev[2]

[1] KRDB Research Centre
Free University of Bozen-Bolzano, Italy
{lastname}@inf.unibz.it
[2] Dept. of Comp. Science and Inf. Sys.
Birkbeck College, London, UK
{roman,michael}@dcs.bbk.ac.uk

Abstract. Temporal data models have been used to describe how data can evolve in the context of temporal databases. Both the Extended Entity-Relationship (EER) model and the Unified Modelling Language (UML) have been temporally extended to design temporal databases. To automatically check quality properties of conceptual schemas various encoding to Description Logics (DLs) have been proposed in the literature. On the other hand, reasoning on temporally extended DLs turn out to be too complex for effective reasoning ranging from 2EXPTIME up to undecidable languages. We propose here to temporalize the 'light-weight' *DL-Lite* logics obtaining nice computational results while still being able to represent various constraints of temporal conceptual models. In particular, we consider temporal extensions of $\mathit{DL\text{-}Lite}^{\mathcal{N}}_{bool}$, which was shown to be adequate for capturing non-temporal conceptual models without relationship inclusion, and its fragment $\mathit{DL\text{-}Lite}^{\mathcal{N}}_{core}$ with most primitive concept inclusions, which are nevertheless enough to represent almost all types of atemporal constraints (apart from covering).

1 Introduction

Conceptual data modelling formalisms such as the Unified Modelling Language (UML) and the Extended Entity-Relationship (EER) model have become a *de facto* standard in database design and software engineering by providing visual means to describe application domains in a declarative and reusable way. Both UML and EER turn out to be closely connected to description logics (DLs), which can encode constraints expressible in these conceptual modelling formalisms (see, e.g., [11, 12, 1]). This encoding provides us with a rigorous definition of various *quality properties* of conceptual schemas. For instance, given a conceptual schema, we can check its *consistency* (i.e., whether its constraints contain no contradictions), entity and relationship *satisfiability* (i.e., whether given entities and relationships in the schema can be instantiated), *instance checking* (i.e., whether a given individual belongs to a given entity in every instance of the schema), and *logical entailment* (i.e., whether a given constraint is logically implied by the schema). The encoding of conceptual models as DL knowledge

C. Tinelli and V. Sofronie-Stokkermans (Eds.): FroCoS 2011, LNAI 6989, pp. 1–11, 2011.

bases (KBs) opens a way for utilizing existing DL reasoning services (reasoners) for automated checking of these quality properties, and so for providing an effective reasoning support for the construction phase of a conceptual model schema.

Temporal conceptual data models [31,20,21,4,26,6,15,7,10] extend standard conceptual schemas with means to visually represent temporal constraints imposed on temporal database instances. Temporal constraints can be grouped in three categories: timestamping, evolution and temporal cardinality constraints. *Timestamping constraints* discriminate between those entities, r relationships and attributes that change over time and those that are time-invariant [31,21,16,7,26]. *Evolution constraints* control how the domain elements evolve over time by 'migrating' from one entity to another [22,25,29,26,6]. We distinguish between *quantitative* evolution constraints that specify the exact time of migration and *qualitative* evolution constraints that describe eventual temporal behaviour (i.e., whether all instances will eventually migrate or will always belong to the same entity). *Temporal cardinality constraints* restrict the number of times an instance participates in a relationship; *snapshot* cardinality constraints do it at each moment of time, while *lifespan* cardinality constraints impose restrictions over the entire existence of the instance [30,24].

Temporal conceptual models can be encoded in various temporal description logics (TDLs), which have been designed and investigated since the seminal paper [28] with the aim of understanding the computational price of introducing a temporal dimension in DLs (see [23] for a survey). A general conclusion one can draw from the obtained results is that—as far as there is a nontrivial interaction between the temporal and DL components—TDLs based on full-fledged DLs like $\mathcal{ALC}$ turn out to be too complex for effective reasoning ranging from 2ExpTime up to undecidable languages.

The aim of this paper is to show how temporalizing the 'light-weight' *DL-Lite* logics [13,14,27,2,3] we can represent various constraints of temporal conceptual models. In particular, we consider *DL-Lite*$^{\mathcal{N}}_{bool}$, which was shown to be adequate for capturing non-temporal conceptual models without relationship inclusion [1], and its fragment *DL-Lite*$^{\mathcal{N}}_{core}$ with most primitive concept inclusions, which are nevertheless enough to represent almost all types of atemporal constraints (apart from covering). To capture temporal constraints, we interpret the TDLs over the flow of time $(\mathbb{Z}, <)$, in which (1) the future and past temporal operators can be applied to concepts (entities); (2) roles can be declared flexible or rigid; (3) the 'undirected' temporal operators 'always' and 'some time' can be applied to roles; (4) the concept inclusions (TBox) hold at all moments of time (i.e., global) and the database assertions (ABox) are specified to hold at particular moments of time.

Complexity results for reasoning in TDLs based on *DL-Lite* have been presented in [5,8,9]. The most expressive TDL based on *DL-Lite*$^{\mathcal{N}}_{bool}$ and featuring all of (1)–(4) turns out to be undecidable. This 'negative' result has motivated our study of various fragments of the full language by restricting not only the DL but also the temporal component. Concerning TDLs with temporalized roles, in addition to the undecidability result, we have also shown that using the undirected temporal operators always/sometime together with temporalized roles over *DL-Lite*$^{\mathcal{N}}_{bool}$ results in an NP-complete language. TDLs with rigid (and flexible but not temporalized) roles turned out to be reducible to propositional linear temporal logic $\mathcal{LTL}$ (and its natural fragments). The absence

of temporalized roles makes reasoning in these logics easier with complexity results ranging from NLOGSPACE to PSPACE.

2 *DL-Lite* Logics

We briefly introduce *DL-Lite* and its relatives (see [14,3] for more details). The language of *DL-Lite*$^{\mathcal{N}}_{bool}$ contains *object names* $a_0, a_1, \dots$, *concept names* $A_0, A_1, \dots$, and *role names* $P_0, P_1, \dots$. *Roles* R, *basic concepts* B and *concepts* C of this language are defined by the rules:

$$\begin{array}{rcl} R & ::= & P_k \mid P_k^-, \\ B & ::= & \bot \mid A_k \mid \geq q\, R, \\ C & ::= & B \mid \neg C \mid C_1 \sqcap C_2, \end{array}$$

where q is a positive integer. A *DL-Lite*$^{\mathcal{N}}_{bool}$ *TBox*, $\mathcal{T}$, is a finite set of *concept inclusion axioms* of the form

$$C_1 \sqsubseteq C_2.$$

An *ABox*, $\mathcal{A}$, is a finite set of assertions of the form

$$A_k(a_i), \qquad \neg A_k(a_i), \qquad P_k(a_i, a_j), \qquad \neg P_k(a_i, a_j).$$

Taken together, $\mathcal{T}$ and $\mathcal{A}$ constitute the *DL-Lite*$^{\mathcal{N}}_{bool}$ *knowledge base* (KB, for short) $\mathcal{K} = (\mathcal{T}, \mathcal{A})$.

An *interpretation* $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ of this and other *DL-Lite* languages consists of a *domain* $\Delta^{\mathcal{I}} \neq \emptyset$ and an interpretation function $\cdot^{\mathcal{I}}$ that assigns to each object name a_i an element $a_i^{\mathcal{I}} \in \Delta^{\mathcal{I}}$, to each concept name A_k a subset $A_k^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$, and to each role name P_k a binary relation $P_k^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$. As in databases, we adopt the *unique name assumption* (UNA) according to which $a_i^{\mathcal{I}} \neq a_j^{\mathcal{I}}$ for all $i \neq j$. The role and concept constructs are interpreted in $\mathcal{I}$ as follows:

$$\begin{array}{rcl} (P_k^-)^{\mathcal{I}} & = & \{(y, x) \in \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} \mid (x, y) \in P_k^{\mathcal{I}}\}, \\ \bot^{\mathcal{I}} & = & \emptyset, \\ (\geq q\, R)^{\mathcal{I}} & = & \big\{x \in \Delta^{\mathcal{I}} \mid \sharp\{y \in \Delta^{\mathcal{I}} \mid (x, y) \in R^{\mathcal{I}}\} \geq q\big\}, \\ (\neg C)^{\mathcal{I}} & = & \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}, \\ (C_1 \sqcap C_2)^{\mathcal{I}} & = & C_1^{\mathcal{I}} \cap C_2^{\mathcal{I}}, \end{array}$$

where $\sharp X$ denotes the cardinality of X. The *satisfaction relation* $\models$ is defined as usual:

$$\begin{array}{rcl} \mathcal{I} \models C_1 \sqsubseteq C_2 & \text{iff} & C_1^{\mathcal{I}} \subseteq C_2^{\mathcal{I}}, \\ \mathcal{I} \models A_k(a_i) & \text{iff} & a_i^{\mathcal{I}} \in A_k^{\mathcal{I}}, \qquad \mathcal{I} \models \neg A_k(a_i) \quad \text{iff} \quad a_i^{\mathcal{I}} \notin A_k^{\mathcal{I}}, \\ \mathcal{I} \models P_k(a_i, a_j) & \text{iff} & (a_i^{\mathcal{I}}, a_j^{\mathcal{I}}) \in P_k^{\mathcal{I}}, \quad \mathcal{I} \models \neg P_k(a_i, a_j) \quad \text{iff} \quad (a_i^{\mathcal{I}}, a_j^{\mathcal{I}}) \notin P_k^{\mathcal{I}}. \end{array}$$

A knowledge base $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ is said to be *satisfiable* (or *consistent*) if there is an interpretation, $\mathcal{I}$, satisfying all the members of $\mathcal{T}$ and $\mathcal{A}$. In this case we write $\mathcal{I} \models \mathcal{K}$ (as well as $\mathcal{I} \models \mathcal{T}$ and $\mathcal{I} \models \mathcal{A}$) and say that $\mathcal{I}$ is a *model of* $\mathcal{K}$ (and of $\mathcal{T}$ and $\mathcal{A}$).

The two sub-languages of *DL-Lite*$^{\mathcal{N}}_{bool}$ we deal with in this article are obtained by restricting the Boolean operators on concepts. In *DL-Lite*$^{\mathcal{N}}_{krom}$ TBoxes,[1] concept inclusions are of the form

$$B_1 \sqsubseteq B_2, \qquad B_1 \sqsubseteq \neg B_2 \qquad \text{or} \qquad \neg B_1 \sqsubseteq B_2. \tag{krom}$$

(Here and below the B_i are basic concepts.) In *DL-Lite*$^{\mathcal{N}}_{core}$, we can only use concept inclusions of the form

$$B_1 \sqsubseteq B_2 \qquad \text{or} \qquad B_1 \sqcap B_2 \sqsubseteq \bot. \tag{core}$$

As $B_1 \sqsubseteq \neg B_2$ is equivalent to $B_1 \sqcap B_2 \sqsubseteq \bot$, *DL-Lite*$^{\mathcal{N}}_{core}$ is a sub-language of *DL-Lite*$^{\mathcal{N}}_{krom}$.

The extra expressive power, gained from covering constraints, comes at a price: the satisfiability problem is NLOGSPACE-complete for *DL-Lite*$^{\mathcal{N}}_{core}$ and *DL-Lite*$^{\mathcal{N}}_{krom}$ KBs and NP-complete for *DL-Lite*$^{\mathcal{N}}_{bool}$ KBs [2].

3 Temporal Conceptual Modelling

Temporal conceptual data models extend standard conceptual schemas with means to visually represent temporal constraints imposed on temporal database instances [31, 20, 21, 4, 26]. When introducing a temporal dimension into conceptual data models, time is usually modelled by a linearly ordered set of time instants, so that at each moment we can refer to its past and its future. We assume that the flow of time is isomorphic to the strictly linearly ordered set $(\mathbb{Z}, <)$ of integer numbers. (For a survey of other options, including various interval-based and branching models of time, consult, e.g. [18, 19, 17].)

A basic assumption made in temporal conceptual models is that entities, relationships and attributes may freely change over time—as long as they satisfy the schema constraints at *each* time instant. Temporal constructs are then used to impose constraints on the temporal behaviour of various components of conceptual schemas. We group these constructs into three categories—*timestamping*, *evolution constraints* and *temporal cardinality constraints*—and illustrate them using the temporal data model in Figure 1.

Timestamping constraints [31, 21, 26] distinguish between entities, relationships and attributes that are *temporary*, i.e., cannot keep a single element over the whole timeline; *snapshot*, or time-invariant; and *unconstrained* (all others). In temporal entity-relationship (TER) diagrams, temporary entities, relationships and attributes are marked with T and snapshot ones with S. In Figure 1, 'Employee' and 'Department' are snapshot entities, 'Name,' 'PaySlipNumber' and 'ProjectCode' are snapshot attributes and 'Member' a snapshot relationship. On the other hand, 'Manager' is a temporary entity, 'Salary' a temporary attribute and 'WorksOn' a temporary relationship.

To represent timestamping constraints in temporal description logics we employ the *temporal operator* $\boxtimes$, which is read as 'always' or 'at all—past, present and future—time instants.' Intuitively, for a concept C, $\boxtimes C$ contains those elements that belong to C

[1] The Krom fragment of first-order logic consists of all formulas in prenex normal form whose quantifier-free part is a conjunction of binary clauses.

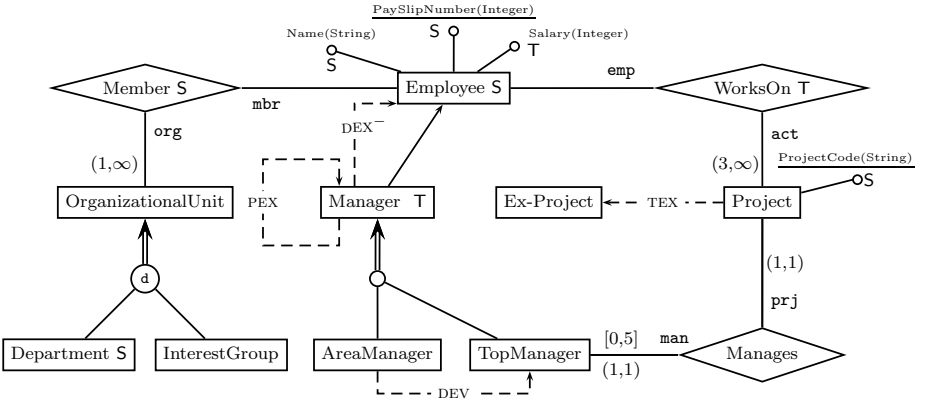


Fig. 1. A temporal conceptual model of a company information system

at all time instants. Using this operator, the constraints 'Employee is a snapshot entity' and 'Manager is a temporary entity' can be represented as follows:

$$\mathit{Employee} \sqsubseteq \boxdot \mathit{Employee}, \tag{1}$$

$$\boxdot \mathit{Manager} \sqsubseteq \bot. \tag{2}$$

The first inclusion says that, at any moment of time, every element of 'Employee' has always been and will always be an element of 'Employee.' The second one states that no element can belong to 'Manager' at all time instants. Note that we consider concept inclusions to hold *globally*, that is, at all moments of time.

The same temporal operator $\boxdot$ together with rigid roles (i.e., roles that do not change over time) can be used to capture timestamping of (reified) relationships. Rigid roles can also represent snapshot attributes, while temporary attributes can be captured by using temporalized roles: $\exists \boxdot \mathit{salary} \sqsubseteq \bot$, where $\boxdot \mathit{salary}$ denotes the intersection of the relations *salary* at all time instants, model *salary* as a temporary attribute.

Evolution constraints control how the domain elements evolve over time by 'migrating' from one entity to another [22, 25, 29, 26, 6]. We distinguish between *qualitative* evolution constraints that describe eventual temporal behaviour and do not specify the moment of migration, and *quantitative* evolution (or transition) constraints that specify the exact moment of migration. The dashed arrow marked with TEX in Figure 1 is an example of a quantitative evolution constraint meaning that each 'Project' expires in exactly one year and becomes an 'Ex-Project.' The dashed arrow marked with DEV is a qualitative evolution constraint meaning that every 'AreaManager' will eventually (at some moment in the future) become a 'TopManager.' The DEX$^-$ dashed arrow says that every 'Manager' was once an 'Employee,' while the PEX dashed arrow means that a 'Manager' will always be a 'Manager' and cannot be demoted. In temporal description logic, these evolution constraints are represented using temporal operators such as 'at the next moment of time' $\bigcirc_F$, 'some time in the future' $\Diamond_F$, 'some time in the past' $\Diamond_P$ and 'always in the future' $\Box_F$:

$$\mathit{Project} \sqsubseteq \bigcirc_F \mathit{Ex\text{-}Project}, \tag{3}$$
$$\mathit{AreaManager} \sqsubseteq \Diamond_F \mathit{TopManager}, \tag{4}$$
$$\mathit{Manager} \sqsubseteq \Diamond_P \mathit{Employee}, \tag{5}$$
$$\mathit{Manager} \sqsubseteq \Box_F \mathit{Manager}. \tag{6}$$

We note again that these concept inclusions hold at every moment of time.

Temporal cardinality constraints [30,24,20] restrict the number of times an instance participates in a relationship. *Snapshot* cardinality constraints do it at each moment of time, while *lifespan* cardinality constraints impose restrictions over the entire existence of the instance. In Figure 1, we use (k, l) to specify the snapshot cardinalities and $[k, l]$ the lifespan cardinalities: for example, every 'TopManager' manages exactly one project at each moment of time (snapshot cardinality), but not more than five different projects over the whole career (lifespan cardinality). If the relationship 'manages' is represented by a role in temporal description logic then these two constraints can be expressed by the following concept inclusions:

$$\mathit{TopManager} \sqsubseteq \leq 1\, \mathit{manages},$$
$$\mathit{TopManager} \sqsubseteq \leq 5\, \diamondsuit\!\!\!\!-\, \mathit{manages},$$

where $\diamondsuit\!\!\!\!-$ means 'sometime' (in the past, present or future), and so $\diamondsuit\!\!\!\!-\mathit{manages}$ is the union of the relations *manages* over *all* time instants.

Finally, to represent temporal database instances associated to a temporal conceptual model, we use assertions like $\bigcirc_P \mathit{Manager}(\mathit{bob})$ for 'Bob was a manager last year' and $\bigcirc_F \mathit{manages}(\mathit{bob}, \mathit{cronos})$ for 'Bob will manage project Cronos next year.'

3.1 Temporal *DL-Lite* Logics

It is known from temporal logic [18] that all the temporal operators used in the previous section can be expressed in terms of the binary operators 'since' $\mathcal{S}$ and 'until' $\mathcal{U}$. So we formulate our 'base' temporal extension $\mathit{T}_{\mathcal{US}}\mathit{DL\text{-}Lite}^{\mathcal{N}}_{\mathit{bool}}$ of the description logic $\mathit{DL\text{-}Lite}^{\mathcal{N}}_{\mathit{bool}}$ using only these two operators. The language of $\mathit{T}_{\mathcal{US}}\mathit{DL\text{-}Lite}^{\mathcal{N}}_{\mathit{bool}}$ contains *object names* $a_0, a_1, \ldots$, *concept names* $A_0, A_1, \ldots$, *flexible role names* $P_0, P_1, \ldots$ and *rigid role names* $G_0, G_1, \ldots$. *Role names* S, *roles* R, *basic concepts* B, *concepts* C and *temporal concepts* D are defined by the following rules:

$$\begin{aligned}
S &::= P_i \mid G_i,\\
R &::= S \mid S^-,\\
B &::= \bot \mid A_i \mid \geq q\, R,\\
C &::= B \mid D \mid \neg C \mid C_1 \sqcap C_2,\\
D &::= C \mid C_1\, \mathcal{U}\, C_2 \mid C_1\, \mathcal{S}\, C_2,
\end{aligned}$$

where, as before, q is a positive integer. A $\mathit{T}_{\mathcal{US}}\mathit{DL\text{-}Lite}^{\mathcal{N}}_{\mathit{bool}}$ *TBox*, $\mathcal{T}$, is a finite set of *concept inclusions* of the form $C_1 \sqsubseteq C_2$. An *ABox*, $\mathcal{A}$, consists of assertions of the form

$$\bigcirc^n A_k(a_i), \qquad \bigcirc^n \neg A_k(a_i), \qquad \bigcirc^n S(a_i, a_j) \quad \text{and} \quad \bigcirc^n \neg S(a_i, a_j),$$

where A_k is a concept name, S a (flexible or rigid) role name, a_i, a_j object names and, for $n \in \mathbb{Z}$,

$$\bigcirc^n = \underbrace{\bigcirc_F \cdots \bigcirc_F}_{n \text{ times}}, \text{ if } n \geq 0 \qquad \text{and} \qquad \bigcirc^n = \underbrace{\bigcirc_P \cdots \bigcirc_P}_{-n \text{ times}}, \text{ if } n < 0.$$

Taken together, the TBox $\mathcal{T}$ and ABox $\mathcal{A}$ form the *knowledge base* (KB) $\mathcal{K} = (\mathcal{T}, \mathcal{A})$.

A *temporal interpretation*, $\mathcal{I}$, gives a standard DL interpretation, $\mathcal{I}(n)$, for each time instant $n \in \mathbb{Z}$:

$$\mathcal{I}(n) \;=\; \big(\Delta^{\mathcal{I}}, a_0^{\mathcal{I}}, \dots, A_0^{\mathcal{I}(n)}, \dots, P_0^{\mathcal{I}(n)}, \dots, G_0^{\mathcal{I}}, \dots\big).$$

We assume, however, that the domain $\Delta^{\mathcal{I}}$ and the interpretations $a_i^{\mathcal{I}} \in \Delta^{\mathcal{I}}$ of the object names and $G_0^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$ of rigid role names are fixed for all time. (Recall also that we adopt the UNA.) The interpretations $A_i^{\mathcal{I}(n)} \subseteq \Delta^{\mathcal{I}}$ of concept names and $P_i^{\mathcal{I}(n)} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$ of flexible role names can vary. The atemporal constructs are interpreted in $\mathcal{I}(n)$ as before; we write $C^{\mathcal{I}(n)}$ for the extension of concept C in the interpretation $\mathcal{I}(n)$. The interpretation of the temporal operators is as in temporal logic:

$$\begin{aligned}
(C_1 \,\mathcal{U}\, C_2)^{\mathcal{I}(n)} \;&=\; \bigcup_{k>n} \big(C_2^{\mathcal{I}(k)} \cap \bigcap_{n<m<k} C_1^{\mathcal{I}(m)}\big),\\
(C_1 \,\mathcal{S}\, C_2)^{\mathcal{I}(n)} \;&=\; \bigcup_{k<n} \big(C_2^{\mathcal{I}(k)} \cap \bigcap_{n>m>k} C_1^{\mathcal{I}(m)}\big).
\end{aligned}$$

Concept inclusions are interpreted in $\mathcal{I}$ *globally*:

$$\mathcal{I} \models C_1 \sqsubseteq C_2 \quad \text{iff} \quad C_1^{\mathcal{I}(n)} \subseteq C_2^{\mathcal{I}(n)} \quad \text{for } \textit{all } n \in \mathbb{Z}.$$

And for the ABox assertions, we set:

$$\begin{aligned}
&\mathcal{I} \models \bigcirc^n A_k(a_i) \text{ iff } a_i^{\mathcal{I}} \in A_k^{\mathcal{I}(n)}, && \mathcal{I} \models \bigcirc^n \neg A_k(a_i) \text{ iff } a_i^{\mathcal{I}} \notin A_k^{\mathcal{I}(n)},\\
&\mathcal{I} \models \bigcirc^n S(a_i, a_j) \text{ iff } (a_i^{\mathcal{I}}, a_j^{\mathcal{I}}) \in S^{\mathcal{I}(n)}, && \mathcal{I} \models \bigcirc^n \neg S(a_i, a_j) \text{ iff } (a_i^{\mathcal{I}}, a_j^{\mathcal{I}}) \notin S^{\mathcal{I}(n)}.
\end{aligned}$$

We call $\mathcal{I}$ a *model* of a KB $\mathcal{K}$ and write $\mathcal{I} \models \mathcal{K}$ if $\mathcal{I}$ satisfies all elements of $\mathcal{K}$. If $\mathcal{K}$ has a model then it is said to be *satisfiable*. A concept C (role R) is *satisfiable* w.r.t. $\mathcal{K}$ if there are a model $\mathcal{I}$ of $\mathcal{K}$ and $n \in \mathbb{Z}$ such that $C^{\mathcal{I}(n)} \neq \emptyset$ (respectively, $R^{\mathcal{I}(n)} \neq \emptyset$). It is readily seen that the concept and role satisfiability problems are equivalent to KB satisfiability.

We now define a few fragments and extensions of the base language $\mathit{T}_{\mathcal{US}}\mathit{DL\text{-}Lite}^{\mathcal{N}}_{\mathit{bool}}$. Recall that to say that C is a snapshot concept, we need the 'always' operator $\circledast$ with the following meaning:

$$(\circledast C)^{\mathcal{I}(n)} \;=\; \bigcap_{k \in \mathbb{Z}} C^{\mathcal{I}(k)}.$$

In terms of $\mathcal{S}$ and $\mathcal{U}$, this operator can be represented as $\circledast C = \neg(\top \,\mathcal{S}\, \neg C) \sqcap C \sqcap \neg(\top \,\mathcal{U}\, \neg C)$. Define $\mathit{T}_{\mathcal{U}}\mathit{DL\text{-}Lite}^{\mathcal{N}}_{\mathit{bool}}$ to be the sublanguage of $\mathit{T}_{\mathcal{US}}\mathit{DL\text{-}Lite}^{\mathcal{N}}_{\mathit{bool}}$ the temporal concepts D in which are of the form:

$$D \;::=\; C \;\mid\; \circledast C. \tag{U}$$

Thus, in T_U*DL-Lite*$^{\mathcal{N}}_{bool}$, we can express timestamping constraints (see Section 3).

The temporal operators $\Diamond_F$ ('some time in the future') and $\Diamond_P$ ('some time in the past') that are required for qualitative evolution constraints with the standard temporal logic semantics

$$(\Diamond_F C)^{\mathcal{I}(n)} = \bigcup_{k>n} C^{\mathcal{I}(k)} \quad \text{and} \quad (\Diamond_P C)^{\mathcal{I}(n)} = \bigcup_{k<n} C^{\mathcal{I}(k)}$$

can be expressed via $\mathcal{U}$ and $\mathcal{S}$ as $\Diamond_F C = \top \,\mathcal{U}\, C$ and $\Diamond_P C = \top \,\mathcal{S}\, C$; the operators $\Box_F$ ('always in the future') and $\Box_P$ ('always in the past') are defined as dual to $\Diamond_F$ and $\Diamond_P$: $\Box_F C = \neg\Diamond_F\neg C$ and $\Box_P C = \neg\Diamond_P\neg C$. We define the fragment T_{FP}*DL-Lite*$^{\mathcal{N}}_{bool}$ of $T_{\mathcal{US}}$*DL-Lite*$^{\mathcal{N}}_{bool}$ by restricting the temporal concepts D to the form:

$$D \;::=\; C \;\mid\; \Box_F C \;\mid\; \Box_P C. \tag{FP}$$

Clearly, we have the following equivalences:

$$\boxplus C = \Box_F\Box_P C \qquad \text{and} \qquad \boxdot C = \Diamond_F\Diamond_P C.$$

In what follows they will be regarded as definitions for $\boxplus$ and $\boxdot$ in the languages, where they are not explicitly present. Thus, T_{FP}*DL-Lite*$^{\mathcal{N}}_{bool}$ is capable of expressing both timestamping and qualitative (but not quantitative) evolution constraints.

The temporal operators $\bigcirc_F$ ('next time') and $\bigcirc_P$ ('previous time'), used in quantitative evolution constraints, can be defined as $\bigcirc_F C = \bot \,\mathcal{U}\, C$ and $\bigcirc_P C = \bot \,\mathcal{S}\, C$, so that we have:

$$(\bigcirc_F C)^{\mathcal{I}(n)} = C^{\mathcal{I}(n+1)} \quad \text{and} \quad (\bigcirc_P C)^{\mathcal{I}(n)} = C^{\mathcal{I}(n-1)}.$$

The fragment of $T_{\mathcal{US}}$*DL-Lite*$^{\mathcal{N}}_{bool}$ with temporal concepts of the form

$$D \;::=\; C \;\mid\; \Box_F C \;\mid\; \Box_P C \;\mid\; \bigcirc_F C \;\mid\; \bigcirc_P C \tag{FPX}$$

will be denoted by T_{FPX}*DL-Lite*$^{\mathcal{N}}_{bool}$. In this fragment, we can express timestamping, qualitative and quantitative evolution constraints.

We have the following inclusions between the languages:

$$T_U\text{DL-Lite}^{\mathcal{N}}_{bool} \;\subseteq\; T_{FP}\text{DL-Lite}^{\mathcal{N}}_{bool} \;\subseteq\; T_{FPX}\text{DL-Lite}^{\mathcal{N}}_{bool} \;\subseteq\; T_{\mathcal{US}}\text{DL-Lite}^{\mathcal{N}}_{bool}.$$

Similarly to the non-temporal case, we can also identify sub-Boolean fragments of the above languages. A temporal TBox $\mathcal{T}$ will be called a *Krom* (*core*) TBox if it contains only concept inclusions of the form:

$$D_1 \sqsubseteq D_2, \qquad D_1 \sqsubseteq \neg D_2, \qquad \neg D_1 \sqsubseteq D_2, \tag{Krom}$$

$$D_1 \sqsubseteq D_2, \qquad D_1 \sqcap D_2 \sqsubseteq \bot, \tag{core}$$

respectively, where the D_i are temporal concepts defined by (FPX), (FP) or (U) with $C ::= B \mid D$ (so, no Boolean operators are allowed in the D_i). This gives us 6 different fragments T_{FPX}*DL-Lite*$^{\mathcal{N}}_{\alpha}$, T_{FP}*DL-Lite*$^{\mathcal{N}}_{\alpha}$ and T_U*DL-Lite*$^{\mathcal{N}}_{\alpha}$, for $\alpha \in \{core, krom\}$. We do not consider the core and Krom fragments of the full language with $\mathcal{U}/\mathcal{S}$ because these operators allow one to go beyond the language of binary clauses of the

core and Krom fragments; the resulting languages would have the same complexity as $T_{\mathcal{US}}$*DL-Lite*$^{\mathcal{N}}_{bool}$ and yet be less expressive (see [9] for more details).

We note here that both Krom and Bool TBoxes have the full negation, and so one can freely use $\Diamond$-shaped counterparts of the $\Box$ temporal operators allowed in the language. This is not the case for the core fragments where timestamping can still be expressed (cf. (1) and (2)) but evolution constraints involving $\Diamond$ (e.g., a *Manager* was once an *Employee*; cf. (5)) are not expressible.

Table 1. The temporal extended *DL-Lite* family and complexity of its members

concept inclusions	temporal constructs		
	$\mathcal{U}/\mathcal{S}, \bigcirc_F/\bigcirc_P, \Box_F/\Box_P$ [a]	$\Box_F/\Box_P$	$\circledast$
Bool	$T_{\mathcal{US}}$*DL-Lite*$^{\mathcal{N}}_{bool}$ T_{FPX}*DL-Lite*$^{\mathcal{N}}_{bool}$ PSPACE	T_{FP}*DL-Lite*$^{\mathcal{N}}_{bool}$ NP	T_U*DL-Lite*$^{\mathcal{N}}_{bool}$ NP
Krom	T_{FPX}*DL-Lite*$^{\mathcal{N}}_{krom}$ NP	T_{FP}*DL-Lite*$^{\mathcal{N}}_{krom}$ NP	T_U*DL-Lite*$^{\mathcal{N}}_{krom}$ NLOGSPACE
core	T_{FPX}*DL-Lite*$^{\mathcal{N}}_{core}$ in PTIME	T_{FP}*DL-Lite*$^{\mathcal{N}}_{core}$ in PTIME	T_U*DL-Lite*$^{\mathcal{N}}_{core}$ NLOGSPACE
temporalized roles	$T^{\mathcal{R}}_X$*DL-Lite*$^{\mathcal{N}}_{bool}$ undec.	?	$T^{\mathcal{R}}_U$*DL-Lite*$^{\mathcal{N}}_{bool}$ NP

[a] Sub-boolean fragments of the language with $\mathcal{U}/\mathcal{S}$ are not defined.

As we have seen in our running example, in order to express lifespan cardinality constraints, temporal operators on roles are required: for a role R of the form

$$R \;::=\; S \;\mid\; S^- \;\mid\; \diamondsuit R \;\mid\; \boxplus R,$$

the extensions of $\diamondsuit R$ and $\boxplus R$ in an interpretation $\mathcal{I}$ are defined as

$$(\diamondsuit R)^{\mathcal{I}(n)} = \bigcup_{k\in\mathbb{Z}} R^{\mathcal{I}(k)} \quad \text{and} \quad (\boxplus R)^{\mathcal{I}(n)} = \bigcap_{k\in\mathbb{Z}} R^{\mathcal{I}(k)}.$$

We denote by $T^{\mathcal{R}}_{\beta}$*DL-Lite*$^{\mathcal{N}}_{bool}$, for $\beta \in \{FPX, FP, U\}$, the extensions of the respective Bool fragments with temporalized roles.

To summarize, the temporal extensions of the *DL-Lite* logics we consider in this paper are collected in Table 1. The tight (unless specified otherwise) complexity bounds of Table 1 have been established in [9].

4 Conclusions

From the complexity-theoretic point of view, the best candidates for reasoning about TCMs appear to be the TDLs T_{FPX}*DL-Lite*$^{\mathcal{N}}_{core}$ and T_{FPX}*DL-Lite*$^{\mathcal{N}}_{bool}$, the former of

which is NP-complete and the latter PSPACE-compete. Moreover, as showed in [9], the reduction of T_{FPX} *DL-Lite*$^{\mathcal{N}}_{core}$ to $\mathcal{LTL}$ can be done deterministically, thus standard $\mathcal{LTL}$ provers can be used for TCM reasoning. We also believe that T_{FPX} *DL-Lite*$^{\mathcal{N}}_{core}$ extended with temporalized roles can be decidable, which remains one of the most challenging open problems. But it seems to be next to impossible to reason in an effective way about all TCM constrains without any restrictions.

References

1. Artale, A., Calvanese, D., Kontchakov, R., Ryzhikov, V., Zakharyaschev, M.: Reasoning over extended ER models. In: Parent, C., Schewe, K.-D., Storey, V.C., Thalheim, B. (eds.) ER 2007. LNCS, vol. 4801, pp. 277–292. Springer, Heidelberg (2007)
2. Artale, A., Calvanese, D., Kontchakov, R., Zakharyaschev, M.: DL-Lite in the light of first-order logic. In: Proc. of the 22nd Nat. Conf. on Artificial Intelligence (AAAI 2007), pp. 361–366 (2007)
3. Artale, A., Calvanese, D., Kontchakov, R., Zakharyaschev, M.: The DL-Lite family and relations. J. of Artificial Intelligence Research 36, 1–69 (2009)
4. Artale, A., Franconi, E., Mandreoli, F.: Description logics for modelling dynamic information. In: Chomicki, J., van der Meyden, R., Saake, G. (eds.) Logics for Emerging Applications of Databases. LNCS. Springer, Heidelberg (2003)
5. Artale, A., Kontchakov, R., Lutz, C., Wolter, F., Zakharyaschev, M.: Temporalising tractable description logics. In: 14th Int. Symposium on Temporal Representation and Reasoning (TIME 2007). IEEE Computer Society, Los Alamitos (2007)
6. Artale, A., Parent, C., Spaccapietra, S.: Evolving objects in temporal information systems. Annals of Mathematics and Artificial Intelligence 50(1-2), 5–38 (2007)
7. Artale, A., Franconi, E.: Foundations of temporal conceptual data models. In: Borgida, A.T., Chaudhri, V.K., Giorgini, P., Yu, E.S. (eds.) Conceptual Modeling: Foundations and Applications. LNCS, vol. 5600, pp. 10–35. Springer, Heidelberg (2009)
8. Artale, A., Kontchakov, R., Ryzhikov, V., Zakharyaschev, M.: DL-Lite with temporalised concepts, rigid axioms and roles. In: Ghilardi, S., Sebastiani, R. (eds.) FroCoS 2009. LNCS, vol. 5749, pp. 133–148. Springer, Heidelberg (2009)
9. Artale, A., Kontchakov, R., Ryzhikov, V., Zakharyaschev, M.: TDL-Lite: How to cook decidable temporal description logics (to be submitted, 2011)
10. Artale, A., Kontchakov, R., Ryzhikov, V., Zakharyaschev, M.: Complexity of reasoning over temporal data models. In: Parsons, J., Saeki, M., Shoval, P., Woo, C., Wand, Y. (eds.) ER 2010. LNCS, vol. 6412, pp. 174–187. Springer, Heidelberg (2010)
11. Baader, F., Calvanese, D., McGuinness, D., Nardi, D., Patel-Schneider, P.F. (eds.): The Description Logic Handbook: Theory, Implementation and Applications, 2nd edn. Cambridge University Press, Cambridge (2003)
12. Berardi, D., Calvanese, D., De Giacomo, G.: Reasoning on UML class diagrams. Artificial Intelligence 168(1-2), 70–118 (2005)
13. Calvanese, D., De Giacomo, G., Lembo, D., Lenzerini, M., Rosati, R.: DL-Lite: Tractable description logics for ontologies. In: Proc. of the 20th Nat. Conf. on Artificial Intelligence (AAAI 2005), pp. 602–607 (2005)
14. Calvanese, D., De Giacomo, G., Lembo, D., Lenzerini, M., Rosati, R.: Tractable reasoning and efficient query answering in description logics: The DL-Lite family. J. of Artificial Intelligence Research (JAIR) 39(3), 385–429 (2007)
15. Combi, C., Degani, S., Jensen, C.S.: Capturing temporal constraints in temporal ER models. In: Li, Q., Spaccapietra, S., Yu, E., Olivé, A. (eds.) ER 2008. LNCS, vol. 5231, pp. 397–411. Springer, Heidelberg (2008)

16. Finger, M., McBrien, P.: Temporal conceptual-level databases. In: Gabbay, D., Reynolds, M., Finger, M. (eds.) Temporal Logics – Mathematical Foundations and Computational Aspects, pp. 409–435. Oxford University Press, Oxford (2000)
17. Gabbay, D., Kurucz, A., Wolter, F., Zakharyaschev, M.: Many-dimensional modal logics: theory and applications. In: Studies in Logic. Elsevier, Amsterdam (2003)
18. Gabbay, D., Hodkinson, I., Reynolds, M.: Temporal Logic: Mathematical Foundations and Computational Aspects, vol. 1. Oxford University Press, Oxford (1994)
19. Gabbay, D., Finger, M., Reynolds, M.: Temporal Logic: Mathematical Foundations and Computational Aspects, vol. 2. Oxford University Press, Oxford (2000)
20. Gregersen, H., Jensen, J.S.: Conceptual modeling of time-varying information. Technical Report TimeCenter TR-35. Aalborg University, Denmark (1998)
21. Gregersen, H., Jensen, J.S.: Temporal Entity-Relationship models – a survey. IEEE Transactions on Knowledge and Data Engineering 11(3), 464–497 (1999)
22. Hall, G., Gupta, R.: Modeling transition. In: Proc. of ICDE 1991, pp. 540–549 (1991)
23. Lutz, C., Wolter, F., Zakharyaschev, M.: Temporal description logics: A survey. In: Proc. of 15th Int. Symposium on Temporal Representation and Reasoning (TIME 2008). IEEE Computer Society, Los Alamitos (2008)
24. McBrien, P., Seltveit, A.H., Wangler, B.: An Entity-Relationship model extended to describe historical information. In: Proc. of CISMOD 1992, Bangalore, India, pp. 244–260 (1992)
25. Mendelzon, A.O., Milo, T., Waller, E.: Object migration. In: Proc. of the 13th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS 1994), pp. 232–242. ACM Press, New York (1994)
26. Parent, C., Spaccapietra, S., Zimanyi, E.: Conceptual Modeling for Traditional and Spatio-Temporal Applications—The MADS Approach. Springer, Heidelberg (2006)
27. Poggi, A., Lembo, D., Calvanese, D., De Giacomo, G., Lenzerini, M., Rosati, R.: Linking data to ontologies. In: Spaccapietra, S. (ed.) Journal on Data Semantics X. LNCS, vol. 4900, pp. 133–173. Springer, Heidelberg (2008)
28. Schild, K.: Combining terminological logics with tense logic. In: Proc. of the 6th Portuguese Conf. on Artificial Intelligence, pp. 105–120. Springer, London (1993)
29. Su, J.: Dynamic constraints and object migration. Theoretical Computer Science 184(1-2), 195–236 (1997)
30. Tauzovich, B.: Towards temporal extensions to the entity-relationship model. In: Proc. of the Int. Conf. on Conceptual Modeling (ER 1991). Springer, Heidelberg (1991)
31. Theodoulidis, C., Loucopoulos, P., Wangler, B.: A conceptual modelling formalism for temporal database applications. Information Systems 16(3), 401–416 (1991)

Automatic Proof and Disproof in Isabelle/HOL

Jasmin Christian Blanchette, Lukas Bulwahn, and Tobias Nipkow

Fakultät für Informatik, Technische Universität München

Abstract. Isabelle/HOL is a popular interactive theorem prover based on higher-order logic. It owes its success to its ease of use and powerful automation. Much of the automation is performed by external tools: The metaprover Sledgehammer relies on resolution provers and SMT solvers for its proof search, the counter-example generator Quickcheck uses the ML compiler as a fast evaluator for ground formulas, and its rival Nitpick is based on the model finder Kodkod, which performs a reduction to SAT. Together with the Isar structured proof format and a new asynchronous user interface, these tools have radically transformed the Isabelle user experience. This paper provides an overview of the main automatic proof and disproof tools.

1 Introduction

In the tradition of LCF-style interactive theorem provers [21], Isabelle [35] has long emphasized tactics: functions written in ML that operate on the proof state via a trusted inference kernel. Tactics discharge a proof goal directly or, more often, break it down into one or more subgoals that must then be tackled by other tactics. In the last decade, the structured Isar language [34,57] has displaced ML as the language of choice for Isabelle proofs, but the most important ML tactics are still available as Isar proof methods.

Much effort has been devoted to developing general-purpose proof methods (or tactics) that work equally well on all object logics supported by Isabelle, notably higher-order logic (HOL) [20] and Zermelo–Fraenkel set theory (ZF) [37,38]. The most important methods are the simplifier, which rewrites the goal using equations as oriented rewrite rules, and the tableau prover (Section 2). These are complemented by specialized decision procedures, especially for arithmetic. For the users of an interactive theorem prover, one of the main challenges is to find out which proof methods to use and which arguments to specify.

Although proof methods are still the mainstay of Isabelle proofs, the last few years have seen the focus move toward advisory tools that work outside the LCF-style inference kernel. Some of these tools are very simple and yet surprisingly effective; for example, one searches Isabelle's libraries for a lemma that can prove the current goal directly, and another tries the most common proof methods.

The most important proof tool besides the simplifier and the tableau prover is probably Sledgehammer, which connects Isabelle with external resolution provers and SMT solvers (Section 3). It boasts a fairly high success rate on goals that cannot be discharged directly by standard proof methods: In a recent study involving older Isabelle proof scripts, Sledgehammer could prove 43% of the more difficult goals contained

C. Tinelli and V. Sofronie-Stokkermans (Eds.): FroCoS 2011, LNAI 6989, pp. 12–27, 2011.

in those proofs [6]. The addition of SMT solvers is recent and helps solve both arithmetic and nonarithmetic problems [6]. Sledgehammer works well in combination with structured Isar proofs: The new way of teaching Isabelle is to let students think up intermediate properties and rely on automatic tools to fill in the gaps, rather than teach them low-level tactics and have them memorize lemma libraries [41, §4].

As useful as they might be, most automatic proof tools are helpless in the face of an invalid conjecture. Novices and experts alike can enter invalid formulas and find themselves wasting hours (or days) on an impossible proof; once they identify and correct the error, the proof is often easy. To make proving more enjoyable and productive, Isabelle includes counterexample generators that complement the proof tools. The main ones are Quickcheck (Section 4) and Nitpick (Section 5).

Quickcheck [3] combines Isabelle's code generation infrastructure with random testing, in the style of the QuickCheck tool for Haskell [14]. It analyses the definitions of inductively defined predicates to generate values that satisfies them by construction [11] and has recently been extended with exhaustive testing and narrowing.

A radically different approach is based on systematic model enumeration using a SAT solver. This approach was pioneered by the tool Refute [54] and is now embodied by Nitpick [8]. Nitpick looks for finite fragments (substructures) of infinite countermodels, soundly approximating problematic constructs. Common Isabelle idioms, such as inductive and coinductive predicates and datatypes as well as recursive and corecursive functions, are treated specially to ensure efficient SAT solving. The actual reduction to SAT is performed by the Kodkod library [53] (the Alloy Analyzer's [25] backend).

With so many tools at their disposal, users run the risk of forgetting to invoke them at the right point; this is especially true for the counterexample generators, given that humans have a natural tendency to trust their own conjectures. For this reason, the proof and disproof tools can be set up to run automatically in parallel for a few seconds on all newly entered conjectures. They can of course also be launched at any point in a proof with a more liberal time limit. Either mode of operation exploits multiple processor cores if they are available, and Sledgehammer also sends its problems to remote servers to further distribute the load.

2 Standard Proof Methods

Isabelle provides the user with an array of general-purpose proof methods that perform proof search. We discuss the most important ones.

2.1 Simplification

Just as in ACL2 [26], simplification is the main workhorse in Isabelle. It performs conditional, contextual rewriting with a number of hooks for customizations:

- *Pattern-driven simplification procedures* that derive and apply rewrite rules dynamically. Many such procedures are preinstalled, notably arithmetic simplification procedures for numerals and symbolic terms.
- *Special solvers for conditional rewrite rules*. Typical examples are fragments of linear arithmetic and a transitive closure prover for arbitrary transitive relations.

– *Special "loopers"* that massage the goal after each round of simplification. Case splitting methods are provided this way.

The power of the simplifier is due to these extensions to rewriting together with the vast and growing library of registered rewrite rules.

2.2 Auto and Co.

On the user level, the simplifier is eclipsed by *auto*, a proof method that interleaves simplification with a small amount of proof search. It is impossible to describe succinctly what *auto* does due to its heuristic, ad hoc nature. Its great strength is its ability to discharge the easy parts of a goal and leave the user with the more difficult ones. This helps the user to quickly focus on the core of a problem.

Strengthened versions of *auto* perform more sophisticated proof search, while still interleaving it with simplification. The search is based on tableau methods [39]. These methods are often useful, but since search is involved, not only are they slower than the simplifier and *auto*, they are endgame provers that do not provide any hints when they fail to prove the goal.

2.3 Blast and Metis

The tableau implementation mentioned above can be very slow because every inference step is performed directly on the proof state, via the Isabelle kernel. For more performance, users can choose *blast* [40], a tableau prover written directly in ML that bypasses the kernel; once a proof has been found, it is replayed in the kernel to check it. The *blast* method outperforms the kernel-based tableau implementation by a wide margin but is no match for the best automatic provers. Nor does it know about simplification, which is a great loss.

Taking this one step further, Metis is a resolution theorem prover written in ML by Hurd [24]. Metis is sufficiently capable that it is a respectable competitor at CASC [51]. It has been ported to Isabelle and follows the same philosophy as *blast*: The proof search is performed directly in ML, and any proof found is checked by the Isabelle kernel.

The *blast* method relies on an extensible lemma database that drives the search and that is preconfigured to reason about sets, functions and relations, which makes it quite user-friendly. In contrast, Isabelle's version of Metis knows only about pure logic and derives its knowledge about other operators from explicitly supplied lemmas. Although Metis can be invoked directly, in practice Metis calls are almost always generated by Sledgehammer for reconstructing external resolution proofs (Section 3.4).

3 Sledgehammer: Proof Discovery Using External Provers

Sledgehammer [31, 42] is Isabelle's subsystem for harnessing the power of first-order automatic theorem provers. Given a conjecture, it heuristically selects a few hundred relevant facts (lemmas, definitions, or axioms) from Isabelle's libraries, translates them to first-order logic along with the conjecture, and delegates the proof search to external

resolution provers (E [48], SPASS [56], and Vampire [44]) and SMT solvers (CVC3 [2], Yices [16], and Z3 [33]). Sledgehammer is very effective [9] and has achieved great popularity with users, novices and experts alike.

3.1 Relevance Filtering

Most automatic provers perform poorly in the presence of thousands of axioms. Sledgehammer employs a simple relevance filter [32] to extract a few hundred facts from Isabelle's libraries that seem relevant to the problem at hand. Despite its simplicity, this filter greatly improves Sledgehammer's success rate.

The filter works iteratively. The first iteration selects facts that share all or nearly all of their constants (symbols) with the conjecture. Further iterations also include facts that share constants with previously selected facts, until the desired number of facts is reached. Observing that some provers cope better with large axiom bases than others, that number was optimized independently for each prover.

3.2 Translation to First-Order Logic

Isabelle's formalism, polymorphic higher-order logic with type classes [59], is much richer than the first-order logics supported by the automatic provers. Sledgehammer relies on different translations depending on the class of prover [6, 31].

For resolution provers, standard techniques are employed to translate HOL formulas to classical first-order logic: λ-abstractions are rewritten to combinators, and curried functions are passed varying numbers of arguments by means of an explicit apply operator. Until recently, the translation of types was unsound: It provided enough type information to enforce correct type class reasoning but not to specify the type of every term. (Because the proofs are rechecked by Isabelle's inference kernel, soundness is not crucial.) The current implementation safely erases most type information by inferring type monotonicity [7, 15], resulting in a sound and efficient encoding.

For SMT solvers, the translation maps equality and arithmetic operators to the corresponding SMT-LIB [43] concepts. The SMT-LIB logic is many-sorted, which would seem to make it more appropriate to encode HOL typing information than classical first-order logic, but it does not support polymorphism. The solution is to monomorphize the formulas: Polymorphic formulas are iteratively instantiated with relevant ground instances of their polymorphic constants. This process is iterated to obtain the monomorphized problem. Partial applications are translated using an apply operator, but in contrast with the combinator approach used when communicating with resolution provers, λ-abstractions are lifted into new rules, thereby introducing fresh constants.

3.3 Invocation of External Provers

Sledgehammer lets the external provers run in parallel, either locally or remotely. On a typical Isabelle installation, E, SPASS, and Z3 are run on the user's machine, whereas Vampire and the SInE metaprover [23] are provided via the remote SystemOnTPTP service [50]. Users can also enable CVC3 and Yices.

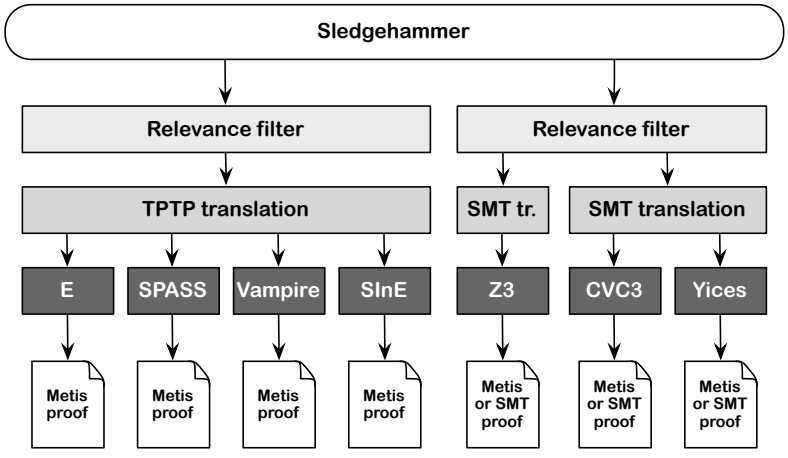


Fig. 1. Sledgehammer's architecture

Figure 1 depicts the architecture, omitting proof reconstruction and minimization. Two instances of the relevance filter are run, to account for different sets of built-in constants. The relevant facts and the conjecture are translated to the TPTP [52] or SMT version of first-order logic, and the resulting problems are passed to the provers. The translation for Z3 is done slightly differently than for CVC3 and Yices to profit from Z3's support for nonlinear arithmetic.

Third-party provers should ideally be bundled with Isabelle and ready to be used without requiring configuration. Isabelle includes CVC3, E, SPASS, and Z3 executables for the major hardware platforms; users can download Yices and Vampire, whose licenses forbid redistribution, but most simply run Vampire remotely on SystemOnTPTP. In addition, we set up a server in Munich in the style of SystemOnTPTP for running CVC3 and Z3 remotely.

Remote servers are satisfactory for proof search, at least when they are up and running and the user has Internet access. They also help distribute the load: Unless the user's machine has eight processor cores, it would be reckless to launch four resolution provers and three SMT solvers and expect the Isabelle user interface to remain responsive. The parallel invocation of provers is invaluable: Running E, SPASS, and Vampire together for five seconds solves as many problems as running a single prover for two minutes [9, §8].

3.4 Proof Reconstruction

In keeping with the LCF philosophy [21], Isabelle theorems can only be generated within a small inference kernel. It is possible to bypass this safety mechanism, if some external tool is to be trusted as an oracle, but all oracle inferences are tracked.

For resolution provers, Sledgehammer performs true proof reconstruction by running Isabelle's built-in resolution prover, Metis, supplying it with the short list of facts used in the proof found by the prover. Given only a handful of facts, Metis usually

succeeds within milliseconds. Since Metis has to re-find the proof, the external provers are essentially used as very precise relevance filters.

As an example, consider the conjecture "$length\ (tl\ xs) \leq length\ xs$", which states that the length of a list's tail (its "cdr") is less than or equal to the length of the entire list. Thanks to Vampire, Sledgehammer finds the following proof:

by (*metis append_Nil2 append_eq_conv_conj drop_eq_Nil drop_tl tl.simps*(1))

Proof reconstruction using Metis loses about 4% of resolution proofs because Metis times out, typically because the proof found by the external prover is too long. Sledgehammer then falls back on a detailed Isabelle proof, expressed in the structured Isar language. While the detailed output is primarily designed for replaying resolution proofs, it also has a pedagogical value. Unlike Isabelle's automatic tactics, which are black boxes, the proofs delivered by Sledgehammer can be inspected and understood, as in the example below:

proof –
  have "$tl\ [] = []$" **by** (*metis tl.simps*(1))
  hence "$\exists u.\ xs\ @\ u = xs \wedge tl\ u = []$" **by** (*metis append_Nil2*)
  hence "$tl\ (drop\ (length\ xs)\ xs) = []$" **by** (*metis append_eq_conv_conj*)
  hence "$drop\ (length\ xs)\ (tl\ xs) = []$" **by** (*metis drop_tl*)
  thus "$length\ (tl\ xs) \leq length\ xs$" **by** (*metis drop_eq_Nil*)
qed

The generated proofs often require some postediting to make them syntactically correct. Efforts are underway to make the generated output both more robust and more concise.

On the SMT side of things, proofs that involve no arithmetic reasoning steps can usually be replayed by Metis; otherwise, step-by-step proof replay is supported for Z3 [10], whereas CVC3 and Yices can be invoked as oracles. Z3 proof replay relies extensively on Isabelle's simplifier, tableau prover, and arithmetic decision procedures. Certificates make it possible to store Z3 proofs alongside Isabelle formalizations, allowing proof replay without Z3; only if the formalizations change must the certificates be regenerated. Using SMT solvers as oracles requires trusting both the solvers and the translation to first-order logic, so it is generally frowned upon.

3.5 Proof Minimization

The external provers frequently use many more facts than are necessary. Sledgehammer's minimization tool takes the set of used facts returned by a prover and repeatedly invokes the prover with subsets of the facts to find a minimal set. Depending on the number of initial facts, it relies either on a naive linear algorithm that attempts to remove one fact at a time or on a binary algorithm that recursively bisects the facts [9, §7].

Minimization often improves Metis's performance and success rate, while removing clutter from the Isabelle formalizations. For some provers, it is difficult or impossible to extract the list of used facts from the proof; minimization is then the only option. For example, the detailed proofs returned by CVC3 always refer to all facts, whether they are actually needed or not, and there is no easy criterion to isolate the needed facts.

4 Quickcheck: Counterexample Generation by Testing

Isabelle's proof methods and Sledgehammer are effective for proving valid conjectures, but given an invalid conjecture they normally fail to detect the invalidity, let alone produce an informative counterexample. This is where Quickcheck comes into play.

Quickcheck was originally modeled after the QuickCheck tool for Haskell [14], which tests user-supplied properties of a Haskell program for randomly generated values. We recently extended Quickcheck with exhaustive and narrowing-based testing as complements to random testing. Exhaustive testing checks the formula for every possible set of values up to a given bound, as in SmallCheck [46], and hence finds counterexamples that random testing might miss. Narrowing can be more precise and more efficient than the other two approaches because it considers the formula symbolically, instead of testing a finite set of ground values.

Thanks to a static data-flow analysis inspired by logic programming [11], Quickcheck derives test data generators that take premises into account to help avoid the vacuous test cases that plague most specification testing tools.

4.1 Random and Exhaustive Testing

Quickcheck's random testing strategy repeatedly evaluates the conjecture with pseudo-random values for its free variables. The procedure is parameterized by a size bound on the generated values and the number of tests to perform. The distribution is biased toward smaller values [3, §4].

In principle, Quickcheck could use the Isabelle simplifier to evaluate the conjecture for specific values of its free variables, but it is much more efficient to translate the conjecture and related definitions to an ML (or Haskell) program, exploiting Isabelle's code generation infrastructure [22]: The ML runtime environment can check millions of test cases within seconds, which is thousands of times faster than the simplifier.

Random testing tends to be fast and sometimes finds large counterexamples. Indeed, the QuickCheck tool for Haskell includes a minimizer to reduce overly large counterexamples, a refinement that our Quickcheck implementation currently lacks. But random testing can easily miss counterexamples, even seemingly obvious ones. It also struggles with conjectures that have hard-to-satisfy premises.

An alternative strategy is exhaustive testing, which systematically enumerates values up to a size bound (e.g., all lists of length up to 5). This ensures that all possible variable assignments up to a given size are tested. Hence, if there is a small enough counterexample, it will be found. The main drawback of this strategy is that the number of test cases quickly explodes with increasing size bounds.

Through empirical testing we found the two strategies to be roughly comparable on most types of formula, but exhaustive testing tends to be more successful on conjectures with hard-to-satisfy premises, simply because it will encounter the few small values that fulfill the conditions if such values exist, whereas random testing might miss them. The following conjecture about lists illustrates this point:

$$nth\ (xs\ @\ ys)\ (length\ xs + n) = nth\ xs\ n$$

The *nth* function returns the element at a given index in a list, and @ is the append operator. The conjecture attempts to relate the elements of *xs* @ *ys* with those of *ys*, but a typo slipped in: The right-hand side should read *nth ys n*. Exhaustive testing immediately finds a counterexample with $xs = [\mathsf{a}_1]$ and $ys = [\mathsf{a}_2]$ (for $\mathsf{a}_1 \neq \mathsf{a}_2$). Random testing typically fails to find the counterexample, even with hundreds of iterations, because randomly chosen values for *n* are almost always out of bounds. Since such examples occur frequently in practice, we have now made exhaustive testing the default strategy.

4.2 Test Data Generation

Random and exhaustive testing generate values without analyzing the conjecture. This can lead to many vacuous test cases, as in this simple example:

$$length\ xs = length\ ys \wedge zip\ xs\ ys = zs \Longrightarrow map\ fst\ zs = xs \wedge map\ snd\ zs = ys$$

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.

We recently extended Quickcheck with test data generators that construct values in a bottom-up fashion, simultaneously testing the conjecture and generating appropriate values [11]. Briefly, we synthesize the test data generator associated with a given premise by reformulating the premise as Horn clauses and computing their data-flow dependencies; from this data-flow analysis, we synthesize generators that directly compute appropriate value.

When transforming the premises to Horn clauses, we replace *n*-ary functions with $(n+1)$-ary predicates; this gives more freedom to the data-flow analysis, which can then invert functions. The data-flow analysis is an extension of a classic analysis from logic programming. To execute a predicate, its arguments are classified as input or output, made explicit by means of modes. A *mode* is a data-flow assignment that annotates all arguments of a predicate as input (*i*) or output (*o*). For example, the binary predicate of type $\alpha\ list \to nat \to bool$ corresponding to the function *length* supports several modes:

- From the first argument *xs*, we can compute the second argument by evaluating *length xs*. This corresponds to the mode $i \to o \to bool$.
- Inversely, we can enumerate lists of a given length: $o \to i \to bool$.
- Given a list and a natural number, we can check whether the list's length equals that number: $i \to i \to bool$.
- Or we can simply enumerate all pairs (xs, n) such that $length\ xs = n$. This is the mode $o \to o \to bool$.

In the classic analysis, a mode is only possible if the Horn clauses allow a complete data-flow from input to output values. For Quickcheck, if the mode analysis fails to produce a complete mode assignment because the values of some variables are not constrained by the premises, we fall back on the random or exhaustive strategy to fill in the gaps in the data flow. For example, given the Horn clause $P\ x \Longrightarrow Q\ x\ y$, where *P* supports

the modes $i \to bool$ and $o \to bool$, the classic analysis fails to find a consistent mode assignment for Q with mode $o \to o \to bool$ because y is unconstrained. To generate values for x and y that fulfill Q, we can generate x values using P with $o \to bool$ and set y to an arbitrary value, then check $Q\ x\ y$.

If the conjecture is polymorphic, we can instantiate the type variables with any concrete type for refuting it. Older versions of Quickcheck instantiated type variables with the type of integers, but it is usually preferable to use a small finite type instead, so that existential conjectures $\exists x :: \alpha.\ P\ x$ can be refuted by a finite number of P tests.

4.3 Narrowing

The random and exhaustive strategies suffer from two important limitations: They cannot refute propositions that existentially quantify over infinite types, and they often repeatedly test formulas with values that check essentially the same execution (e.g., because of symmetries).

Both issues arise from the use of ground values and can be addressed by evaluating the formula symbolically. The technique is called narrowing and is well known from term rewriting. The main idea is to evaluate the conjecture with partially instantiated terms and to progressively refine these terms as needed. Technically, this can be achieved in at least three different ways:

1. Target a language that natively supports narrowing, such as the functional-logical language Curry [1], instead of ML.
2. Simulate narrowing by generating a functional program that includes its own refinement algorithm [46].
3. Simulate narrowing by embedding the narrowing-based execution with a library of combinators [18,30] in a functional language.

We tried out the first two approaches and found that the second approach is faster. The third approach looks promising but would require a more involved translation.

The main benefit of narrowing is its generality: Unlike the random and exhaustive strategies, it can refute existential quantifications over infinite types. Consider the following conjecture:

$$\forall n.\ \exists m :: nat.\ n = Suc\ m$$

To disprove it, we must exhibit a natural number n such that $\forall m :: nat.\ n \neq Suc\ m$. Taking a symbolic view, if we choose $n = 0$, it is easy to see that $n \neq Suc\ m$ is true for every natural number m without having to instantiate m.

The above example is perhaps too simple to be convincing. A more realistic example is based on the observation that the palindrome $[a, b, b, a]$ can be split into the list $[a, b]$ and its reverse $[b, a]$. Generalizing this to arbitrary lists, we boldly conjecture that

$$rev\ xs = xs \Longrightarrow \exists ys.\ xs = ys\ @\ rev\ ys$$

The narrowing approach immediately finds the counterexample $xs = [\mathsf{a}_1]$, inferring that there is no witness for ys in the infinite domain of lists: If ys is empty, $ys\ @\ rev\ ys$

$= [] \neq [a_1]$, and if *ys* is not empty, *ys* @ *rev ys* consists of at least two elements and hence cannot be equal to $[a_1]$.

Narrowing tends to scale better than the random and exhaustive strategies. Consider red–black trees, a binary search data structure with two kinds of node, red and black, that must satisfy a sophisticated invariant involving node coloring. The invariant is captured by a predicate *is_rbt*. If the *delete* operation is properly implemented, the following property should hold:

$$is\_rbt\ t \Longrightarrow is\_rbt\ (delete\ k\ t)$$

The premise *is_rbt t* ensures that the tree *t* has a black root node, and in fact, after a few refinements, narrowing will only test symbolic values satisfying this property, already pruning away about half of the overall test cases. As expected, narrowing finds many more counterexamples than random and exhaustive testing on this kind of example. Interestingly, it even performs slightly better than a custom generator that constructs well-formed trees using a sequence of *insert* operations.

5 Nitpick: Countermodel Generation Using SAT Solvers

Irrespective of which strategy is used, Quickcheck recasts the conjecture to disprove into a functional program. An alternative is to let a SAT solver enumerate models of the negated conjecture and relevant definitions and axioms. This approach is implemented in a separate tool called Nitpick [8], which relies on the highly optimized Kodkod library [53] for the actual reduction to SAT.

Given a conjecture, Nitpick (via Kodkod and the SAT solver) searches for a standard set-theoretic model that falsifies it while satisfying any relevant axioms and definitions. Unlike Quickcheck, which performs its sophisticated code transformations using the Isabelle inference kernel, Nitpick does not certify any of its results and must be trusted.

Nitpick's design was inspired by its predecessor Refute [54], which performed a direct reduction to SAT. Nitpick works by systematically enumerating the domain cardinalities for the atomic types (type variables and other uninterpreted types) occurring in the conjecture and generates one Kodkod problem (and ultimately one SAT problem) per cardinality specification [5]. To exhaust all models up to a given cardinality bound k for a formula involving n atomic types, it must in principle iterate through k^n combinations of cardinalities, but a sophisticated monotonicity inference helps prune the search space [7]. If the conjecture has a finite countermodel, the tool eventually finds it, unless it runs out of resources.

5.1 Basic Translation to Relational Logic

Kodkod's input is expressed in first-order relational logic (FORL), an idiosyncratic formalism that combines elements from first-order logic and relational calculus, extended with a transitive closure operator. SAT solvers are particularly sensitive to the encoding of problems, so special care is needed when translating HOL formulas to FORL. Whenever practicable, HOL constants are mapped to their FORL equivalents, rather than expanded to their definitions.

As a rule, HOL scalars are mapped to FORL singletons and functions are mapped to FORL relations accompanied by a constraint. An n-ary first-order function can be coded as an $(n+1)$-ary relation accompanied by a constraint. However, if the return type is *bool*, the function is more efficiently coded as an unconstrained n-ary relation. This allows formulas such as $A^+ \cup B^+ = (A \cup B)^+$ to be translated without taking a detour through ternary relations.

Higher-order quantification and functions bring complications of their own. For example, assuming the cardinality constraints $|\alpha| = 2$ and $|\beta| = 3$, we would like to translate $\forall g :: \beta \to \alpha.\ g\ x \neq y$ into something like

$$\forall g \subseteq \{\mathsf{a}_3, \mathsf{a}_4, \mathsf{a}_5\} \times \{\mathsf{a}_1, \mathsf{a}_2\}.\ (\forall a \in \{\mathsf{a}_3, \mathsf{a}_4, \mathsf{a}_5\}.\ |g(a)| = 1) \longrightarrow g(x) \neq y$$

but since Kodkod is first-order, the $\subseteq$ symbol is not allowed at the binding site; only $\in$ is. Skolemization solves half the problem, but for the remaining quantifiers we are forced to adopt an unwieldy n-tuple singleton representation of functions, where n is the cardinality of the domain. The n-tuple simply encodes g's function table. For the formula above, this gives

$$\forall G \in \{\mathsf{a}_1, \mathsf{a}_2\}^3.\ \overbrace{\big(\{\mathsf{a}_3\} \times \pi_1(G) \cup \{\mathsf{a}_4\} \times \pi_2(G) \cup \{\mathsf{a}_5\} \times \pi_3(G)\big)}^{g}(x) \neq y$$

where G is the triple corresponding to g and $\pi_i(G)$ is its ith component (i.e., the ith entry in the function table). In the body, we convert the singleton G to the relational representation, then we apply x on it. The singleton encoding is also used for passing functions to other functions; fortunately, two optimizations, function specialization and boxing [8, §5], make this rarely necessary.

5.2 Approximation of Infinite Types and Partiality

Because of the axiom of infinity, the type *nat* of natural numbers does not admit any finite models. To work around this, Nitpick considers finite subsets $\{0, 1, \ldots, K-1\}$ of *nat* and maps numbers $\geq K$ to the undefined value, denoted by $\star$ and coded as the empty set. Formulas of the form $\forall n :: nat.\ P\ n$ are treated as $(\forall n < K.\ P\ n) \wedge P\ \star$, which usually evaluates to either *False* (if $P\ i$ gives *False* for some $i < K$) or $\star$, but not to *True*, since we generally cannot determine statically whether $P\ K$, $P\ (K+1)$, ..., collectively represented by $P\ \star$, are true. Partiality leads to a Kleene three-valued logic, which is soundly encoded in Kodkod's two-valued logic.

5.3 Encoding of (Co)inductive Predicates

Isabelle lets users specify (co)inductive predicates p by their introduction rules and synthesizes a fixed point definition $p = \mathit{lfp}\ F$ or $p = \mathit{gfp}\ F$ behind the scenes. For performance reasons, Nitpick handles (co)inductive predicates specially rather than simply expanding *lfp* and *gfp* to their definitions.

An inductive predicate p is a fixed point, so Nitpick can use the equation $p = F\ p$ as the axiomatic specification of p. In general, this is unsound since it underspecifies p, but there are two important cases for which this method is sound:

- If the recursion in F is well-founded [12], the fixed point equation $p = F\ p$ admits exactly one solution that can safely be taken as p's specification.
- If p occurs negatively in the formula, these occurrences can be soundly replaced by a fresh constant q satisfying the axiom $q = F\ q$.

For the remaining positive occurrences of p, Nitpick unrolls the predicate a given number of times, as in bounded model checking [4]. The situation is mirrored for coinductive predicates: Positive occurrences are coded using the fixed-point equation, and negative occurrences are unrolled.

5.4 Encoding of (Co)inductive Datatypes

In contrast to Isabelle's constructor-oriented treatment of inductive datatypes, Nitpick's FORL axiomatization revolves around selectors and discriminators, following a standard Alloy idiom [28]. The selector/discriminator view is usually more efficient than the constructor view because it breaks high-arity constructors into several low-arity selectors, with correspondingly smaller function tables in the SAT encoding. For example, the type $\alpha\ list$ generated from $Nil :: \alpha\ list$ and $Cons :: \alpha \to \alpha\ list \to \alpha\ list$ is axiomatized in terms of the discriminators *nilp* and *consp* and the selectors *hd* and *tl*, which give access to a nonempty list's head and tail.

The FORL axiomatization specifies a subterm-closed finite substructure of lists. Examples of subterm-closed list substructures using traditional notation are $\{[], [0], [1]\}$ and $\{[], [1], [2,1], [0,2,1]\}$. On the other hand, the set $L = \{[], [1,1]\}$ is not subterm-closed, because $tl\,[1,1] = [1] \notin L$. Given cardinalities for the list type and the item type, the SAT solver enumerates all corresponding subterm-closed list substructures.

Nitpick supports coinductive datatypes, even though Isabelle does not provide a high-level mechanism for defining them. Users can define custom coinductive datatypes from first principles and tell Nitpick to substitute its efficient FORL axiomatization for their definitions.

6 Related Work

Isabelle is not the only interactive theorem prover that provides a palette of automatic proof and disproof tools. We briefly review what the other popular provers have to offer.

- HOL4 [20, 49] includes the original version of Metis [24] and an integration of SMT solvers [55] with proof reconstruction for Z3 [10].
- PVS includes a Quickcheck-like random testing tool [36] and integrates the SMT solver Yices as an oracle [47].
- For Mizar, the MizAR web service [45] is a recent addition that exploits external resolution provers in the style of Sledgehammer.
- The Sedan version of ACL2 includes a counterexample generator based on random testing [13]. The tool analyses the goal to compute dependencies between free variables, similar to Quickcheck's data-flow analysis.

- Although Coq has a considerable user base, advisory tools are conspicuously missing. An SMT integration with proof certification is in the works [27].
- Earlier versions of the Agda proof assistant included a version of QuickCheck [17], but like the original QuickCheck for Haskell it required users to write dedicated data generators for custom datatypes. The Agsy tool [29,30] implements narrowing for both counterexample generation and proof search. An integration of the equational prover Waldmeister is under development [19].

7 Conclusion

Isabelle offers a wide range of automatic tools for proving and disproving conjectures. Some of them are built into the theorem prover, but increasingly these activities are delegated to highly optimized external tools, such as resolution provers, SAT solvers, and SMT solvers. While there have been several attempts at integrating external provers and disprovers in various interactive theorem provers, Isabelle is probably the only interactive prover where external tools play such a prominent role, to the extent that they are now seen as indispensable by many if not most users.

In terms of usefulness, Sledgehammer is second only to the simplifier and tableau prover. But the counterexample generators also provide invaluable help and encourage a lightweight explorative style to formal proof development, as championed by Alloy [25]. Because it is so fast, Quickcheck is enabled by default to run on all conjectures. Users are so accustomed to its feedback that they rarely realize to what extent they benefit from it. Every now and then, Nitpick finds a counterexample beyond Quickcheck's reach. As developers of both tools, we frequently receive emails from users grateful to have been spared "several hours of hard work."

An explanation for Sledgehammer, Quickcheck, and Nitpick's success is that they are included with Isabelle and require no additional installation steps. External tools necessary to their operation are either included in the official Isabelle packages or accessible as online services. Multi-core architectures and remote servers help to bear the burden of (dis)proof, so that users can continue working on a manual proof while the tools run in the background.

Another important design goal for all three tools was one-click invocation. Users should not need to preprocess the goals, specify options, or implement custom data generators. Even better than one-click invocation is zero-click invocation, whereby the tools spontaneously run on newly entered conjectures. A more flexible user interface, such as the experimental jEdit-based PIDE [58], could help further here, by asynchronously dispatching the tools to tackle any unfinished proofs in the current proof document, irrespective of the text cursor's location.

Interactive theorem proving is still challenging, but thanks to a new generation of automatic proof and disproof tools and the wide availability of multi-core processors with spare CPU cycles, it is much easier and more enjoyable now than it was only a few years ago.

Acknowledgment. We thank Alexander Krauss, Mark Summerfield, and Thomas Türk for suggesting several textual improvements.

References

1. Antoy, S., Hanus, M.: Functional logic programming. Commun. ACM 53, 74–85 (2010)
2. Barrett, C., Tinelli, C.: CVC3. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 298–302. Springer, Heidelberg (2007)
3. Berghofer, S., Nipkow, T.: Random testing in Isabelle/HOL. In: Cuellar, J., Liu, Z. (eds.) SEFM 2004, pp. 230–239. IEEE C.S., Los Alamitos (2004)
4. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without bDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)
5. Blanchette, J.C.: Relational analysis of (co)inductive predicates (co)inductive datatypes, and (co)recursive functions. Softw. Qual. J. (to appear)
6. Blanchette, J.C., Böhme, S., Paulson, L.C.: Extending Sledgehammer with SMT Solvers. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS (LNAI), vol. 6803, pp. 116–130. Springer, Heidelberg (2011)
7. Blanchette, J.C., Krauss, A.: Monotonicity inference for higher-order formulas. J. Auto. Reas. (to appear)
8. 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)
9. Böhme, S., Nipkow, T.: Sledgehammer: Judgement day. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS (LNAI), vol. 6173, pp. 107–121. Springer, Heidelberg (2010)
10. Böhme, S., Weber, T.: Fast LCF-style proof reconstruction for Z3. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 179–194. Springer, Heidelberg (2010)
11. Bulwahn, L.: Smart test data generators via logic programming. In: Gallagher, J.P., Gelfond, M. (eds.) ICLP 2011 (Technical Communications). Leibniz International Proceedings in Informatics, vol. 11, pp. 139–150. Schloss Dagstuhl, Leibniz-Zentrum für Informatik (2011)
12. Bulwahn, L., Krauss, A., Nipkow, T.: Finding lexicographic orders for termination proofs in isabelle/HOL. In: Schneider, K., Brandt, J. (eds.) TPHOLs 2007. LNCS, vol. 4732, pp. 38–53. Springer, Heidelberg (2007)
13. Chamarthi, H.R., Dillinger, P., Kaufmann, M., Manolios, P.: Integrating testing and interactive theorem proving (2011), http://arxiv.org/pdf/1105.4394
14. Claessen, K., Hughes, J.: QuickCheck: A lightweight tool for random testing of Haskell programs. In: ICFP 2000, pp. 268–279. ACM, New York (2000)
15. Claessen, K., Lillieströ m, A., Smallbone, N.: Sort it out with monotonicity: Translating between many-sorted and unsorted first-order logic. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS (LNAI), vol. 6803, pp. 207–221. Springer, Heidelberg (2011)
16. Dutertre, B., de Moura, L.: The Yices SMT solver (2006), http://yices.csl.sri.com/tool-paper.pdf
17. Dybjer, P., Haiyan, Q., Takeyama, M.: Combining testing and proving in dependent type theory. In: Basin, D., Wolff, B. (eds.) TPHOLs 2003. LNCS, vol. 2758, pp. 188–203. Springer, Heidelberg (2003)
18. Fischer, S., Kiselyov, O., Shan, C.: Purely functional lazy non-deterministic programming. In: ICFP 2009, pp. 11–22. ACM, New York (2009)
19. Foster, S., Struth, G.: Integrating an automated theorem prover into Agda. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 116–130. Springer, Heidelberg (2011)
20. Gordon, M.J.C., Melham, T.F. (eds.): Introduction to HOL: A Theorem Proving Environment for Higher Order Logic. Cambridge University Press, Cambridge (1993)

21. Gordon, M.J.C., Milner, R., Wadsworth, C.P.: Edinburgh LCF: A Mechanised Logic of Computation. LNCS, vol. 78. Springer, Heidelberg (1979)
22. 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)
23. Hoder, K., Voronkov, A.: Sine qua non for large theory reasoning. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS (LNAI), vol. 6803, pp. 299–314. Springer, Heidelberg (2011)
24. Hurd, J.: First-order proof tactics in higher-order logic theorem provers. In: Archer, M., Di Vito, B., Muñoz, C. (eds.) Design and Application of Strategies/Tactics in Higher Order Logics, pp. 56–68 (2003); No. CP-2003-212448 in NASA Technical Reports
25. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press, Cambridge (2006)
26. Kaufmann, M., Manolios, P., Moore, J.S.: Computer-Aided Reasoning: An Approach. Kluwer, Dordrecht (2000)
27. Keller, C.: Cooperation between SAT, SMT provers and Coq
28. Kuncak, V., Jackson, D.: Relational analysis of algebraic datatypes. In: Gall, H.C. (ed.) ESEC/FSE 2005. ACM, New York (2005)
29. Lindblad, F.: Higher-order proof construction based on first-order narrowing. Electr. Notes Theor. Comput. Sci. 196, 69–84 (2008)
30. Lindblad, F.: Property directed generation of first-order test data. In: Morazán, M. (ed.) TFP 2007, pp. 105–123. Intellect, Bristol (2008)
31. Meng, J., Paulson, L.C.: Translating higher-order clauses to first-order clauses. J. Auto. Reas. 40(1), 35–60 (2008)
32. Meng, J., Paulson, L.C.: Lightweight relevance filtering for machine-generated resolution problems. J. Applied Logic 7(1), 41–57 (2009)
33. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
34. Nipkow, T.: A tutorial introduction to structured Isar proofs (2011), http://isabelle.in.tum.de/dist/Isabelle/doc/isar-overview.pdf
35. Nipkow, T., Paulson, L.C., Wenzel, M.T.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002)
36. Owre, S.: Random testing in PVS. In: AFM 2006 (2006)
37. Paulson, L.C.: Set theory for verification: I. From foundations to functions. J. Auto. Reas. 11(3), 353–389 (1993)
38. Paulson, L.C.: Set theory for verification: II. Induction and recursion. J. Auto. Reas. 15(2), 167–215 (1995)
39. Paulson, L.C.: Generic automatic proof tools. In: Veroff, R. (ed.) Automated Reasoning and its Applications: Essays in Honor of Larry Wos, pp. 23–47. MIT Press, Cambridge (1997)
40. Paulson, L.C.: A generic tableau prover and its integration with Isabelle. J. Univ. Comp. Sci. 5(3), 73–87 (1999)
41. Paulson, L.C., Blanchette, J.C.: Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers. In: Sutcliffe, G., Ternovska, E., Schulz, S. (eds.) IWIL 2010 (2010)
42. Paulson, L.C., Susanto, K.W.: Source-level proof reconstruction for interactive theorem proving. In: Schneider, K., Brandt, J. (eds.) TPHOLs 2007. LNCS, vol. 4732, pp. 232–245. Springer, Heidelberg (2007)
43. Ranise, S., Tinelli, C.: The SMT-LIB standard: Version 1.2 (2006), http://goedel.cs.uiowa.edu/smtlib/papers/format-v1.2-r06.08.30.pdf
44. Riazanov, A., Voronkov, A.: The design and implementation of Vampire. AI Comm. 15(2-3), 91–110 (2002)

45. Rudnicki, P., Urban, J.: Escape to ATP for Mizar. In: PxTP 2011 (2011)
46. Runciman, C., Naylor, M., Lindblad, F.: SmallCheck and Lazy SmallCheck: Automatic exhaustive testing for small values. In: Haskell Symposium 2008, pp. 37–48. ACM, New York (2008)
47. Rushby, J.M.: Tutorial: Automated formal methods with PVS, SAL, and Yices. In: Hung, D.V., Pandya, P. (eds.) SEFM 2006, p. 262. IEEE, Los Alamitos (2006)
48. Schulz, S.: System description: E 0.81. In: Basin, D., Rusinowitch, M. (eds.) IJCAR 2004. LNCS (LNAI), vol. 3097, pp. 223–228. Springer, Heidelberg (2004)
49. Slind, K., Norrish, M.: A brief overview of HOL4. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 28–32. Springer, Heidelberg (2008)
50. Sutcliffe, G.: System description: SystemOnTPTP. In: McAllester, D. (ed.) CADE 2000. LNCS (LNAI), vol. 1831, pp. 406–410. Springer, Heidelberg (2000)
51. Sutcliffe, G.: The CADE-21 automated theorem proving system competition. AI Commun. 21(1), 71–82 (2008)
52. Sutcliffe, G., Zimmer, J., Schulz, S.: TSTP data-exchange formats for automated theorem proving tools. In: Zhang, W., Sorge, V. (eds.) Distributed Constraint Problem Solving and Reasoning in Multi-Agent Systems. Frontiers in Artificial Intelligence and Applications, vol. 112, pp. 201–215. IOS Press, Amsterdam (2004)
53. Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 632–647. Springer, Heidelberg (2007)
54. Weber, T.: SAT-Based Finite Model Generation for Higher-Order Logic. Ph.D. thesis, Dept. of Informatics, T.U. München (2008)
55. Weber, T.: SMT solvers: New oracles for the HOL theorem prover. In: VSTTE 2009 (2009)
56. Weidenbach, C.: Combining superposition, sorts and splitting. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, pp. 1965–2013. Elsevier, Amsterdam (2001)
57. Wenzel, M.: Isabelle/Isar—a generic framework for human-readable proof documents. In: Matuszewski, R., Zalewska, A. (eds.) From Insight to Proof—Festschrift in Honour of Andrzej Trybulec. Studies in Logic, Grammar and Rhetoric, vol. 10(23). University of Białystok (2007)
58. Wenzel, M.: Asynchronous proof processing with Isabelle/Scala and Isabelle/jEdit. In: Coen, C.S., Aspinall, D. (eds.) UITP 2010 (2010)
59. Wenzel, M.: Type classes and overloading in higher-order logic. In: Gunter, E.L., Felty, A.P. (eds.) TPHOLs 1997. LNCS, vol. 1275, pp. 307–322. Springer, Heidelberg (1997)

Size-Change Termination and Satisfiability for Linear-Time Temporal Logics

Martin Lange

School of Electr. Eng. and Computer Science, University of Kassel, Germany

Abstract. In the automata-theoretic framework, finite-state automata are used as a machine model to capture the operational content of temporal logics. Decision problems like satisfiability, subsumption, equivalence, etc. then translate into questions on automata like emptiness, inclusion, language equivalence, etc. Linear-time temporal logics like LTL, PSL and the linear-time μ-calculus have relatively simple translations into alternating parity automata, and this automaton model is closed under all Boolean operations with very simple constructions. Thus, the typical decision problems for such linear-time temporal logics reduce relatively simply to the emptiness problem for alternating parity automata. In this paper we present a method for decision this emptiness problem without going through intermediate automaton models like nondeterministic ones. The method is a direct adaptation of the size-change termination principle which was orgininally used to decide termination of abstract functional programs.

1 Introduction

Temporal logics are some of the most well-known and established tools for the specification of systems evolving in time. In computer science, they are mainly interesting as formal languages used to describe, reason about, analyse and verify program behaviour [7].

Temporal logics come in two different kinds depending on the nature of time underlying the models that they get interpreted about: linear-time and branching-time [20,21,25]. The viewpoint of linear time is that every moment in time has a unique successor, i.e. the future is determined. In branching time, a moment may have several successors. Here we only deal with linear time, namely we consider the well-known simple linear-time temporal logic LTL [16], as well as two of its extensions: a core of the industry standard property specification language PSL [2] which extends LTL with semi-regular expressions in order to remedy LTL's weakness of not being able to define all ω-regular properties; as well as the linear-time μ-calculus [22,3] which uses second-order quantification in the form of least and greatest fixpoints in order to obtain higher expressivity.

A prominent methodology for obtaining decision procedures is the automata-theoretic framework. It is particularly suitable for linear-time logics since their models can be seen as infinite words which immediately links logics and automata as two different specification formalisms for languages of such words. Logics are

C. Tinelli and V. Sofronie-Stokkermans (Eds.): FroCoS 2011, LNAI 6989, pp. 28–39, 2011.

often more natural to use as specification languages but automata are often closer to a decision procedure. Hence, translations from formulas to automata preserving their set of models are desirable, and they exist for the aforementioned logics.

It is known that every LTL formula can be translated into a nondeterministic Büchi automaton [26] with an expontial blow-up. The same holds for PSL [4] although the blow-up is in general doubly exponential. LT_μ can also be translated into such automata at a singly exponential blow-up only [26].

Translations into nondeterministic automata are particularly useful in order to decide satisfiability problems because satisfiability on the logical side corresponds to non-emptiness on the automata side, and non-emptiness problems for nondeterministic automata are usually solved via simple reachability problems on graphs. Other problems, however, in particular the universality and inclusion problem are as difficult for nondeterministic automata as the satisfiability problem for the corresponding temporal logics is. Note that on the logical side, problems like validity, subsumption and equivalence easily reduce to the satisfiability problem. This has led to the use of a richer automaton model: alternating automata. They typically enable a simple translation from formulas and on top of that a more difficult decision procedure as opposed to nondeterministic ones which usually come with a difficult translation and then simpler decision procedures. If "simple" means "polynomial" and "difficult" means "exponential" then the route via alternating automata may even be better in terms of complexity.

Translations from the linear-time temporal logics mentioned above into alternating automata are known [23,24,14,10]. In order to obtain decision procedures for these logics, one then only needs decision procedures for the corresponding problems on alternating automata. In fact, it suffices to be able to solve the emptiness problem for alternating automata just like it suffices to solve the satisfiability problem for temporal logics in order to solve all sorts of other problems through simple reductions.

The standard way to solve the emptiness problem for alternating automata has been using translations into nondeterministic automata. It may be the surprising simplicity of the Miyano-Hayashi construction [15] translating alternating Büchi into nondeterministic Büchi automata in comparison to the problem of turning a nondeterministic one into a deterministic one, that has put a brake onto research on different and possibly direct methods for the emptiness problem for alternating automata. This construction can be generalised to richer acceptance conditions like Streett automata [6], yet it still aims at translating into a nondeterministic model first.

Here we propose a different and direct method for the analysis of the emptiness problem for alternating parity automata. It originates from termination analysis for abstract functional programs and is called *size-change termination* (SCT) [13]. It is noted that the problem underlying this particular termination analysis can be solved by a reduction to the inclusion problem for nondeterministic Büchi automata but, since this requires complementation, SCT is proposed.

SCT has not passed unnoticed in the world of temporal logics and automata on infinite words: it has first been used to decide validity for the linear-time μ-calculus [5] and then prosed as a method to decide universality and inclusion for nondeterministic Büchi automata [8,9,1]. In fact, the first real work in this area is Büchi's original complementation proof for nondeterministic Büchi automata since the decision problems based on SCT use the same techniques. They are often called *Ramsey-based* because their correctness proof usually relies on the famous combinatorial Ramsey Theorem [17].

This paper is organised as follows. In Sect. 2 we recall the three important temporal logics mentioned above: LTL, PSL and the linear-time μ-calculus. In Sect. 3 we recall alternating parity automata and various subclasses thereof, and sketch how their emptiness problems characterise typical problems like satisfiability, subsumption, etc. for the temporal logics at hand. In Sect. 4 we describe an SCT based method to decide emptiness of alternating parity automata.

2 Linear-Time Temporal Logics

2.1 Infinite Words

Let $\mathcal{P} = \{p, q, \ldots\}$ be a countably infinite set of *atomic propositions*. Linear-time temporal logics are interpreted over ω-sequences of sets of such propositions: an *infinite word* w is an element of $(2^{\mathcal{P}})^\omega$. A *finite word* is a $v \in (2^{\mathcal{P}})^*$. We write ϵ for the empty word of length 0, and $|v|$ in general for the length of the finite word v.

If w is a word $A_0 A_1 \ldots$ for $A_i \subseteq \mathcal{P}$ then $w(i)$ is used to denote A_i. We write $w(i,j)$ to denote the finite subsequence $A_i \ldots A_j$. Note that $w(i,j) = \epsilon$ if $j < i$.

2.2 LTL

One of the simplest and most well-known linear-time temporal logics is LTL. Its formulas are built from atomic propositions using Boolean operators and two temporal operators: the *next state* operator $\bigcirc$, and the *until* operator $\mathtt{U}$. Formulas of LTL are given by the following grammar.

$$\varphi \quad ::= \quad q \mid \varphi \wedge \varphi \mid \neg\varphi \mid \bigcirc \varphi \mid \varphi \,\mathtt{U}\, \varphi$$

Formulas of LTL are interpreted in positions i of an infinite word as follows.

$$\begin{array}{lll}
w, i \models q & \text{iff} & q \in w(i) \\
w, i \models \varphi \wedge \psi & \text{iff} & w, i \models \varphi \text{ and } w, i \models \psi \\
w, i \models \neg\varphi & \text{iff} & w, i \not\models \varphi \\
w, i \models \bigcirc \varphi & \text{iff} & w, i+1 \models \varphi \\
w, i \models \varphi \,\mathtt{U}\, \psi & \text{iff} & \text{there is } j \geq i \text{ s.t. } w, j \models \psi \text{ and for all } h \text{ with } i \leq h < j \\
 & & \quad \text{we have } w, h \models \varphi
\end{array}$$

Further Boolean operators like disjunction $\varphi \vee \psi$, implication $\varphi \rightarrow \psi$, are defined as abbreviations in the usual way. Other temporal operators can also be derived, for instance $\mathtt{F}\,\varphi := \mathtt{true}\,\mathtt{U}\,\varphi$ ("finally") where $\mathtt{true} := q \vee \neg q$ for some $q \in \mathcal{P}$; $\mathtt{G}\,\varphi := \neg\,\mathtt{F}\,\neg\varphi$ ("generally"), etc.

We write $|\varphi|$ for the size of the formula φ, measured in terms of its DAG representation or, equivalently, the number of different subformulas.

Example 1. LTL can easily express properties concerning infinite occurrences of some atomic proposition in a word. For example, $\mathtt{G}\,\mathtt{F}\,q \wedge \mathtt{F}\,\mathtt{G}\,\neg p$ expresses that q holds in infinitely many positions and p holds almost everywhere, i.e. in all but finitely many positions.

A *model* for φ is a $w \in (2^{\mathcal{P}})^{\omega}$ s.t. $w, 0 \models \varphi$. We write $L(\varphi)$ for the set of all models of φ. The *satisfiability problem* is: given a formula φ, decide whether or not there is a model for it, i.e. whether or not $L(\varphi) \neq \emptyset$.

Proposition 1 ([19]). *The satisfiability problem for* LTL *is PSPACE-complete.*

We also describe two extensions of LTL in the following. They differ in their syntax and their semantics need more technicalities, but central concepts like formula size as well as set of models are defined as they are for LTL. Hence, we will not repeat them explicitly anymore.

2.3 PSL – An Extension of LTL

It is known that LTL cannot express counting properties like "q holds in every second position of a word" [27]. Note that this is an ω-regular property. There are several ways to overcome this weakness, for instance by introducing quantification over positions, i.e. by extending LTL with stronger logical connectives. PSL extends LTL with tools from the domain of formal languages, namely semi-extended regular expressions.

The language of all *Boolean expressions* over atomic propositions as above is given by the following grammar.

$$\zeta \;::=\; q \mid \zeta \wedge \zeta \mid \neg\zeta$$

Other Boolean operators can be defined as abbreviations. The satisfaction relation between a set $A \subseteq \mathcal{P}$ and a Boolean expression ζ is defined straightforwardly, i.e. $A \models \zeta$ iff ζ evaluates to 1 under the usual rules for Boolean connectives when all atomic propositions in A are set to 1 and all in $\mathcal{P} \setminus A$ are set to 0.

Semi-extended regular expressions (SERE) over Boolean expressions are built according to the following grammar.

$$\alpha \;::=\; \zeta \mid \alpha \cup \alpha \mid \alpha \cap \alpha \mid \alpha;\alpha \mid \alpha^*$$

where ζ is a Boolean expression as above. We write α^+ to abbreviate $\alpha;\alpha^*$.

A SERE is interpreted in a finite subword of an infinite $w \in (2^{\mathcal{P}})^\omega$ as follows. Note that such a subword is uniquely identified by two positions $i, j \in \mathbb{N}$ with $i \leq j$.

$$\begin{array}{ll}
w,i,j \models \zeta & \text{iff } i = j \text{ and } w(i) \models \zeta \\
w,i,j \models \alpha \cup \beta & \text{iff } w,i,j \models \alpha \text{ or } w,i,j \models \beta \\
w,i,j \models \alpha \cap \beta & \text{iff } w,i,j \models \alpha \text{ and } w,i,j \models \beta \\
w,i,j \models \alpha;\beta & \text{iff there is } h \text{ s.t. } i \leq h \leq j \text{ and } w,i,h \models \alpha \text{ and } w,h,j \models \beta \\
w,i,j \models \alpha^* & \text{iff there are } n \geq 0 \text{ and } h_0,\ldots,h_n \text{ s.t. } h_0 = i, h_n = j \text{ and} \\
 & \quad w,h_k,h_{k+1} \models \alpha \text{ for all } k = 0,\ldots,n-1
\end{array}$$

Note that in the first clause, satisfaction of a SERE in a finite word is reduced to satisfaction of a Boolean expression in a symbol of that word.

Formulas of PSL are then built by extending the syntax of LTL with operators that make use of SERE. Note that the standard of PSL [2] describes many operators for the logic; here we concentrate on two of them only – the "and then" and the "closure" operator. The constructions to follow can easily be extended to cover other PSL operators as well though.

$$\varphi \;::=\; q \mid \varphi \wedge \varphi \mid \neg\varphi \mid \varphi \,\mathtt{U}\, \varphi \mid \alpha \diamond\!\!\rightarrow \varphi \mid \mathtt{Cl}\,\alpha$$

The interpretation in positions of a word $w \in (2^{\mathcal{P}})^\omega$ extends the one for LTL given above by two clauses.

$$\begin{array}{ll}
w,i \models \alpha \diamond\!\!\rightarrow \varphi & \text{iff there is } j \geq i \text{ s.t. } w,i,j \models \alpha \text{ and } w,j \models \varphi \\
w,i \models \mathtt{Cl}\,\alpha & \text{iff for all } j \geq i \text{ exists } v \in (2^{\mathcal{P}})^* \text{ s.t. } w(i,j)v \models \alpha
\end{array}$$

Example 2. The aforementioned language $L = \{w \in (2^{\mathcal{P}})^\omega \mid \forall i \in \mathbb{N} : q \in w(2i)\}$ of all words in which q holds in every even position can be defined in PSL by the formula $\mathtt{Cl}(q;\mathtt{true})^*$. Equally, it is defined by $q \wedge \neg\big((\mathtt{true};\mathtt{true})^+ \diamond\!\!\rightarrow \bigcirc\neg q\big)$.

Proposition 2 ([4,12]). *The satisfiability problem for* PSL *is EXPSPACE-complete.*

The exponential increase in complexity compared to LTL is owed to the use of the intersection operator in semi-extended regular expressions. Note that these can be translated into nondeterministic finite automata (NFA) at a blow-up that is polynomial in the size of such an SERE but exponential in the nesting depth of the intersection operator. The logic obtained by replacing SERE with NFA is closely related to LTL with automata connective which also has a PSPACE-complete satisfiability problem [11].

2.4 The Linear-Time μ-Calculus

The Linear-Time μ-Calculus LT_μ is not directly an extension of LTL. It obtains ω-regular expressive power by adding least (finite iteration) and greatest (infinite

iteration) fixpoint operators to the fragment of LTL without the until operator. Let $\mathcal{V} = \{X, Y, \ldots\}$ be a countably infinite set of variable. Formulas of LT_μ are constructed as follows.

$$\varphi \quad ::= \quad q \mid X \mid \varphi \wedge \varphi \mid \neg\varphi \mid \bigcirc \varphi \mid \mu X.\varphi$$

where $q \in \mathcal{P}$ and $X \in \mathcal{V}$. We require that each formula is well-formed in the sense that every variable is bound by a binder *mu* at most once, and in every subformula $\mu X.\psi$ every occurrence of X is in the scope of an even number of negation symbols. For instance, $\mu X.\neg\mu Y.(\neg p \vee \bigcirc \neg Y) \wedge \bigcirc \neg X$ is well-formed because both variable occurrences are under two (different) nested negation operators.

Alongside the *least fixpoint quantifier* μ we introduce the *greatest fixpoint quantifier* ν via $\nu X.\varphi := \mu X.\neg\varphi[\neg X/X]$ where $\varphi[\psi/X]$ denotes the formula that is obtained from φ by replacing every free occurrence of X with ψ. Then the formula above can be written entirely without negation symbols as $\mu X.\nu Y.(p \wedge \bigcirc Y) \vee \bigcirc X$.

In order to interpret an LT_μ formula with free variables in a position of a word $w \in (2^{\mathcal{P}})^\omega$ we need the help of environments $\rho : \mathcal{V} \to 2^{\mathbb{N}}$. We write $\rho[X \mapsto P]$ for the environment that maps X to P and behaves like ρ on all other arguments.

$$\begin{aligned}
w, i, \rho &\models q && \text{iff } q \in w(i) \\
w, i, \rho &\models \varphi \wedge \psi && \text{iff } w, i, \rho \models \varphi \text{ and } w, i, \rho \models \psi \\
w, i, \rho &\models \neg\varphi && \text{iff } w, i, \rho \not\models \varphi \\
w, i, \rho &\models \mu X.\varphi && \text{iff } i \in \bigcap \{P \subseteq \mathbb{N} \mid P \supseteq \{j \mid w, j, \rho[X \mapsto P] \models \varphi\}\}
\end{aligned}$$

Example 3. The language of all words containing q in every even position can easily be defined in LT_μ: $\nu X.q \wedge \bigcirc\bigcirc X$.

The formula $\mu X.\nu Y.(p \wedge \bigcirc Y) \vee \bigcirc X$ mentioned above states that p holds in almost all positions. I.e. it is equivalent to the LTL formula $\mathtt{F\,G}\,p$.

Proposition 3 ([22]). *The satisfiability problem for* LT_μ *is PSPACE-complete.*

The reason for introducing LT_μ is the fact that it subsumes the two important temporal logics LTL and PSL. While this is also trivially true for PSL, LT_μ provides a clean (albeit not necessarily intuitive) syntax which is advantageous for the further treatment of these logics.

Proposition 4. *There are equivalence-preserving translations from ...*

- LTL *into* LT_μ *that incur a linear blow-up,*
- PSL *into* LT_μ *that incur a blow-up which is polynomial in the size of the formula and exponential in the nesting depth of the intersection operators [12].*

The translation from LTL into LT_μ is realised by the fact that $\varphi\,\mathtt{U}\,\psi$ can be expressed as $\mu X.\psi \vee (\varphi \wedge \bigcirc X)$. The translation from PSL is more complicated and uses the fact that SERE can be translated into NFA, as well as a close

resemblance between LT_μ formulas and automata. This close resemblance will be used in the following where we introduce alternating parity automata, a model of finite automata operating on infinite words that easily captures LT_μ and allows several problems on other automata to be regarded as emptiness problems.

3 Automata on Infinite Words

3.1 Alternating Parity Automata

We introduce a very powerful automaton model which captures many well-known models of automata on infinite words, including (non)deterministic Büchi and co-Büchi automata.

For a set M let $\mathbb{B}^+(M)$ denote that set of all positive Boolean formulas over M, i.e. the least set that contains M and satisfies: if $\{\varphi, \psi\} \subseteq \mathbb{B}^+(M)$ then $\{\varphi \wedge \psi, \varphi \vee \psi\} \subseteq \mathbb{B}^+(M)$.

A *alternating parity automaton* (APA) is a tuple $\mathcal{A} = (Q, \mathrm{AP}, q_0, \delta, \Omega)$ where Q is a finite set of states, AP is a finite subset of $\mathcal{P}$ as used in the previous section, $q_0 \in Q$ is a designated starting state, $\delta : Q \to \mathbb{B}^+(Q \cup \mathrm{AP} \cup \neg\mathrm{AP})$ with $\neg\mathrm{AP} := \{\neg q \mid q \in \mathrm{AP}\}$ is the transition function, and $\Omega : Q \to \mathbb{N}$ assigns priorities to the states.

Here we measure the size of an automaton, $|\mathcal{A}|$, as the number of its states.

A *run* of the APA $\mathcal{A}$ on a word $w \in (2^{\mathcal{P}})^\omega$ is a leveled DAG t whose nodes are labeled with states from Q, that has a single root on level 0, and the succesors of a node on level n are all on level $n + 1$. Furthermore, it obeys the following rules. We write $t(v)$ for the label of node v.

1. We have $t(v_0) = q_0$ for the root v_0.
2. Take any node v on some level n and let $u_1, \ldots, u_k$ be the set of its successors. Then we have $\{t(u_1), \ldots, t(u_k)\} \models \delta(t(v), w(n))$.

Such a run t is *accepting* if on every path through t the greatest priority of states that occur infinitely often is even. The *language* of $\mathcal{A}$ is $L(\mathcal{A}) = \{w \mid \text{there is an accepting run of } \mathcal{A} \text{ on } w\}$.

Example 4. Take the language $L = \{w \in (2^{\{p,q\}})^\omega \mid \text{if } q \text{ holds infinitely often, then } q \wedge p \text{ holds infinitely often in } w\}$. It is accepted by the APA $(\{0, 1, 2\}, \{p, q\}, 0, \delta, \Omega)$ where Ω is the identity function, and the transition function is defined for all three states i as

$$\delta(i) = (p \wedge q \wedge 2) \vee 1 \vee (\neg q \wedge 0)$$

It is also accepted by the APA $(\{0, 1, 2\}, \{p, q\}, 0, \delta, \Omega)$ where $\Omega(0) = \Omega(2) = 0$ and $\Omega(1) = 1$, and

$$\begin{aligned} \delta(0) &= (\neg q \vee p \vee 1) \wedge 0 \\ \delta(1) &= (q \wedge p \wedge 2) \vee 1 \\ \delta(2) &= 2 \end{aligned}$$

3.2 Subclasses of Alternating Parity Automata

Let $\mathcal{A} = (Q, \mathrm{AP}, q_0, \delta, \Omega)$ be an APA. Then $\mathcal{A}$ is an *alternating Büchi automaton* (ABA) if $\Omega : Q \to \{1, 2\}$. It is an *alternating co-Büchi automaton* (AcoBA) if $\Omega : Q \to \{0, 1\}$. I.e. Büchi acceptance is concerned with the infinite occurrence of some states whereas co-Büchi acceptance demands that certain states occur almost everywhere in a path of a run.

The literature contains different definitions of a weak automaton, sometimes constraining the graph structure of the automaton, sometimes weakening the acceptance condition by changing "occurs infinitely often" into "occurs" simply. These notions are equivalent though [14]. Here we consider the former. $\mathcal{A}$ is called *weak* (WAPA) if for all $q, q' \in Q$ s.t. q' occurs in $\delta(q)$ we have $\Omega(q') \leq \Omega(q)$. Hence, the priorities on every path in a run of a weak automaton are monotonically decreasing, and the largest priority that occurs infinitely often is automatically the one that occurs almost everywhere. Consequently, weak alternating automata can easily be defined as ABA or AcoBA.

$\mathcal{A}$ is *nondeterministic* if for all $q \in Q$ we have that $\delta(q)$ is a disjunction of minterms containing exactly one state, i.e. all other conjuncts are atomic propositions or negations thereof.

3.3 Constructions on Alternating Parity Automata

Alternating automata are closed under all Boolean operations.

Proposition 5. *Let $\mathcal{A}$ and $\mathcal{B}$ be two APA. There are APA*

1. *$\overline{\mathcal{A}}$ s.t. $L(\overline{\mathcal{A}}) = (2^{\mathcal{P}})^\omega \setminus L(\mathcal{A})$ and $|\overline{\mathcal{A}}| = |\mathcal{A}|$;*
2. *$\mathcal{A}_1$ s.t. $L(\mathcal{A}_1) = L(\mathcal{A}) \cup L(\mathcal{B})$ and $|\mathcal{A}_1| = \mathcal{O}(|\mathcal{A}| + |\mathcal{B}|)$;*
3. *$\mathcal{A}_2$ s.t. $L(\mathcal{A}_2) = L(\mathcal{A}) \cap L(\mathcal{B})$ and $|\mathcal{A}_2| = \mathcal{O}(|\mathcal{A}| + |\mathcal{B}|)$;*

Proof. (1) Let $\mathcal{A} = (Q, \mathrm{AP}, q_0, \delta, \Omega)$. We define $\overline{\mathcal{A}} = (Q, \mathrm{AP}, q_0, \overline{\delta}, \overline{\Omega})$ where $\overline{\Omega(q)} := \Omega(q) + 1$, and $\overline{\delta}(q) := \overline{\delta(q)}$ where $\overline{p} = \neg p$, $\overline{\neg p} = p$, $\overline{\zeta \vee \eta} = \overline{\zeta} \wedge \overline{\eta}$ and $\overline{\zeta \wedge \eta} = \overline{\zeta} \vee \overline{\eta}$. Clearly, the dual APA is not any bigger than its counterpart. A careful inspection reveals that it recognises the complement language.

(2+3) The APA $\mathcal{A}_1$ and $\mathcal{A}_2$ are obtained by taking the union of $\mathcal{A}$ and $\mathcal{B}$ and adding a new state q_0 with arbitrary priority and transitions obtained as the disjunction, resp. conjunction of the transitions of the two original starting states. □

This makes APA a rich model for various decision problems.

3.4 Decision Problems for Alternating Parity Automata

Important decision problems for automata are the following.

- Non-Emptiness: given an APA $\mathcal{A}$, is $L(\mathcal{A}) \neq \emptyset$?
- Universality: given an APA $\mathcal{A}$, is $L(\mathcal{A}) = (2^{\mathcal{P}})^\omega$?
- Subsumption: given APA $\mathcal{A}$ and $\mathcal{B}$, is $L(\mathcal{A}) \subseteq L(\mathcal{B})$?
- Equivalence: given APA $\mathcal{A}$ and $\mathcal{B}$, is $L(\mathcal{A}) = L(\mathcal{B})$?

Thanks to Prop. 5 these problems are all interreducible in linear time. For instance, equivalence reduces to non-emptiness because $L(\mathcal{A}) \neq L(\mathcal{B})$ iff $L(\mathcal{C}) \neq \emptyset$ where $\mathcal{C}$ recognises $\big(L(\mathcal{A}) \cap L(\overline{\mathcal{B}})\big) \cup \big(L(\overline{\mathcal{A}}) \cap L(\mathcal{B})\big)$. Equally, non-emptiness and universality are interreducible because $L(\mathcal{A}) \neq \emptyset$ iff $L(\overline{\mathcal{A}}) \neq (2^{\mathcal{P}})^{\omega}$.

Note that these reductions need the full power of alternation as well as the full power of the parity acceptance condition unless they are weak. The dual of a Büchi automaton for instance is not a Büchi automaton but a co-Büchi automaton and vice-versa. Weakness is preserved by dualisation though. Also, the dual of a nondeterministic automaton is in general not a nondeterministic automaton anymore. However, regarding it as an alternating automaton enables easy dualisation.

3.5 Alternating Parity Automata and Temporal Logics

Many decision problems for linear-time temporal logics can be phrased as an emptiness problem for (a subclass) of alternating parity automata. The crucial ingredient for this is of course an equivalence-preserving translation from formulas to automata. With Prop. 4 above it suffices to check that LT_μ can be translated into alternating parity automata that way. However, translating logics like LTL and PSL separately can be benefitial because they may not need the full power of the parity acceptance condition.

Proposition 6 ([23,14,10,4]). *For every ...*

- LTL *formula* φ *there is a weak APA* $\mathcal{A}_\varphi$ *s.t.* $L(\mathcal{A}_\varphi) = L(\varphi)$ *and* $|\mathcal{A}_\varphi| = \mathcal{O}(|\varphi|)$*;*
- PSL *formula* φ *there is a weak APA* $\mathcal{A}_\varphi$ *s.t.* $L(\mathcal{A}_\varphi) = L(\varphi)$ *and* $|\mathcal{A}_\varphi| = 2^{\mathcal{O}(|\varphi|)}$*;*
- LT_μ *formula* φ *there is an APA* $\mathcal{A}_\varphi$ *s.t.* $L(\mathcal{A}_\varphi) = L(\varphi)$ *and* $|\mathcal{A}_\varphi| = \mathcal{O}(|\varphi|)$.

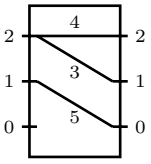
4 The Size-Change Termination Principle for Alternating Parity Automata

4.1 Boxes and Their Composition

We use $<$ to denote the usual total ordering on $\mathbb{N}$ or $\mathbb{Z}$, and introduce a second (non-well-founded) total ordering called *reward ordering*: $i \prec j$ iff $(-1)^i \cdot i < (-1)^j \cdot j$. I.e. we have $\ldots \prec 3 \prec 1 \prec 0 \prec 2 \prec \ldots$

For the remainder of this section we fix an APA $\mathcal{A} = (Q, \mathrm{AP}, q_0, \delta, \Omega)$. Let $P = \{\Omega(q) \mid q \in Q\}$ be the set of all priorities occurring in $\mathcal{A}$. We write $P_\perp$ for $P \cup \{\perp\}$ which will be used to model partial functions into P.

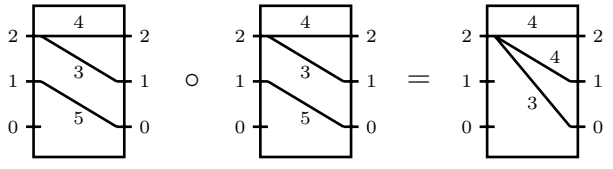
A *box* is an element of type $Q \times Q \to P_\perp$. The name suggest a particular visual representation of such a function. We regard the elements of Q as ports of a circuit, and a box has in- and out-ports which can be connected by an edge labeled with a number. For instance, if $Q = \{0, 1, 2\}$ and $P = \{3, 4, 5\}$ then the following is a box over Q.



We define a *composition* operation $\circ$ on two boxes f, g by

$$(f \circ g)(q, p) = \max_{\prec}\{\max_{<}\{f(q, q'), g(q', p)\} \mid q' \in Q\}$$

Here we use the important convention that $\bot$ is the *maximal* element w.r.t. $<$, but the *minimal* element w.r.t. $\prec$. Then box composition works as one would expect it from the graphical representation.



Box composition is lifted to sets of boxes in the natural way: $F \circ G := \{f \circ g \mid f \in F, g \in G\}$.

With every finite word $v \in (2^{\mathrm{AP}})^\omega$ we associate a set of boxes $[v]$ as follows: $[\epsilon]$ contains only a single box f which is defined by $f(q, q') = \Omega(q)$ if $q' = q$ and $f(q, q') = \bot$ otherwise. Furthermore, for every one-letter word $a \subseteq \mathrm{AP}$ we have that $f \in [a]$ if for all $q, q' \in Q$:

1. $f(q, q') \in \{\Omega(q), \bot\}$, and
2. $\{p \mid f(q, p) \neq \bot\} \cup a \models \delta(q)$

Intuitively, a box belongs to $[a]$ if it connects every in-port to all the out-ports in some model that agrees with a on the atomic propositions and their negations. The labels on the connections simply reflect the priority of the in-port.

Using composition it is easy to define $[v]$ for longer words v: $[av] := [a] \circ [v]$. We write $[\mathcal{A}^*]$ for $\{[v] \mid v \in (2^{\mathrm{AP}})^*$ and $[\mathcal{A}^+]$ for $\{[v] \mid v \in (2^{\mathrm{AP}})^+$. Note that these are finite sets.

A box f is called *idempotent* if $f \circ f = f$. It is called *good* w.r.t. some $Q' \subseteq Q$ if for all $q \in Q'$ we have that $f(q, q)$ is even.

4.2 Characterising the Emptiness Problem for Alternating Parity Automata

Theorem 1. *$L(\mathcal{A}) \neq \emptyset$ iff there are $f \in [\mathcal{A}^*]$ and $g \in [\mathcal{A}^+]$ s.t. g is idempotent and good w.r.t. $\{q \mid f(q_0, q) \neq \bot\}$.*

Proof. "$\Leftarrow$" Suppose such $f, g \in [\mathcal{A}]$ exist. Then there must be words $v_f = a_1 \dots a_n$ and $v_g = b_1 \dots b_m$ with $n \geq 0$, $m \geq 1$ s.t. $f \in [v_f]$ and $g \in [v_g]$. I.e. there must be $f_1 \in [a_1], \dots, f_n \in [a_n], g_1 \in [b_1], \dots, g_m \in [b_m]$ s.t. $f =$

$f_1 \circ \ldots \circ f_n$ and $g = g_1 \circ \ldots \circ g_m$. A run on $v_f(v_g)^\omega$ can easily be extracted from the sequence $f_1, \ldots, f_n, g_1, \ldots, g_m, g_1, \ldots$ by following all connections starting from q_0. Suppose this run was not accepting. Then it would contain a path on which the highest priority seen infinitely often is odd. Note that idempotency of g means that this path is compressed into a connection from some q' to itself in this box. Furthermore, q' must be reachable from q_0 in f. But then $g(q', q')$ must be odd which contradicts the assumption that g is good.

"$\Rightarrow$" Suppose that $L(\mathcal{A}) \neq \emptyset$, i.e. there is a $w \in (2^{\mathrm{AP}})^\omega$ s.t. $w \in L(\mathcal{A})$. Then there is an accepting run t of $\mathcal{A}$ on w. Let $w = a_0 a_1 \ldots$ The run t can easily be transformed into a sequence of boxes $f_0, f_1, \ldots$ by possibly adding nodes to each level s.t. every state is present on every level, and adding corresponding connections. Now consider a colouring of all ordered pairs of levels i, j with $i \leq j$, assigning to this pair the box $f_{i,j} := f_i \circ \ldots \circ f_j$. Since there are only finitely many boxes, Ramsey's Theorem [18] gives us an infinite sequence $j_0 < j_1 < \ldots$ of indices s.t. all pairs of indices from this sequence get assigned to the same box. In particular we have $f_{j_0,j_1} = f_{j_1,j_2} = f_{j_0,j_2} = f_{j_0,j_1} \circ f_{j_1,j_2}$ which shows that it is idempotent. Define the required boxes as $f := f_{0,j_0}$ and $g := f_{j_0,j_1}$. What remains to be seen is that g is good w.r.t. the set of all states that q_0 is connected to in f. As above, suppose it was not. Then the run would have contained an infinite path on which the highest priority occurring infinitely often was odd which would contradict the assumption that it was accepting. □

References

1. Abdulla, P.A., Chen, Y.-F., Clemente, L., Holík, L., Hong, C.-D., Mayr, R., Vojnar, T.: Simulation subsumption in ramsey-based büchi automata universality and inclusion testing. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 132–147. Springer, Heidelberg (2010)
2. Inc. Accellera Organization. Formal semantics of Accellera property specification language (2004), In Appendix B of http://www.eda.org/vfv/docs/PSL-v1.1.pdf
3. Banieqbal, B., Barringer, H.: Temporal logic with fixed points. In: Banieqbal, B., Pnueli, A., Barringer, H. (eds.) Temporal Logic in Specification. LNCS, vol. 398, pp. 62–73. Springer, Heidelberg (1989)
4. Bustan, D., Fisman, D., Havlicek, J.: Automata constructions for PSL. Technical Report MCS05-04, The Weizmann Institute of Science (2005)
5. Dax, C., Hofmann, M., Lange, M.: A proof system for the linear time μ-calculus. In: Arun-Kumar, S., Garg, N. (eds.) FSTTCS 2006. LNCS, vol. 4337, pp. 274–285. Springer, Heidelberg (2006)
6. Dax, C., Klaedtke, F.: Alternation elimination by complementation (Extended abstract). In: Cervesato, I., Veith, H., Voronkov, A. (eds.) LPAR 2008. LNCS (LNAI), vol. 5330, pp. 214–229. Springer, Heidelberg (2008)
7. Emerson, E.A.: Temporal and modal logic. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science. Formal Models and Semantics, vol. B, ch. 16, pp. 996–1072. Elsevier, MIT Press, New York, USA (1990)
8. Fogarty, S., Vardi, M.Y.: Büchi complementation and size-change termination. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 16–30. Springer, Heidelberg (2009)

9. Fogarty, S., Vardi, M.Y.: Efficient büchi universality checking. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 205–220. Springer, Heidelberg (2010)
10. Kaivola, R.: Using Automata to Characterise Fixed Point Temporal Logics. PhD thesis, LFCS, Division of Informatics, The University of Edinburgh, Tech. Rep. ECS-LFCS-97-356 (1997)
11. Kupferman, O., Piterman, N., Vardi, M.Y.: Extended temporal logic revisited. In: Larsen, K.G., Nielsen, M. (eds.) CONCUR 2001. LNCS, vol. 2154, pp. 519–535. Springer, Heidelberg (2001)
12. Lange, M.: Linear time logics around PSL: Complexity, expressiveness, and a little bit of succinctness. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 90–104. Springer, Heidelberg (2007)
13. Lee, C.S., Jones, N.D., Ben-Amram, A.M.: The size-change principle for program termination. ACM SIGPLAN Notices 36(3), 81–92 (2001)
14. Löding, C., Thomas, W.: Alternating automata and logics over infinite words. In: Watanabe, O., Hagiya, M., Ito, T., van Leeuwen, J., Mosses, P.D. (eds.) TCS 2000. LNCS, vol. 1872, pp. 521–535. Springer, Heidelberg (2000)
15. Miyano, S., Hayashi, T.: Alternating finite automata on omega-words. TCS 32(3), 321–330 (1984)
16. Pnueli, A.: The temporal logic of programs. In: Proc. 18th Symp. on Foundations of Computer Science, FOCS 1977, pp. 46–57. IEEE, Providence (1977)
17. Ramsey, F.P.: On a problem of formal logic. Proc. London Mathematical Society, Series 2 30(4), 338–384 (1928)
18. Ramsey, F.P.: On a problem in formal logic. Proc. London Math. Soc. 30(3), 264–286 (1930)
19. Sistla, A.P., Clarke, E.M.: The complexity of propositional linear temporal logics. Journal of the Association for Computing Machinery 32(3), 733–749 (1985)
20. Stirling, C.: Comparing linear and branching time temporal logics. In: Banieqbal, B., Pnueli, A., Barringer, H. (eds.) Temporal Logic in Specification. LNCS, vol. 398, pp. 1–20. Springer, Heidelberg (1989)
21. Vardi.: Linear vs. branching time: A complexity-theoretic perspective. In: LICS: IEEE Symposium on Logic in Computer Science (1998)
22. Vardi, M.Y.: A temporal fixpoint calculus. In: ACM (ed.) Proc. Conf. on Principles of Programming Languages, POPL 1988, pp. 250–259. ACM Press, New York (1988)
23. Vardi, M.Y.: Alternating automata and program verification. In: van Leeuwen, J. (ed.) Computer Science Today. LNCS, vol. 1000, pp. 471–485. Springer, Heidelberg (1995)
24. Vardi, M.Y.: An Automata-Theoretic Approach to Linear Temporal Logic. In: Moller, F., Birtwistle, G. (eds.) Logics for Concurrency. LNCS, vol. 1043, pp. 238–266. Springer, Heidelberg (1996)
25. Vardi, M.Y.: Branching vs. Linear time: Final showdown. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 1–22. Springer, Heidelberg (2001)
26. Vardi, M.Y., Wolper, P.: Reasoning about infinite computations. Information and Computation 115(1), 1–37 (1994)
27. Wolper, P.: Temporal logic can be more expressive. Information and Control 56, 72–99 (1983)

Combining Theories: The Ackerman and Guarded Fragments*

Carlos Areces[1,2] and Pascal Fontaine[1,3]

[1] INRIA Nancy-Grand Est, Nancy, France
[2] FaMAF, Universidad Nacional de Córdoba, Córdoba, Argentina
[3] Université de Nancy, Loria, Nancy, France
Carlos.Areces@gmail.com,
Pascal.Fontaine@loria.fr

Abstract. Combination of decision procedures is at the heart of Satisfiability Modulo Theories (SMT) solvers. It provides ways to compose decision procedures for expressive languages which mix symbols from various decidable theories. Typical combinations include (linear) arithmetic, uninterpreted symbols, arrays operators, etc. In [7] we showed that any first-order theory from the Bernays-Schönfinkel-Ramsey fragment, the two variable fragment, or the monadic fragment can be combined with virtually any other decidable theory. Here, we complete the picture by considering the Ackermann fragment, and several guarded fragments. All theories in these fragments can be combined with other decidable (combinations of) theories, with only minor restrictions. In particular, it is not required for these other theories to be stably-infinite.

1 Introduction

Devising satisfiability decision procedures for the combination of logical theories has been a very active research subject during the last fifteen years. It is the theoretical background on which Satisfiability Modulo Theories (SMT) solvers are built. For instance, the set of literals

$$L = \{a \leq b, b \leq a + f(a), P(h(a) - h(b)), \neg P(0), f(a) = 0\}$$

can be shown to be unsatisfiable by an SMT solver, implementing the Nelson-Oppen framework [16] combining a decision procedure for the theory of uninterpreted symbols and a decision procedure for linear arithmetic. SMT solvers (see [2] for a thorough presentation of the techniques behind SMT solvers) are now widely used, notably for model-checking and formal verification.

Initial combination results (e.g., [16,17]) imposed strong conditions to ensure decidability of the satisfiability problem for the combined theories, such as requiring the theories to be *stably infinite*, i.e., requiring any satisfiable set of literals within the theories to have an infinite model. Many theories, and specially, many theories interesting for formal verification of hardware and software,

* This work is partly supported by the ANR project DECERT.

C. Tinelli and V. Sofronie-Stokkermans (Eds.): FroCoS 2011, LNAI 6989, pp. 40–54, 2011.

are expressive enough to restrict the size of a model to be finite and, hence, are not stably infinite. In other words, stable infiniteness is a sufficient condition for theory combination, but it is too restrictive.

In recent years, much of the research in the area has focused on finding more relaxed conditions for combination that would ensure decidability of the combined theories. Tinelli and Zarba introduced in [19] the notion of *shiny theories* (see Definition 4) and proved that the disjoint combination of one shiny theory with an arbitrary (that is, not necessarily stably infinite) decidable theory is decidable. In [7] we considered the Bernays-Schönfinkel-Ramsey fragment, the two variables fragment, and the monadic fragment. These fragments include non stably infinite theories. We introduced the notion of *gentleness* (see Definition 5) and proved that the disjoint combination of one gentle theory with an arbitrary decidable theory (modulo a minor restriction[1]) is also decidable. All theories in the considered fragments are gentle.

In this article we first investigate the combination of *guarded fragments* of first-order logic with other decidable fragments. Guarded fragments, originally introduced in [1] as first-order counterparts of modal languages, are very expressive. In contrast to other well-known decidable classes, the guarded fragments impose no restriction on the number of variables, alternations of quantifiers, or symbol arity. Instead, quantification is restricted to occur only in guarded form. Relational properties such as symmetry of a relation (written as $\forall xy\,.\,R(x,y) \rightarrow R(y,x)$) can readily be expressed with these fragments, as well as various graph properties such as $\forall xy\,.\,R(x,y) \rightarrow \exists z\,.\,R(y,z)$ stating that every node with an incoming edge has an outgoing edge, or constraints such as $\forall yz\,.\,R(y,y,z) \rightarrow \bot$ which forbids certain kinds of tuples to appear in a relation.

In this article we will show that the guarded fragment [1], the loosely guarded fragment [20] and the packed guarded fragment [15] are shiny, and hence, they can be combined in a decidable way, with an arbitrary decidable theory. This can be seen as further explanation of the good computational behavior of many modal logics [21,9].

To complete the picture of combination of theories from decidable first-order fragments, we also consider the well-known Ackermann fragment, i.e. formulas of the form $\exists^*\forall x\exists^*\varphi$, where φ is a function- and quantifier-free first-order formula. In this paper we will show that this fragment is gentle and, thus, easily combinable with arbitrary theories (with a minor restriction).

After introducing notations and definitions in Section 2, we discuss combination of decision procedures in the disjoint case in Section 3. Section 4 introduces the guarded fragments we will consider. The status of constants and equality in these fragments is sometimes unclear in the literature; since these are of foremost importance in our context, they will be handled with special care. Section 5 considers the Ackermann fragment. The proofs we present are straightforward but, to our knowledge, this is the first time that these fragments have been explored in the framework of combined theories.

[1] This theory should fall in one of the three cases of Theorem 3. These cases are such that unsuitable theories would be very particular.

2 Notations and Basic Definitions

A first-order language is a tuple $\mathcal{L} = \langle \mathcal{V}, \mathcal{F}, \mathcal{P} \rangle$ such that $\mathcal{V}$ is an enumerable set of variables, while $\mathcal{F}$ and $\mathcal{P}$ are sets of function and predicate symbols. Every function and predicate symbol has an arity. Nullary predicates symbols are called proposition symbols, and nullary function symbols are called constant symbols. A first-order language is called relational if it only contains function symbols of arity zero. A relational formula is a formula in a relational language.

Terms and formulas over the language $\mathcal{L}$ are defined in the usual way. An atomic formula is either an equality statement $(t = t')$ where t and t' are terms, or a predicate symbol applied to the right number of terms. Formulas are built from atomic formulas, Boolean connectives ($\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$), and quantifiers ($\forall$, $\exists$). A literal is an atomic formula or the negation of an atomic formula. The set of free variables $\mathrm{Free}(\varphi)$ in a formula φ is defined as usual. A formula with no free variables is closed. A theory is a set of closed formulas. Two theories are disjoint if no predicate or function symbol appears in both theories; the theories can however share constants.

An interpretation $\mathcal{I}$ for a first-order language $\mathcal{L}$ provides a non empty domain D, a total function $\mathcal{I}[f] : D^r \rightarrow D$ of appropriate arity for every function symbol f, a predicate $\mathcal{I}[p] : D^r \rightarrow \{\top, \bot\}$ of appropriate arity for every predicate symbol p, and an element $\mathcal{I}[x] \in D$ for every variable x. By extension, an interpretation defines a value in D for every term, and a truth value for every formula. The cardinality of an interpretation is the cardinality of its domain. The notation $\mathcal{I}_{x_1/d_1,\ldots,x_n/d_n}$ for $x_1, \ldots, x_n$ different variables stands for the interpretation that agrees with $\mathcal{I}$, except that it associates $d_i \in D$ to the variable x_i, $1 \leq i \leq n$. Given an interpretation $\mathcal{I}$ on domain D, an extension $\mathcal{I}'$ of $\mathcal{I}$ is an interpretation on a domain including D such that $\mathcal{I}'$ restricted to the domain D is exactly $\mathcal{I}$.

A model of a formula (or a theory) is an interpretation in which the formula (resp., every formula in the theory) evaluates to true. A formula or theory is satisfiable if it has a model, and it is unsatisfiable otherwise. A formula G is $\mathcal{T}$-satisfiable if it is satisfiable in the theory $\mathcal{T}$, that is, if $\mathcal{T} \cup \{G\}$ is satisfiable. A $\mathcal{T}$-model of G is a model of $\mathcal{T} \cup \{G\}$. A formula G is $\mathcal{T}$-unsatisfiable if it has no $\mathcal{T}$-models. A decidable theory $\mathcal{T}$ is a theory such that the $\mathcal{T}$-satisfiability problem for sets of literals in the language of $\mathcal{T}$ is decidable.

The bold notation $\boldsymbol{x}$ denotes a tuple, and stands for a sequence of variables or constants (or both) depending on the context. For instance, in $\forall \boldsymbol{x}\, \varphi$, formula φ is quantified universally over all variables in $\boldsymbol{x}$. Expressions such as $p(\boldsymbol{x})$, $p(\boldsymbol{y}, c)$, $p(\boldsymbol{z}, \boldsymbol{d})$ and $\mathcal{I}_{\boldsymbol{x}/\boldsymbol{d}}$, where p is a predicate and $\mathcal{I}$ an interpretation, may be used, with straightforward meaning. When used with set operators, tuples behave like the set of the elements in the tuple, whereas $|\boldsymbol{x}|$ gives the length of the tuple.

3 Combination of Theories

To study the satisfiability of a set of literals like

$$L = \{a \leq b, b \leq a + f(a), P(h(a) - h(b)), \neg P(0), f(a) = 0\}$$

that mixes symbols from the integer linear arithmetic theory $\mathcal{T}_1$ and the theory of uninterpreted symbols $\mathcal{T}_2$, one uses a combination framework to design a decision procedure for the joint language from the simple component decision procedures for one theory only. To divide the above satisfiability problem into problems for the component decision procedures, a *separation* is first built by introducing fresh uninterpreted constants[2], to produce an equisatisfiable problem:

$$L_1 = \{a \leq b,\, b \leq a + v_1,\, v_1 = 0,\, v_2 = v_3 - v_4,\, v_5 = 0\}$$
$$L_2 = \{P(v_2),\, \neg P(v_5),\, v_1 = f(a),\, v_3 = h(a),\, v_4 = h(b)\}.$$

The set L_1 only contains arithmetic symbols and uninterpreted constants. The symbols in L_2 are all uninterpreted. The decision procedure for linear arithmetic and the one for uninterpreted symbols can thus handle the sets L_1 and L_2 respectively. However, although L is unsatisfiable in $\mathcal{T}_1 \cup \mathcal{T}_2$, L_1 is $\mathcal{T}_1$-satisfiable, and L_2 is $\mathcal{T}_2$-satisfiable; it is not sufficient for the decision procedures for $\mathcal{T}_1$ and $\mathcal{T}_2$ to only examine the satisfiability of their part of the separation. Indeed, the decision procedures also have to "agree" on the symbols that are shared, namely the uninterpreted constants in the set $S = \{a, b, v_1, v_2, v_3, v_4, v_5\}$. In order to make sure that both decision procedures will interpret those shared symbols coherently, the notion of arrangement is useful:

Definition 1. *An arrangement $\mathcal{A}$ for a set of constant symbols S is a maximal satisfiable set of equalities and inequalities $a = b$ or $a \neq b$, with $a, b \in S$.*

That is, an arrangement $\mathcal{A}$ for S cannot be extended with any equality or inequality over S and remain consistent.

The following theorem (other formulations can be found in [18,19,8]) then states the completeness of the combination of decision procedures:

Theorem 1. *Assume $\mathcal{T}_1$ and $\mathcal{T}_2$ are theories over the disjoint languages $\mathcal{L}_1$ and $\mathcal{L}_2$, and L_i ($i = 1, 2$) is a set of literals in $\mathcal{L}_i$ augmented by a finite set of fresh constant symbols S. Then $L_1 \cup L_2$ is $\mathcal{T}_1 \cup \mathcal{T}_2$-satisfiable if and only if there exists an arrangement $\mathcal{A}$ of S, a cardinality k, and two models $\mathcal{M}_1$ and $\mathcal{M}_2$ of cardinality k, such that $\mathcal{M}_1$ is a $\mathcal{T}_1$-model of $\mathcal{A} \cup L_1$ and $\mathcal{M}_2$ is a $\mathcal{T}_2$-model of $\mathcal{A} \cup L_2$.*

Intuitively, if a set of literals is satisfiable in the combination of theories, a model of this set defines in a straightforward way an arrangement and two models with the same cardinality for the two parts of the separation. The converse is also true: from models of the two parts of the separation (augmented with the arrangement), it is possible to build a unique model for both parts, since both models agree on the cardinality, and on the interpretation of the shared constants in S (thanks to the arrangement). The cardinality condition is essential to be able to map elements in both domains together into a unique domain.

[2] Traditionally, combination schemes use variables for this role. Since variables will be used in quantifiers in the following sections, for consistency and clarity we will rather use uninterpreted constants here.

Relying on the above theorem, an algorithm implementing a satisfiability decision procedure for the combination of two disjoint decidable theories $\mathcal{T}_1$ and $\mathcal{T}_2$ could be as follows:

1. Build a separation (L_1, L_2) for the set of literals L which mix symbols from $\mathcal{T}_1$ and $\mathcal{T}_2$. L_1 contains symbols from $\mathcal{T}_1$ only and symbols from a finite set of fresh constant symbols S, and likewise for L_2;
2. Guess an arrangement $\mathcal{A}$ for the set of constants shared between L_1 and L_2;
3. If $\mathcal{A} \cup L_i$ is $\mathcal{T}_i$-satisfiable for $i = 1, 2$, then, if there exists a (finite or infinite) cardinality k such that $\mathcal{A} \cup L_i$ has a model of cardinality k for $i = 1, 2$, then L is $\mathcal{T}_1 \cup \mathcal{T}_2$-satisfiable. Otherwise, $\mathcal{A} \cup L$ is $\mathcal{T}_1 \cup \mathcal{T}_2$-unsatisfiable.

If we want to ensure that the above algorithm is indeed a decision procedure, two issues needs to be solved. First, as we presented it above the algorithm is non-deterministic but this is not fundamental. Since the number of arrangements for a fixed finite set of constants is finite (although large), the non-deterministic choice can be turned into a loop over this set. The second issue is, however, essential. It involves being able to compare the cardinalities of the models for both parts of the arrangement. To handle this problem, combination of decision procedures and SMT solvers usually consider only stably infinite theories:

Definition 2. *A theory $\mathcal{T}$ is said to be stably infinite when every $\mathcal{T}$-satisfiable set of literals has a model with cardinality $\aleph_0$.*

By definition, when dealing with stably infinite theories, if both parts of the separation are satisfiable in their corresponding theory, then both have an infinite model of cardinality $\aleph_0$.

Consider again the above example. As both the theory for uninterpreted symbols and the theory of integer linear arithmetic are stably infinite, the set of literals L in our example is $\mathcal{T}_1 \cup \mathcal{T}_2$-satisfiable if and only if there exists an arrangement $\mathcal{A}$ of the seven variables in S such that $\mathcal{A} \cup L_i$ is $\mathcal{T}_i$-satisfiable for $i = 1$ and $i = 2$. No such arrangements exist. Indeed, consider an arrangement $\mathcal{A}$ such that $\mathcal{A} \cup L_1$ is $\mathcal{T}_1$-satisfiable and $\mathcal{A} \cup L_2$ is $\mathcal{T}_2$-satisfiable. Such an arrangement contains $a = b$, otherwise $\mathcal{A} \cup L_1$ would not be $\mathcal{T}_1$-satisfiable. It also contains $v_3 = v_4$ since $\mathcal{A} \cup L_2$ is $\mathcal{T}_2$-satisfiable, and as a consequence, $v_2 = v_5$ should also be in $\mathcal{A}$. But if $\mathcal{A}$ contains $v_2 = v_5$, $\mathcal{A} \cup L_2$ is not $\mathcal{T}_2$-satisfiable.

Considering stably infinite theories only is one way to fulfill the cardinality requirement for disjoint combination. It is, however, very restrictive. While some very useful theories are stably infinite, many are not. For instance, there exist theories that only have finite models. Many first-order decidable classes allow to write formulas that constrain the cardinality of the models. Consider, for example, the Ackermann theory $\varphi = \{\forall x \,.\, p(c) \rightarrow (x = a \vee x = b)\}$ that requires the cardinality of the model to be at most two whenever $p(c)$ is true. While $\varphi \cup \{\neg p(c)\}$ does have infinite models, $\varphi \cup \{p(c)\}$ only has finite models.

Of course, there are other ways to ensure that the cardinality requirement is fulfilled. They allow to build decision procedures for union of theories that are

not all stably infinite. To examine the cardinality requirements in Theorem 1, the notion of spectrum[3] is convenient.

Definition 3. *The* spectrum *of a theory $\mathcal{T}$ is the set of cardinalities k such that $\mathcal{T}$ is satisfiable in a model of cardinality k.*

Theorem 1 now becomes: $L_1 \cup L_2$ is $\mathcal{T}_1 \cup \mathcal{T}_2$-satisfiable if and only if there exists an arrangement $\mathcal{A}$ of S, such that the spectra of $\mathcal{T}_1 \cup \mathcal{A} \cup L_1$ and $\mathcal{T}_2 \cup \mathcal{A} \cup L_2$ have a non-empty intersection. The intersection of spectra is the crucial difficulty for combination frameworks. Fortunately, the spectrum for many theories (as we will see for guarded fragments and the Ackermann class) is such that the computation of the intersection with another spectrum is easy.

Some theories (e.g., the empty theory, the theory of partial orders, the theory of total orders) have spectral properties that allow combination with any other decidable disjoint theory; these are called *shiny* theories [19].

Definition 4. *A decidable theory $\mathcal{T}$ is* shiny *if, for every $\mathcal{T}$-satisfiable set of literals L, there is a finite computable number k such that the spectrum of $\mathcal{T} \cup L$ is the set of cardinalities greater than or equal to k.*

In the following sections, we show that the guarded, the loosely guarded, and the packed guarded fragments are all shiny. They can thus be combined with any disjoint theory:

Theorem 2. *Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be two disjoint decidable theories sharing only constants. If $\mathcal{T}_1$ is shiny then $\mathcal{T}_1 \cup \mathcal{T}_2$ is decidable.*

In [7], we show that theories in the Bernays-Schönfinkel-Ramsey class, the two variables fragment, and the monadic fragment have interesting spectral properties, though weaker than shininess. Every theory $\mathcal{T}$ in these classes is gentle:

Definition 5. *A theory $\mathcal{T}$ is* gentle *if, for every $\mathcal{T}$-satisfiable set of literals L, the spectrum of $\mathcal{T} \cup L$ can be computed and is either*

- *a finite set of finite cardinalities*
- *the union of a finite set of finite cardinalities and all the (finite and infinite) cardinalities greater than a computable finite cardinality; it is thus co-finite.*

The definition of shininess and gentleness are quite similar; considering only sufficiently large cardinalities, both notions express the same property. Notice that a shiny theory is also gentle. Furthermore, the union of disjoint gentle theories is also a gentle theory [7]. Some widely used theories are not gentle, but in practical cases they can be combined with gentle theories [7]:

Theorem 3. *Given a gentle theory $\mathcal{T}$ and another disjoint theory $\mathcal{T}'$, the $\mathcal{T} \cup \mathcal{T}'$-satisfiability problem for sets of literals written in the union of their language is decidable if one of the following cases holds:*

[3] The spectrum of a theory is usually defined as the set of the *finite* cardinalities of its models. We here slightly extend the definition for convenience.

- $\mathcal{T}'$ *is gentle;*
- $\mathcal{T}'$ *is a decidable finitely axiomatized first-order theory;*
- $\mathcal{T}'$ *is a decidable theory that only admits a fixed finite (possibly empty) known set of finite cardinalities for its models, and possibly infinite models.*

In the next sections we will prove that guarded fragments are shiny, and that the Ackermann theories are gentle.

4 The Guarded Fragments

The guarded fragment (GF) was originally introduced in [1] as a suitable counterpart and generalization of modal logics. To make this article self contained, let us start with a brief recapitulation of modal logics (see [3,4] for further details). Consider the language defined as

$$\mathcal{BML} := p_i \mid \neg\varphi \mid \varphi \vee \psi \mid \Diamond\varphi,$$

where p_i is a propositional symbol and $\varphi, \psi \in \mathcal{BML}$. Syntactically, the language $\mathcal{BML}$ (the basic modal language) is a slight extension of propositional logic (we have only added the unary operator $\Diamond$). Semantically, on the other hand, the change is radical. We interpret formulas of $\mathcal{BML}$ on first-order relational models $M = \langle D, \mathcal{I}\rangle$ over a signature with only one binary relational symbol R and uncountably many propositional symbols $\{p_1, p_2, \ldots\}$. Given such a model M and an element a in the domain, semantics is defined as follows:

$$\begin{aligned}
M[p](a) = \top &\text{ iff } \mathcal{I}[p](a)\\
M[\neg\varphi](a) = \top &\text{ iff } M[\varphi](a) = \bot\\
M[\varphi \vee \psi](a) = \top &\text{ iff } M[\varphi](a) = \top \text{ or } M[\psi](a) = \top\\
M[\Diamond\varphi](a) = \top &\text{ iff for some } b \in D,\ \mathcal{I}[R](a,b) \text{ and } M[\varphi](b) = \top.
\end{aligned}$$

These semantic conditions should tip us off on the close connections between modal and first-order languages. Indeed, it is simple to define an equivalence preserving translation from the former to the latter. Define recursively the translation Tr_x for x a first-order variable as:

$$\begin{aligned}
\mathsf{Tr}_x(p) &= P(x)\\
\mathsf{Tr}_x(\neg\varphi) &= \neg\mathsf{Tr}_x(\varphi)\\
\mathsf{Tr}_x(\varphi \vee \psi) &= \mathsf{Tr}_x(\varphi) \vee \mathsf{Tr}_x(\psi)\\
\mathsf{Tr}_x(\Diamond\varphi) &= \exists y.R(x,y) \wedge \mathsf{Tr}_y(\varphi),
\end{aligned}$$

where y is a new variable, not yet used in the translation. A simple induction shows that for any formula $\varphi \in \mathcal{BML}$, any model M (in the proper signature) and any element a in the domain of M, $M[\varphi](a) = \top$ iff $M[\mathsf{Tr}_x(\varphi)](a) = \top$. In other words, $\mathcal{BML}$ can be seen as nothing else than a fragment of first-order logic in disguise. But $\mathcal{BML}$ is only *one* among many modal logics. Other modal operators such as the inverse modality, the universal modality, the difference modality, etc. can be defined (see [4] for details). Most of them can be translated

into first-order logic preserving equivalence. A natural question is then, whether it is possible to define a fragment of first-order logic that can be the range of these translations, and that will preserve the common modal aspects of all these logical languages. The answer to this question was the guarded fragment GF.

Definition 6. *A formula* γ guards *another formula* φ *if every free variable of* φ *also occurs free in* γ *(i.e.,* $\text{Free}(\varphi) \subseteq \text{Free}(\gamma)$*).*

Definition 7. *A formula in the* guarded fragment GF *of first-order logic is a relational formula such that all quantified sub-formulas are of the form* $\forall \boldsymbol{x} \,.\, \gamma \rightarrow \psi$ *or* $\exists \boldsymbol{x} \,.\, \gamma \wedge \psi$ *where*

- γ *is an atom, but not an equality,*
- ψ *is guarded by* γ*,*
- $\boldsymbol{x}$ *is a tuple of variables in* $\text{Free}(\gamma)$*,*

The atom γ *is called the* guard*.*

Formulas in the fragment might contain an arbitrary number of variables (i.e., GF is not contained in any finite variable fragment of first-order logic). Similarly, formulas in GF might contain an arbitrary number of quantifier alternations, and hence they cannot be defined in terms of prenex normal form prefixes. Also, the arity of relational symbols is not bounded. Moreover, many natural properties expressible in first-order logic fall in GF. Some examples, besides those we mentioned in Section 1, are:

$\forall x \,.\, R(x, x)$	reflexivity
$\forall x \,.\, \neg R(x, x)$	irreflexivity
$\exists v_1 \,.\, (R(a, v_1) \wedge \exists v_2 \,.\, (R(v_1, v_2) \wedge R(v_2, b)))$	there is a path of length 3 between a and b

On the other hand, some simple formulas, such as transitivity $\forall xyz \,.\, (R(x, y) \wedge R(y, z)) \rightarrow R(x, z)$, are neither in GF nor equivalent to any formula of GF (i.e., transitivity is not expressible in GF).

Guarded fragments have been defined and redefined repeatedly, looking for the largest fragment of first-order logic with a nice 'modal' behavior. The original definition of [1] contained the restriction on equality atoms not appearing in guards we introduced above. This restriction was later removed (even though the exact status of equality in the different definitions of guarded fragments is sometimes unclear), but it is crucial for the results we will present.

Suppose we eliminate this restriction. Then equality atoms could occur as a guard in one of the following shapes (let's consider only universal quantification):

1. $\forall x \,.\, x = x \rightarrow \psi(x)$
2. $\forall x \,.\, x = y \rightarrow \psi(x, y)$
3. $\forall xy \,.\, x = y \rightarrow \psi(x, y)$
4. $\forall x \,.\, x = c \rightarrow \psi(x)$

Cases 2 and 4 can be rewritten as $\psi(y, y)$ and $\psi(c)$, respectively, eliminating the quantifier and resulting in a formula in GF. Cases 1 and 3 rewrite to $\forall x \,.\, \psi(x)$ and $\forall x \,.\, \psi(x, x)$, respectively. The resulting formulas in the scope of the quantifier contain at most one free variable, but this variable is not guarded. Without the restriction on the use of equality in guards GF would include formulas such as $\forall x \,.\, x = a_1 \vee \ldots x = a_n$ that restricts the domain to a finite cardinality smaller or equal to n. These improper guarded formulas would invalidate the good properties necessary for combining GF theories (see Corollary 1 below).

Many good properties of GF are shown in [1]. In particular, the authors prove that its satisfiability problem is decidable (it is actually 2ExpTime-complete, and only ExpTime-complete if the number of variables is bounded by any finite number k, see [12]), and that the fragment has the finite-model property (i.e., every satisfiable formula is satisfied in a finite model).

Different variations of GF were introduced, gradually relaxing the conditions imposed on the guard to obtain larger fragments. We present the loosely guarded fragment introduced in [20], and the packed guarded fragment introduced in [15].

Definition 8. *A formula in the* loosely guarded fragment LGF *of first-order logic is a relational formula such that all quantified sub-formulas are of the form* $\forall \boldsymbol{x} \,.\, \gamma \to \psi$ *or* $\exists \boldsymbol{x} \,.\, \gamma \wedge \psi$ *where*

- $\gamma = \alpha_1 \wedge \cdots \wedge \alpha_m$ *is an equality-free conjunction of atoms,*
- ψ *is guarded by* γ*,*
- *for every variable* y *in* $\boldsymbol{x}$ *and every variable* $z \in \mathrm{Free}(\gamma)$ *with* $y \neq z$*, there is at least one atom* α_j *that contains both* y *and* z

The conjunction of atoms γ *is called the* guard.

Notice that GF is a proper subset of LGF. The loosely guarded fragment is decidable [20] and has the finite model property [14]. Its satisfiability problem is 2ExpTime-complete [12].

Definition 9. *A formula in the* packed guarded fragment PGF *of first-order logic is a relational formula such that all quantified sub-formulas are of the form* $\forall \boldsymbol{x} \,.\, \gamma \to \psi$ *or* $\exists \boldsymbol{x} \,.\, \gamma \wedge \psi$ *where*

- $\gamma = \alpha_1 \wedge \cdots \wedge \alpha_m$ *is an equality-free conjunction of atoms and existentially-quantified atomic formulas,*
- ψ *is guarded by* γ*,*
- *for every variables* $y, z \in \mathrm{Free}(\gamma)$ *there is at least one conjunct* α_j *such that* $\{y, z\} \subseteq \mathrm{Free}(\alpha_j)$

The conjunction γ *is called the* guard.

Although LGF is not a subset of PGF, PGF is (strictly) more expressive than LGF: any LGF formula can be rewritten to a logically equivalent PGF formula (see [10]). The packed guarded fragment is also known as the clique-guarded fragment. Both definitions are equivalent [10]. The packed guarded fragment is

decidable and has the finite model property [14]. The satisfiability problem for PGF is 2EXPTIME-complete [13].

The status of constants in guarded fragments has sometimes been vague. Constants are crucial for our goal, as they will be used to link the combined theories. Notice that in our definitions, all guarded fragments allow constants. The following theorem, adapted from [12], shows that constants can always be added to guarded fragments without interfering with decidability, the finite model property or complexity.

Theorem 4. *Adding constants to the languages for* GF, LGF *and* PGF, *preserves decidability, the finite model property, and complexity.*

Proof. Assume φ is a formula in GF, LGF or PGF with constants from a finite set C. Let $\boldsymbol{c}$ be a sequence containing all constants in C. Let G be the set of all predicates occurring in guards (remember that guards are equality free, so G does not include equality). For every n-ary predicate $p \in G$, let p' be a fresh $(n+|\boldsymbol{c}|)$-ary predicate. The formula φ' is built from φ by replacing every occurrence $p(\boldsymbol{x})$, for every $p \in G$ and every sequence of variables and constants $\boldsymbol{x}$ by $p'(\boldsymbol{x}, \boldsymbol{c})$. Let Z be a fresh $|\boldsymbol{c}|$-ary predicate. The formula $\psi = \exists \boldsymbol{c}\,(Z(\boldsymbol{c}) \wedge \varphi')$ — where the constants $\boldsymbol{c}$ in φ are variables in ψ — is equisatisfiable to φ. From a model of φ it is possible to build a model on the same domain for ψ, and conversely, thus the finite model property (and consequently, decidability) is preserved. ψ is constant-free, and it is properly guarded (in the same fragment GF, LGF or PGF than φ). Replacing constants by variables may involve a polynomial growth of the formula. This does not affect the 2EXPTIME-complete complexity. □

4.1 The Spectra of Guarded Fragments

The following theorem states that an interpretation of a formula in the guarded fragments GF, LGF or PGF, can always be extended by new elements without changing the truth value of the considered formula. Intuitively, it suffices for those new elements to be "disconnected" from the other elements, that is, those new elements make every guard false. This, together with the finite model property, will directly imply that these fragments are shiny.

Theorem 5. *Given any interpretation M on domain D for a formula φ in* GF, LGF *or* PGF, *then for every $D' \supset D$ there is an extension M' of M on domain D' such that $M'[\varphi] = M[\varphi]$.*

Proof. Given an interpretation M on domain D for a formula φ in GF, LGF or PGF, the interpretation M' on D' is defined as follows:

- for every constant a, $M'[a] = M[a]$;
- for every variable $x \in \text{Free}(\varphi)$, $M'[x] = M[x]$;
- for every n-ary predicate p, and for $a_i \in D'$ $(1 \leq i \leq n)$
  - $M'[p](a_1, \ldots a_n) = M[p](a_1, \ldots a_n)$ if $a_i \in D$ for all i $(1 \leq i \leq n)$;
  - $M'[p](a_1, \ldots a_n) = \bot$ otherwise.

To be able to handle PGF as the two other fragments in the following, first consider an existentially-quantified atomic formula $\gamma = \exists \boldsymbol{x} \,.\, p(\boldsymbol{y})$. Notice that (1) for any interpretation M'' defined as M' is above, but assigning at least one free variable of γ to an element in $D' \setminus D$, $M''[\gamma] = \bot$ (2) for any interpretation M' defined as above, $M'[\gamma] = M[\gamma]$. The first point is direct. To prove the second, notice that if $M[\gamma] = \top$, then $M_{\boldsymbol{x}/\boldsymbol{d}}[p(\boldsymbol{y})] = \top$ for some tuple $\boldsymbol{d}$ of elements in D. Then $M'_{\boldsymbol{x}/\boldsymbol{d}}[p(\boldsymbol{y})]$ is also true and as a consequence, $M'[\gamma] = \top$. If $M[\gamma] = \bot$, notice that, for any tuple $\boldsymbol{d}$ of elements in D', $M'_{\boldsymbol{x}/\boldsymbol{d}}[p(\boldsymbol{y})] = M_{\boldsymbol{x}/\boldsymbol{d}}[p(\boldsymbol{y})] = \bot$ if all arguments of p are assigned to elements in D, and $M'_{\boldsymbol{x}/\boldsymbol{d}}[p(\boldsymbol{y})] = \bot$ if one argument of p is in $D' \setminus D$. As a consequence $M'[\gamma] = \bot$.

Theorem 5 is proved by showing by structural induction that $M'[\varphi] = M[\varphi]$, for M' defined from M as above. It is trivial if φ is atomic, a negation, or a Boolean combination of several formulas. The only remaining cases are the quantified constructions.

Let $\varphi = \forall x_1 \ldots x_n \,.\, \gamma \to \psi$ (where γ is the guard) belong to GF, LGF or PGF. For simplicity and without loss of generality, assume that $x_i \in \mathrm{Free}(\gamma \to \psi)$ for every $i \in \{1, \ldots n\}$. Consider an interpretation M on domain D for φ, $D' \supset D$, and M' as defined above. For $d_1, \ldots d_n \in D'$ one of the two cases hold:

- if $d_i \in D' \setminus D$ for some $i \in \{1, \ldots n\}$, then $M'_{x_1/d_1,\ldots x_n/d_n}[\gamma] = \bot$, and hence $M'_{x_1/d_1,\ldots x_n/d_n}[\gamma \to \psi] = \top$. Indeed, since γ is a guard, x_i appears free in γ. Since the guard is either (GF) an atom, (LGF) a conjunction of atoms, (PGF) or a conjunction of atoms and existentially quantified atoms, the atom having x_i as an argument is interpreted as false, and so is the whole guard.
- if $d_i \in D$ for all $i \in \{1, \ldots n\}$, then $M'_{x_1/d_1,\ldots x_n/d_n}$ and $M_{x_1/d_1,\ldots x_n/d_n}$ agree on $(\gamma \to \psi)$, i.e., $M'_{x_1/d_1,\ldots x_n/d_n}[\gamma \to \psi] = M_{x_1/d_1,\ldots x_n/d_n}[\gamma \to \psi]$. Indeed, by the inductive hypothesis, $M'_{x_1/d_1,\ldots x_n/d_n}[\psi] = M_{x_1/d_1,\ldots x_n/d_n}[\psi]$, for all $d_1, \ldots d_n \in D$. Furthermore, for all $d_1, \ldots d_n \in D$ then $M'_{x_1/d_1,\ldots x_n/d_n}[\gamma] = M_{x_1/d_1,\ldots x_n/d_n}[\gamma]$. This is trivial for GF and LGF thanks to the inductive hypothesis, since guards are Boolean combinations of atoms. This is also true for PGF, given the previous remarks on existentially-quantified atomic formulas. Hence, $M'_{x_1/d_1,\ldots x_n/d_n}[\gamma \to \psi] = M_{x_1/d_1,\ldots x_n/d_n}[\gamma \to \psi]$.

It follows that $M'[\varphi] = M[\varphi]$. The existential case is handled similarly. □

Corollary 1. *Any theory in* GF, LGF, *or* PGF *is shiny.*

Proof. Assume $\mathcal{T}$ is a theory in GF, LGF, or PGF. For any set of literals L in the language of $\mathcal{T}$, $\mathcal{T} \cup L$ is also a theory in GF, LGF, or PGF. Thanks to the finite model property of GF, LGF, and PGF, if $\mathcal{T} \cup L$ is satisfiable, it has a finite model. It is thus possible to compute the minimum cardinality of $\mathcal{T} \cup L$. Furthermore, thanks to the previous theorem, its spectrum is an unbounded interval. □

5 The Ackermann Class

The Ackermann class (with equality) is the set of formulas of the form

$$\exists \,.\, z_1 \cdots \exists z_n \,.\, \forall x \,.\, \exists y_1 \,.\, \cdots \exists y_m \,.\, \varphi(x, y_1, \ldots, y_m, z_1, \ldots, z_n),$$

where $\varphi(x, y_1, \ldots, y_m, z_1, \ldots, z_n)$ is quantifier-free and function-free. Checking the satisfiability of formulas of the above form can be reduced (using Skolemization) to checking the satisfiability of formulas without existental quantifiers of the form $\psi = \forall x \,.\, \varphi(x, f_1(x), \ldots, f_m(x))$, where $\varphi(x, y_1, \ldots, y_m)$ is quantifier-free and function-free.

Theorem 6. *The class of formulas of the form $\psi = \forall x \,.\, \varphi(x, f_1(x), \ldots, f_m(x))$, where $\varphi(x, y_1, \ldots, y_m)$ is quantifier and function-free (constants are allowed) has the finite model property.*

The proof may be found for instance in [5]. The following theorem will allow to determine that the Ackermann theories are gentle. An equivalent property for the Ackermann fragment is discussed in [6].

Theorem 7. *Consider a formula $\psi = \forall x \,.\, \varphi(x, f_1(x), \ldots, f_m(x))$, where formula $\varphi(x, y_1, \ldots, y_m)$ is quantifier and function-free (constants are allowed). If ψ has a model of cardinality κ strictly greater than the number of constants in ψ, then it has models with any cardinality greater than κ.*

Proof. Consider a model $\mathcal{M}$ of ψ on domain D such that $|D|$ is greater than the number of constants in ψ. Then there exists an extension $\mathcal{M}'$ on any domain D' with $D \subset D'$ that is also a model of ψ.

Let $\Phi(x) = \varphi(x, f_1(x), \ldots, f_m(x))$ and $d \in D$ be an element of the domain, not assigned by $\mathcal{M}$ to a constant in the formula. Obviously $\mathcal{M}_{x/d}$ is a model of $\Phi(x)$. Consider $d' \in D' \setminus D$. For every n-ary predicate p, and n-uple $\mathbf{d}'$ of elements in $(D \setminus \{d\}) \cup \{d'\}$, let $\mathbf{d}$ be a n-uple of elements in D obtained from $\mathbf{d}'$ by changing d' by d whenever d' is an element of the tuple $\mathbf{d}'$, and set $\mathcal{M}'[p](\mathbf{d}') = \mathcal{M}[p](\mathbf{d})$. For every function f_i $(1 \leq i \leq m)$ let $\mathcal{M}'[f_i](d') = \mathcal{M}[f](d)$ if $\mathcal{M}[f](d) \neq d$, and let $\mathcal{M}'[f_i](d') = d'$ otherwise. Functions and predicates are only partially defined above, but they can be completed arbitrarily without any influence on the result. One can show by structural induction that $\mathcal{M}'_{x/d'}[\Phi(x)] = \mathcal{M}'_{x/d}[\Phi(x)]$.

Indeed, according to our definition of $\mathcal{M}'$,

- $\mathcal{M}'_{x/d'}[x] = d'$ whereas $\mathcal{M}_{x/d}[x] = d$,
- $\mathcal{M}'_{x/d'}[c] = \mathcal{M}_{x/d}[c]$ for every constant c in $\Phi(x)$,
- $\mathcal{M}'_{x/d'}[f(x)] = \mathcal{M}_{x/d}[f(x)]$ if $\mathcal{M}_{x/d}[f(x)] \neq d$,
- $\mathcal{M}'_{x/d'}[f(x)] = d'$ if $\mathcal{M}_{x/d}[f(x)] = d$.

Thus, for every atom $p(t_1, \ldots, t_n)$ (respectively, $t_1 = t_2$) in $\Phi(x)$, $\mathcal{M}'_{x/d'}$ and $\mathcal{M}'_{x/d}$ assign the same values to every t_i except that $\mathcal{M}'_{x/d'}$ assigns d' instead of d for $\mathcal{M}'_{x/d}$. Finally, thanks to the way $\mathcal{M}'$ extends the assignment of predicates, $\mathcal{M}'_{x'/d'}[p(t_1, \ldots, t_n)] = \mathcal{M}_{x/d}[p(t_1, \ldots, t_n)]$ (respectively, $\mathcal{M}'_{x'/d'}[t_1 = t_2] = \mathcal{M}_{x/d}[t_1 = t_2]$). □

Corollary 2. *The spectrum of an Ackermann theory can be computed and expressed either as a finite set of natural numbers, or as the union of a finite set of natural numbers with the set of all the (finite or infinite) cardinalities greater than a natural. The Ackermann theories are gentle.*

Proof. Given $\psi = \forall x \,.\, \varphi(x, f_1(x), \ldots, f_m(x))$, where formula $\varphi(x, y_1, \ldots, y_m)$ is quantifier and function-free, it is possible to establish if ψ has a model of cardinality greater than the number n of constants in ψ. Indeed, formula $\psi' = \psi \wedge \bigwedge_{0 \leq i < j \leq n} a_i \neq a_j$ (where the a_i's are fresh constants) is also in the decidable Ackermann class, and is satisfiable if and only if ψ has a model of cardinality greater than or equal to $n+1$. If ψ' is unsatisfiable, ψ has no model of cardinality greater than or equal to $n+1$. If ψ' is satisfiable, a decision procedure to get the smallest cardinality $m > n$ of the models of ψ can just be a simple test of the (finite) interpretations of increasing cardinality size starting from $n+1$; this procedure will indeed terminate, and ψ will accept models for every cardinality greater than or equal to m. It then only remains to check if ψ accepts models for the cardinalities between 1 and n, which can be done by considering the finitely many interpretations. □

6 Conclusions

The first frameworks to combine disjoint decidable theories were very restrictive: the combined theories were required to be stably infinite. Later results led to more liberal frameworks. In particular, it was proved in [19] that shiny theories are combinable with any other disjoint decidable theory. We have showed that any theory in the guarded fragment, in the loosely guarded fragment, or in the packed guarded fragment, is shiny.

Another well known decidable class with equality (the only relevant classes in our context) that was not yet proved to have good combining properties is the Ackermann class. We showed here that, although not shiny, Ackermann theories are gentle, and, as such, are combinable with non-stably infinite theories with minor requirements. Together with [7], this work then covers the major first-order decidable classes. Interestingly, *all of them are at least gentle.*

The Rabin and the Shelah classes are, respectively, extensions of the Löwenheim class (studied in [7]) and the Ackermann class (studied here), with one unary function. Both are decidable [5]. However, both have infinity axioms [5], and they also contain formulas restricting the cardinality of their models to a finite number. Hence, they are neither shiny nor gentle, and not even stably infinite. It is still an open problem whether these classes have spectral properties that allow liberal combinations. A solution to this problem would probably involve more complex combination frameworks than those discussed in this paper.

Guarded fragments have been extended beyond PGF, even to include fragments of second order logic. The fixed point guarded fragment μGF, for example, was introduced in [11] extending GF with fixed point operators. But unlike GF, LGF, and PGF, μGF even though decidable, does not have the finite model property, and hence it is not shiny. We conjecture, though, that Theorem 5 can be extended to μGF proving it stably infinite.

Our motivation here was mainly to study the decidability of combinations of disjoint theories, without having a practical applications in mind. However, the guarded fragments are highly promising from the point of view of applications.

Indeed, since they can easily express graph properties, we beleive implementations will trigger concrete applications. As a toy example of what can be handled by a combination with the guarded fragments, consider the conjunction of the following formulas[4]:

$$\forall x\, y\,.\, R(x,y) \rightarrow \forall z\,.\,\big(R(y,z) \wedge R(z,x)\big) \rightarrow (x = y \vee y = z \vee z = x)$$
$$R(a,b) \wedge R(b,c) \wedge R(a,c)$$
$$f(b) = f(a) + 1 \wedge f(c) = f(b) + 1$$

This set of formulas is unsatisfiable: the first formula enforces 3-edges loop to have at least one reflexive edge, the second states the existence of a 3-edge loop through a, b and c, and the last formula (using uninterpreted function f and some arithmetic) enforces a, b and c to be distinct, which leads to a contradiction. This formula can be dealt with a classical Nelson-Oppen combination framework since all theories are stably-infinite.

In [22], the authors show that it is possible to combine non-disjoint theories from various decidable classes, those theories sharing monadic predicates. This results in a very expressive language. A future direction for research will be to study if the guarded fragments can also be included in such a framework for combining *non-disjoint* theories.

Acknowledgment. We would like to thank the anonymous reviewers for their helpful comments and suggestions.

References

1. Andréka, H., Németi, I., van Benthem, J.: Modal logics and bounded fragments of predicate logic. Journal of Philosophical Logic 27(3), 217–274 (1998)
2. Barrett, C., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Biere, A., Heule, M.J.H., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185, ch. 26, pp. 825–885. IOS Press, Amsterdam (2009)
3. Blackburn, P., de Rijke, M., Venema, Y.: Modal Logic. Cambridge University Press, Cambridge (2001)
4. Blackburn, P., Wolter, F., van Benthem, J. (eds.): Handbook of Modal Logics. Elsevier, Amsterdam (2006)
5. Börger, E., Grädel, E., Gurevich, Y.: The Classical Decision Problem. Perspectives in Mathematical Logic. Springer, Berlin (1997)
6. Dreben, B., Goldfarb, W.D.: The Decision Problem: Solvable Classes of Quantificational Formulas. Addison-Wesley, Reading (1979)
7. Fontaine, P.: Combinations of theories for decidable fragments of first-order logic. In: Ghilardi, S., Sebastiani, R. (eds.) FroCoS 2009. LNCS, vol. 5749, pp. 263–278. Springer, Heidelberg (2009)

[4] The first formula can also be written in the Bernays-Schönfinkel-Ramsey class, which contains only gentle theories. It is thus more convenient to consider this formula as in the LGF fragment, containing only shiny theory.

8. Fontaine, P., Gribomont, E.P.: Combining non-stably infinite, non-first order theories. In: Ahrendt, W., Baumgartner, P., de Nivelle, H., Ranise, S., Tinelli, C. (eds.) Selected Papers from the Workshops on Disproving and the Second International Workshop on Pragmatics of Decision Procedures (PDPAR 2004). ENTCS, vol. 125, pp. 37–51 (2005)
9. Grädel, E.: Why are modal logics so robustly decidable? In: Current Trends in Theoretical Computer Science. Entering the 21st Century, pp. 393–408. World Scientific, Singapore (2001)
10. Grädel, E.: Guarded fixed point logics and the monadic theory of countable trees. Theoretical Computer Science 288(1), 129–152 (2002)
11. Grädel, E., Walukiewicz, I.: Guarded fixed point logic. In: Logic In Computer Science (LICS), pp. 45–54. IEEE Computer Society Press, Washington, USA (1999)
12. Grädel, E.: On the restraining power of guards. Journal of Symbolic Logic 64, 1719–1742 (1998)
13. Grädel, E.: Decision procedures for guarded logics. In: Ganzinger, H. (ed.) CADE 1999. LNCS (LNAI), vol. 1632, pp. 31–51. Springer, Heidelberg (1999)
14. Hodkinson, I.M.: Loosely guarded fragment of first-order logic has the finite model property. Studia Logica 70(2), 205–240 (2002)
15. Marx, M.: Tolerance logic. Journal of Logic, Language and Information 10(3), 353–374 (2001)
16. Nelson, G., Oppen, D.C.: Simplifications by cooperating decision procedures. ACM Transactions on Programming Languages and Systems 1(2), 245–257 (1979)
17. Tinelli, C., Harandi, M.T.: A new correctness proof of the Nelson–Oppen combination procedure. In: Baader, F., Schulz, K.U. (eds.) Frontiers of Combining Systems (FroCoS), pp. 103–120. Kluwer, Dordrecht (1996)
18. Tinelli, C., Ringeissen, C.: Unions of non-disjoint theories and combinations of satisfiability procedures. Theoretical Computer Science 290(1), 291–353 (2003)
19. Tinelli, C., Zarba, C.G.: Combining non-stably infinite theories. Journal of Automated Reasoning 34(3), 209–238 (2005)
20. van Benthem, J.: Dynamic bits and pieces. Technical Report LP-1997-01, ILLC, University of Amsterdam (January 1997)
21. Vardi, M.: Why is modal logic so robustly decidable? DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 31, pp. 149–184. AMS, Providence (1997)
22. Wies, T., Piskac, R., Kuncak, V.: Combining Theories with Shared Set Operations. In: Ghilardi, S., Sebastiani, R. (eds.) FroCoS 2009. LNCS, vol. 5749, pp. 366–382. Springer, Heidelberg (2009)

On the Undecidability of Fuzzy Description Logics with GCIs and Product T-norm

Franz Baader and Rafael Peñaloza

Theoretical Computer Science, TU Dresden, Germany
{baader,penaloza}@tcs.inf.tu-dresden.de

Abstract. The combination of Fuzzy Logics and Description Logics (DLs) has been investigated for at least two decades because such fuzzy DLs can be used to formalize imprecise concepts. In particular, tableau algorithms for crisp Description Logics have been extended to reason also with their fuzzy counterparts. Recently, it has been shown that, in the presence of general concept inclusion axioms (GCIs), some of these fuzzy DLs actually do not have the finite model property, thus throwing doubt on the correctness of tableau algorithm for which it was claimed that they can handle fuzzy DLs with GCIs.

In a previous paper, we have shown that these doubts are indeed justified, by proving that a certain fuzzy DL with product t-norm and involutive negation is undecidable. In the present paper, we show that undecidability also holds if we consider a t-norm-based fuzzy DL where disjunction and involutive negation are replaced by the constructor implication, which is interpreted as the residuum. The only condition on the t-norm is that it is a continuous t-norm "starting" with the product t-norm, which covers an uncountable family of t-norms.

1 Introduction

Description logics (DLs) [1] are a family of logic-based knowledge representation formalisms, which can be used to represent the conceptual knowledge of an application domain in a structured and formally well-understood way. They were employed in various application domains, such as natural language processing, configuration, and databases, but their main breakthrough arguably came with the adoption of the DL-based language OWL [17] as standard ontology language for the semantic web. Another successful application area for DLs is the definition of medical ontologies, such as SNOMED CT[1] and GALEN.[2]

In Description Logics, concepts are formally described by *concept descriptions*, i.e., expressions that are built from concept names (unary predicates) and role names (binary predicates) using concept constructors. The expressivity of a particular DL is determined by which concept constructors are available in it. From a semantic point of view, concept names and concept descriptions represent sets

[1] http://www.ihtsdo.org/snomed-ct/
[2] http://www.opengalen.org/

C. Tinelli and V. Sofronie-Stokkermans (Eds.): FroCoS 2011, LNAI 6989, pp. 55–70, 2011.

of individuals, whereas roles represent binary relations between individuals. For example, using the concept names Patient and Running-nose, and the role name has-symptom, the concept of all *patients with running noses* can be represented by the concept description

$$\mathsf{Patient} \sqcap \exists \mathsf{has\text{-}symptom}.\mathsf{Running\text{-}nose}.$$

In addition to the description language (i.e., the formalism for constructing concept descriptions), DLs provide their users with a terminological and an assertional formalism. In its simplest form, a DL *terminology* (usually called *TBox*) can be used to introduce abbreviations for complex concept descriptions. For example, the *concept definition*

$$\mathsf{Private\text{-}patient} \equiv \mathsf{Patient} \sqcap \exists \mathsf{has\text{-}insurance}.\mathsf{Private\text{-}health}$$

says that private patients are patients that have a private health insurance. So-called *general concept inclusions (GCIs)* can be used to state additional constraints on the interpretation of concepts and roles. In our medical example, one could express that patients with running noses have a cold or hay fever using the GCI

$$\mathsf{Patient} \sqcap \exists \mathsf{has\text{-}symptom}.\mathsf{Running\text{-}nose} \sqsubseteq \exists \mathsf{has\text{-}disease}.(\mathsf{Cold} \sqcup \mathsf{Hay\text{-}fever}).$$

Note that the concept definition $A \equiv C$ can be expressed using the two GCIs $A \sqsubseteq C$ and $C \sqsubseteq A$.

In the *assertional part (ABox)* of a DL-based ontology, facts about a specific application situation can be stated, by introducing named individuals and relating them to concepts and roles. For example, the assertions

$$\mathsf{LINDA} : \mathsf{Patient},\ (\mathsf{LINDA}, \mathsf{AXA\text{-}PPP}) : \mathsf{has\text{-}insurance},\ \mathsf{AXA\text{-}PPP} : \mathsf{Private\text{-}health}$$

state that Linda is a patient that has the private health insurance AXA-PPP. An *ontology* is a TBox together with an ABox, i.e., finite set of GCIs and assertions.

Knowledge representation systems based on DLs provide their users with various inference services that allow them to deduce implicit knowledge from the explicitly represented knowledge. For example, given the concept definition and the assertions of our example, one can deduce the assertion $\mathsf{LINDA} : \mathsf{Private\text{-}patient}$, i.e., that Linda is a private patient. An important inference service for DL-based ontologies is testing their consistency, i.e., checking whether a given ontology is non-contradictory by testing whether it has a model. In fact, all the other standard inference problems can be reduced to consistency.

Fuzzy variants of Description Logics (DLs) were introduced in order to deal with applications where membership to concepts cannot always be determined in a precise way. For example, assume that we want to express that a patient that has a high temperature and a running nose has a cold using the GCI

$$\mathsf{Patient} \sqcap \exists \mathsf{has\text{-}symptom}.\mathsf{Running\text{-}nose} \sqcap \exists \mathsf{has\text{-}temperature}.\mathsf{High} \sqsubseteq \exists \mathsf{has\text{-}disease}.\mathsf{Cold}.$$

Here it makes sense to view High as a fuzzy concept, to which 36°C belongs with a low membership degree (say 0.2), 38°C with a higher membership degree (say 0.7), and 40°C with an even higher membership degree (say 0.9). In the presence of such fuzzy concepts, ABox assertions must then be equipped with a membership degree. For example, the assertion $\langle \mathsf{T}_1 : \mathsf{High} \geq 0.8 \rangle$ says that temperature T_1 is high with membership degree at least 0.8. If we are not so sure about the measurement (e.g., if it was taken under the armpit), we could also equip the role assertion $(\mathsf{LINDA}, \mathsf{T}_1) : \mathsf{has\text{-}temperature}$ with a membership degree smaller than 1. The use of fuzzy concepts in medical applications is, for instance, described in more detail in [19].

A great variety of fuzzy DLs have been investigated in the literature [18,14]. In fact, compared to crisp DLs, fuzzy DLs offer an additional degree of freedom when defining their expressiveness: in addition to deciding which concept constructors (like conjunction $\sqcap$, disjunction $\sqcup$, existential restriction $\exists r.C$) and which terminological formalism (like no TBox, acyclic concept definitions, general concept inclusions) to use, one must also decide how to interpret the concept constructors by appropriate functions on the domain of fuzzy values $[0, 1]$. For example, conjunction can be interpreted by different t-norms (such as Gödel, Łukasiewicz, and product) and there are also different options for how to interpret negation (such as involutive negation and residual negation). In addition, one can either consider all models or only so-called witnessed models [16] when defining the semantics of fuzzy DLs.

Decidability of fuzzy DLs is often shown by adapting the tableau-based algorithms [3] for the corresponding crisp DL to the fuzzy case. This was first done for the case of DLs without general concept inclusion axioms (GCIs) [26,24,22,9], but then also extended to GCIs [23,25,7,8]. Usually, these tableau algorithm reason w.r.t. witnessed models.[3] It should be noted, however, that in the presence of GCIs there are different ways of extending the notion of witnessed models from [16], depending on whether the witnessed property is required to apply also to GCIs (in which case we talk about strongly witnessed models) or not (in which case we talk about witnessed models).

The paper [7] considers the case of reasoning w.r.t. fuzzy GCIs in the setting of a logic with product t-norm and involutive negation. More precisely, the tableau algorithm introduced in that paper is supposed to check whether an ontology consisting of fuzzy GCIs and fuzzy ABox assertions expressed in this DL has a strongly witnessed model or not.[4] Actually, the proof of correctness of this algorithm given in [7] implies that, whenever such an ontology has a strongly witnessed model, then it has a finite model. However, it was recently shown in [4] that this is not the case in the presence of general concept inclusion axioms, i.e., there is an ontology written in this logic that has a strongly witnessed model, but does not have a finite model. Of course, this does not automatically imply

[3] In fact, witnessed models were introduced in [16] to correct the proof of correctness for the tableau algorithm presented in [26].

[4] Note that the authors of [7] actually use the term "witnessed models" for what we call "strongly witnessed models."

that the algorithm itself is wrong. In fact, if one applies the algorithm from [7] to the ontology used in [4] to demonstrate the failure of the finite model property, then one obtains the correct answer, and in [4] the authors actually conjecture that the algorithm is still correct. However, incorrectness of the algorithm has now independently been shown in [5] and in [2]. Thus, one can ask whether the fuzzy DL considered in [7] is actually decidable. Though this question is not answered in [2], the paper gives strong indications that the answer might in fact be "no." More precisely, [2] contains a proof of undecidability for a variant of the fuzzy DL considered in [7], which (i) additionally allows for strict GCIs, i.e., GCIs whose fuzzy value is required to be *strictly greater* than a given rational number; and (ii) where the notion of strongly witnessed models used in [7] is replaced by the weaker notion of witnessed models.

In the present paper, we show that, in the presence of GCIs, undecidability also holds if we consider a t-norm-based fuzzy DL where disjunction and involutive negation are replaced by the constructor implication, which is interpreted as the residuum.[5] The only condition on the t-norm is that it is a continuous t-norm "starting" with the product t-norm. In particular, this includes the fuzzy DL with product t-norm introduced in [16], where decidability of reasoning w.r.t. witnessed models was shown for the case without GCIs. In [13], an analogous decidability result was shown for the case of reasoning w.r.t. so-called quasi-witnessed models. Following [13], we call this logic $*$-$\mathcal{ALE}$. Note that our undecidability result holds for several variants of the notion of witnessed models (including witnessed, quasi-witnessed, and strongly witnessed models).

In the next section, we introduce basic notions from fuzzy logics, and in Section 3 we introduce the fuzzy DLs considered in this paper. In Section 4 we then show undecidability of these DLs w.r.t. witnessed and quasi-witnessed models, and in Section 5 w.r.t. strongly witnessed and finite models.

2 T-norms and Fuzzy Logic

Fuzzy logics are formalisms introduced to express imprecise or vague information [15]. They extend classical logic by interpreting predicates as fuzzy sets over an interpretation domain. Given a non-empty domain Δ, a *fuzzy set* is a function $F : \Delta \to [0,1]$ from Δ into the real unit interval $[0,1]$, with the intuition that an element $\delta \in \Delta$ belongs to F with *degree* $F(\delta)$. The interpretation of the logical constructors is based on appropriate truth functions that generalize the properties of the connectives of classical logic to the interval $[0,1]$. The most prominent truth functions used in the fuzzy logic literature are based on t-norms.

A *t-norm* is a binary operator $\otimes : [0,1] \times [0,1] \to [0,1]$ that is associative and commutative, has 1 as its unit element, and is monotonic, i.e., for every $x, y, z \in [0,1]$, if $x \leq y$, then $x \otimes z \leq y \otimes z$. The t-norm $\otimes$ is *continuous* if it is continuous as a function, i.e., we have for all convergent sequences $\{x_n\}_{n\geq 0}, \{y_n\}_{n\geq 0}$ that

$$(\lim_{n\to\infty} x_n) \otimes (\lim_{n\to\infty} y_n) = \lim_{n\to\infty} (x_n \otimes y_n).$$

[5] This change of the constructors used is not irrelevant: in general, disjunction and involutive negation cannot be expressed using only conjunction and residua.

Table 1. Gödel, product and Łukasiewicz t-norms and their residua

Name	t-norm $(x \otimes y)$	Residuum $(x \Rightarrow y)$
Gödel	$\min(x, y)$	$\begin{cases} 1 & \text{if } x \leq y \\ y & \text{otherwise} \end{cases}$
product	$x \cdot y$	$\begin{cases} 1 & \text{if } x \leq y \\ y/x & \text{otherwise} \end{cases}$
Łukasiewicz	$\max(x + y - 1, 0)$	$\min(1 - x + y, 1)$

If $\otimes$ is a continuous t-norm, then there exists a unique binary operator $\Rightarrow$, called the *residuum*, that satisfies $z \leq x \Rightarrow y$ iff $x \otimes z \leq y$ for every $x, y, z \in [0, 1]$. Three important continuous t-norms are the Gödel, product and Łukasiewicz t-norms. These t-norms and their corresponding residua are shown in Table 1.

The following are simple consequences of the definition of t-norms and their residua (see [15], Lemma 2.1.6).

Lemma 1. *For every t-norm $\otimes$ and $x, y \in [0, 1]$ the following hold:*

- $x \Rightarrow y = 1$ *iff* $x \leq y$,
- $1 \Rightarrow y = y$, $0 \Rightarrow y = 1$, *and*
- *if* $x > 0$, *then* $x \Rightarrow 0 = 0$.

The t-norms described in Table 1 are *fundamental* in the sense that all other continuous t-norms can be constructed from them: every continuous t-norm can be expressed as the ordered sum of copies of Łukasiewicz, Gödel and product t-norms [20]. More formally, if $\otimes$ is a continuous t-norm, then there exists a (possibly infinite) family $\mathcal{S} = \{\langle (a_i, b_i), \otimes_i \rangle \mid i \in \mathcal{J}\}$, where (a_i, b_i) are non-empty, pairwise disjoint open subintervals of $[0, 1]$ and $\otimes_i$ is either the Łukasiewicz or the product t-norm, such that

$$x \otimes y = \begin{cases} a_i + (b_i - a_i) \cdot (\frac{x - a_i}{b_i - a_i} \otimes_i \frac{y - a_i}{b_i - a_i}) & \text{if } x, y \in [a_i, b_i] \text{ for some } i \in \mathcal{J} \\ \min(x, y) & \text{otherwise} \end{cases}$$

holds for all $x, y \in [0, 1]$. The residuum of this t-norm is given, for every $x, y \in [0, 1]$, by

$$x \Rightarrow y = \begin{cases} 1 & \text{if } x \leq y \\ a_i + (b_i - a_i) \cdot (\frac{x - a_i}{b_i - a_i} \Rightarrow_i \frac{y - a_i}{b_i - a_i}) & \text{if } a_i \leq y < x \leq b_i \text{ for some } i \in \mathcal{J} \\ y & \text{otherwise,} \end{cases}$$

where $\Rightarrow_i$ represents the residuum of the t-norm $\otimes_i, i \in \mathcal{J}$.

In this paper we will focus on t-norms whose expression as an ordered sum use the product t-norm as "first element."

Definition 2. *Given a t-norm $\otimes$ obtained as ordered sum from the family $\mathcal{S} = \{\langle (a_i, b_i), \otimes_i \rangle \mid i \in \mathcal{J}\}$ and a number $q \in (0, 1]$, we say that $\otimes$* q-starts with the

product t-norm *(q-starts with Π for short) if there is an index $i \in \mathcal{J}$ such that $(a_i, b_i) = (0, q)$ and $\otimes_i$ is the product t-norm. It* starts with *Π if it q-starts with Π for some $q \in (0, 1]$.*

Notice that, for every $q \in (0, 1)$, there exist uncountably many t-norms that q-start with Π. In fact, for every real number $r \in (q, 1]$, we can take the family $\{\langle (0, q), \otimes_1 \rangle, \langle (q, r), \otimes_2 \rangle\}$ where $\otimes_1$ is the product t-norm and $\otimes_2$ is the Łukasiewicz t-norm. As a simple consequence of this, there are uncountably many continuous t-norms that q-start with Π for a rational number q. Our undecidability proofs will only deal with such t-norms. The following lemma is a simple consequence of the properties described before.

Lemma 3. *For a given t-norm $\otimes$ and $q \in (0, 1]$, if $\otimes$ q-starts with Π, then for every $x, y \in [0, q]$ the following holds:*

- $x \otimes y = (x \cdot y)/q$, *and*
- *if $x > y$, then $x \Rightarrow y = q \cdot (y/x)$.*

3 Fuzzy Description Logics

In this section, we introduce the fuzzy description logic $*$-$\mathcal{ALE}$ and some of its properties, which will be useful throughout the paper.

The syntax of this logic is slightly different from standard description logics, as it has an implication constructor, but no negation or disjunction constructors. $*$-$\mathcal{ALE}$ *concepts* are built through the syntactic rule

$$C ::= A \mid \bot \mid \top \mid C_1 \sqcap C_2 \mid C_1 \to C_2 \mid \exists r.C \mid \forall r.C$$

where A is a *concept name* and r is a *role name*.

A $*$-$\mathcal{ALE}$ *ABox* is a finite set of *assertion axioms* of the form $\langle a : C \rhd q \rangle$ or $\langle (a, b) : r \rhd q \rangle$, where C is a $*$-$\mathcal{ALE}$ concept, $r \in \mathcal{N}_\mathcal{R}$, q is a rational number in the interval $[0, 1]$, a, b are *individual names* and $\rhd$ is either $\geq$ or $=$. A $*$-$\mathcal{ALE}$ *TBox* is a finite set of *concept inclusion axioms* of the form $\langle C \sqsubseteq D \geq q \rangle$, where C, D are $*$-$\mathcal{ALE}$ concepts and q is a rational number in $[0, 1]$. A $*$-$\mathcal{ALE}$ *ontology* is a tuple $(\mathcal{A}, \mathcal{T})$, where $\mathcal{A}$ is a $*$-$\mathcal{ALE}$ ABox and $\mathcal{T}$ a $*$-$\mathcal{ALE}$ TBox. For the rest of the paper we will often drop the prefix $*$-$\mathcal{ALE}$, and speak simply of e.g. *TBoxes* and *ontologies*.

The semantics of this logic extend the classical DL semantics by interpreting concepts and roles as fuzzy sets over an interpretation domain. The precise semantics depends on the t-norm chosen; thus, in the following, we assume that we have an arbitrary, but fixed, continuous t-norm $\otimes$ and that $\Rightarrow$ is the associated residuum. The semantics of $*$-$\mathcal{ALE}$ is based on interpretations. An *interpretation* is a tuple $\mathcal{I} = (\Delta^\mathcal{I}, \cdot^\mathcal{I})$ where $\Delta^\mathcal{I}$ is a non-empty set, called the *domain*, and the function $\cdot^\mathcal{I}$ maps each individual name a to an element of $\Delta^\mathcal{I}$, each concept name A to a function $A^\mathcal{I} : \Delta^\mathcal{I} \to [0, 1]$ and each role name r to a function $r^\mathcal{I} : \Delta^\mathcal{I} \times \Delta^\mathcal{I} \to [0, 1]$. The interpretation function is extended to arbitrary $*$-$\mathcal{ALE}$ concepts as follows. For every $\delta \in \Delta^\mathcal{I}$,

$$\begin{aligned}
\top^{\mathcal{I}}(\delta) &= 1,\\
\bot^{\mathcal{I}}(\delta) &= 0,\\
(C_1 \sqcap C_2)^{\mathcal{I}}(\delta) &= C_1^{\mathcal{I}}(\delta) \otimes C_2^{\mathcal{I}}(\delta)\\
(C_1 \to C_2)^{\mathcal{I}}(\delta) &= C_1^{\mathcal{I}}(\delta) \Rightarrow C_2^{\mathcal{I}}(\delta)\\
(\exists r.C)^{\mathcal{I}}(\delta) &= \sup_{\gamma \in \Delta^{\mathcal{I}}} r^{\mathcal{I}}(\delta,\gamma) \otimes C^{\mathcal{I}}(\gamma)\\
(\forall r.C)^{\mathcal{I}}(\delta) &= \inf_{\gamma \in \Delta^{\mathcal{I}}} r^{\mathcal{I}}(\delta,\gamma) \Rightarrow C^{\mathcal{I}}(\gamma).
\end{aligned}$$

The interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ satisfies the assertional axiom $\langle a : C \rhd q\rangle$ iff $C^{\mathcal{I}}(a^{\mathcal{I}}) \rhd q$, it satisfies $\langle (a,b) : r \rhd q\rangle$ iff $r^{\mathcal{I}}(a^{\mathcal{I}}, b^{\mathcal{I}}) \rhd q$ and it satisfies the concept inclusion $\langle C \sqsubseteq D \geq q\rangle$ iff $\inf_{\delta \in \Delta^{\mathcal{I}}}(C^{\mathcal{I}}(\delta) \Rightarrow D^{\mathcal{I}}(\delta)) \geq q$. This interpretation is called a *model* of the ontology $\mathcal{O}$ if it satisfies all the axioms in $\mathcal{O}$.

In fuzzy DLs, reasoning is often restricted to witnessed models [16]. An interpretation $\mathcal{I}$ is called *witnessed* if it satisfies the following two conditions:

(wit1) for every $\delta \in \Delta^{\mathcal{I}}$, role r and concept C there exists $\gamma \in \Delta^{\mathcal{I}}$ such that $(\exists r.C)^{\mathcal{I}}(\delta) = r^{\mathcal{I}}(\delta,\gamma) \cdot C^{\mathcal{I}}(\gamma)$, and

(wit2) for every $\delta \in \Delta^{\mathcal{I}}$, role r and concept C there exists $\gamma \in \Delta^{\mathcal{I}}$ such that $(\forall r.C)^{\mathcal{I}}(\delta) = r^{\mathcal{I}}(\delta,\gamma) \Rightarrow C^{\mathcal{I}}(\gamma)$.

This model is called *weakly witnessed* if it satisfies **(wit1)** and *quasi-witnessed* if it satisfies **(wit1)** and the condition

(wit2') for every $\delta \in \Delta^{\mathcal{I}}$, role r and concept C, either $(\forall r.C)^{\mathcal{I}} = 0$ or there exists $\gamma \in \Delta^{\mathcal{I}}$ such that $(\forall r.C)^{\mathcal{I}}(\delta) = r^{\mathcal{I}}(\delta,\gamma) \Rightarrow C^{\mathcal{I}}(\gamma)$.

In the presence of GCIs, witnessed interpretations are sometimes further restricted [9,4,14] to satisfy

(wit3) for every two concepts C, D, there is a γ such that

$$\inf_{\eta \in \Delta^{\mathcal{I}}}(C^{\mathcal{I}}(\eta) \Rightarrow D^{\mathcal{I}}(\eta)) = C^{\mathcal{I}}(\gamma) \Rightarrow D^{\mathcal{I}}(\gamma).$$

Witnessed interpretations that satisfy this third restriction **(wit3)** are called *strongly witnessed* interpretations.

We say that an ontology $\mathcal{O}$ is *consistent* (resp. *weakly witnessed consistent*, *quasi-witnessed consistent*, *witnessed consistent*, *strongly witnessed consistent*) if it has a model (resp. a weakly witnessed model, a quasi-witnessed model, a witnessed model, a strongly witnessed model). Obviously, strongly witnessed consistency implies witnessed consistency, which implies quasi-witnessed consistency, which itself implies weakly witnessed consistency. The converse implications, however, need not hold; for instance, a quasi-witnessed consistent $*$-$\mathcal{ALE}$ ontology that has no witnessed models can be derived from the example in [13].

Witnessed models were introduced to simplify the construction of tableau-based reasoning procedures for fuzzy DLs [16]. Intuitively, with the general semantics for existential restrictions, interpreted as a supremum, it is possible that

an existential restriction is interpreted with a value that is never reached; that is, $(\exists r.C)^{\mathcal{I}}(\delta) > r^{\mathcal{I}}(\delta,\eta) \otimes C^{\mathcal{I}}(\eta)$ for all $\eta \in \Delta^{\mathcal{I}}$. Given an existential restriction, a tableau-based procedure tries to introduce one successor individual that yields this value. Condition **(wit1)** ensures that this approach is sound. Without it, the procedure would also need to address the case where there are infinitely many successors yielding values whose supremum is the value of the existential restriction. It is not clear how to do this with a terminating procedure.

We now derive some properties of the $*$-$\mathcal{ALE}$ axioms and introduce useful abbreviations. First, recall that, for all $x, y \in [0,1]$, it holds that $x \Rightarrow y = 1$ iff $x \leq y$ (Lemma 1). Thus, given two concepts C, D, the axiom $\langle C \sqsubseteq D \geq 1\rangle$ expresses that $C^{\mathcal{I}}(\delta) \leq D^{\mathcal{I}}(\delta)$ for all $\delta \in \Delta^{\mathcal{I}}$.

In the following, we will use the expression $\langle C \overset{r}{\rightsquigarrow} D\rangle$ to abbreviate the axioms $\langle C \sqsubseteq \forall r.D \geq 1\rangle, \langle \exists r.D \sqsubseteq C \geq 1\rangle$. To understand this abbreviation, consider an interpretation $\mathcal{I}$ satisfying $\langle C \overset{r}{\rightsquigarrow} D\rangle$ and let $\delta, \gamma \in \Delta^{\mathcal{I}}$ with $r^{\mathcal{I}}(\delta,\gamma) = 1$. From the first axiom it follows that

$$\begin{aligned} C^{\mathcal{I}}(\delta) \leq (\forall r.D)^{\mathcal{I}}(\delta) &= \inf_{\eta \in \Delta^{\mathcal{I}}} r^{\mathcal{I}}(\delta,\eta) \Rightarrow D^{\mathcal{I}}(\eta) \\ &\leq r^{\mathcal{I}}(\delta,\gamma) \Rightarrow D^{\mathcal{I}}(\gamma) = 1 \Rightarrow D^{\mathcal{I}}(\gamma) = D^{\mathcal{I}}(\gamma). \end{aligned}$$

From the second axiom it follows that

$$\begin{aligned} C^{\mathcal{I}}(\delta) \geq (\exists r.D)^{\mathcal{I}}(\delta) &= \sup_{\eta \in \Delta^{\mathcal{I}}} r^{\mathcal{I}}(\delta,\eta) \otimes D^{\mathcal{I}}(\eta) \\ &\geq r^{\mathcal{I}}(\delta,\gamma) \otimes D^{\mathcal{I}}(\gamma) = 1 \otimes D^{\mathcal{I}}(\gamma) = D^{\mathcal{I}}(\gamma), \end{aligned}$$

and hence, both axioms together imply that $C^{\mathcal{I}}(\delta) = D^{\mathcal{I}}(\gamma)$. In other words, $\langle C \overset{r}{\rightsquigarrow} D\rangle$ expresses that the value of $C^{\mathcal{I}}(\delta)$ is propagated to the valuation of the concept D on all r successors with degree 1 of δ. Conversely, given an interpretation $\mathcal{I}$ such that $r^{\mathcal{I}}(\delta,\gamma) \in \{0,1\}$ for all $\delta, \gamma \in \Delta^{\mathcal{I}}$, if $r^{\mathcal{I}}(\delta,\gamma) = 1$ implies $C^{\mathcal{I}}(\delta) = D^{\mathcal{I}}(\gamma)$, then $\mathcal{I}$ is a model of $\langle C \overset{r}{\rightsquigarrow} D\rangle$.

For a concept C and a natural number $n \geq 1$, the expression C^n denotes the concatenation of C with itself n times, i.e., $C^1 := C$ and $C^{n+1} := C \sqcap C^n$. If $\otimes$ q-starts with Π, then the semantics of $\sqcap$ yields $(C^n)^{\mathcal{I}}(\delta) = (C^{\mathcal{I}}(\delta))^n / q^{n-1}$, for every interpretation $\mathcal{I}$ and every $\delta \in \Delta^{\mathcal{I}}$ with $C^{\mathcal{I}}(\delta) \in [0,q]$ (see Lemma 3).

For the rest of the paper we assume that $\otimes$ q-starts with Π for some arbitrary but fixed rational number $q \in [0,1]$. We will show that, under such a t-norm, consistency of $*$-$\mathcal{ALE}$ ontologies w.r.t. the different variants of witnessed models introduced above is undecidable.

4 Undecidability w.r.t. Witnessed Models

We will show undecidability using a reduction from the Post correspondence problem, which is well-known to be undecidable [21].

Definition 4 (PCP). *Let $((v_1,w_1),\ldots,(v_m,w_m))$ be a finite list of pairs of words over an alphabet $\Sigma = \{1,\ldots,s\}$, $s > 1$. The* Post correspondence problem

(PCP) asks whether there is a non-empty sequence $i_1, i_2, \ldots, i_k$, $1 \leq i_j \leq m$, *such that* $v_{i_1} v_{i_2} \cdots v_{i_k} = w_{i_1} w_{i_2} \cdots w_{i_k}$. *If such a sequence exists, then the word* $i_1 i_2 \cdots i_k$ *is called a* solution *of the problem.*

We assume w.l.o.g. that there is no pair v_i, w_i where both words are empty. For a word $\mu = i_1 i_2 \cdots i_k \in \{1, \ldots, m\}^*$, we will denote as v_μ and w_μ the words $v_{i_1} v_{i_2} \cdots v_{i_k}$ and $w_{i_1} w_{i_2} \cdots w_{i_k}$, respectively.

The alphabet Σ consists of the first s positive integers. We can thus view every word in Σ^* as a natural number represented in base $s+1$ in which 0 never occurs. Using this intuition, we will express the empty word as the number 0.

In the following reductions, we will encode the word w in Σ^* using the number $q \cdot 2^{-w} \in [0, q]$. We will construct an ontology whose models encode the search for a solution. The interpretation of two designated concept names A and B at a node will respectively correspond to the words v_μ and w_μ for $\mu \in \{1, \ldots, m\}^*$.

It should be noted that, in the following constructions, the only relevant values used for interpreting the different concepts will be $[0, q] \cup \{1\}$. For this reason, it is only important that $\otimes$ q-starts with Π, while the precise definition of the t-norm over the rest of the unit interval is irrelevant.

To be more precise, we will show undecidability of consistency w.r.t. witnessed models by constructing, for a given instance $\mathcal{P} = ((v_1, w_1), \ldots, (v_m, w_m))$ of the PCP, an ontology $\mathcal{O}_\mathcal{P}$ such that, for every witnessed model $\mathcal{I}$ of $\mathcal{O}_\mathcal{P}$ and every $\mu \in \{1, \ldots, m\}^*$, there is an element $\delta_\mu \in \Delta^\mathcal{I}$ with $A^\mathcal{I}(\delta_\mu) = q \cdot 2^{-v_\mu}$ and $B^\mathcal{I}(\delta_\mu) = q \cdot 2^{-w_\mu}$. Additionally, we will show that this ontology has a witnessed model whose domain has only these elements. Then, $\mathcal{P}$ has a solution iff for every witnessed model $\mathcal{I}$ of $\mathcal{O}_\mathcal{P}$ there exist a $\delta \in \Delta^\mathcal{I}$ such that $A^\mathcal{I}(\delta) = B^\mathcal{I}(\delta)$.

Let $\delta \in \Delta^\mathcal{I}$ encode the words $v, w \in \Sigma^*$; that is, $A^\mathcal{I}(\delta) = q \cdot 2^{-v}$ and $B^\mathcal{I}(\delta) = q \cdot 2^{-w}$, and let $i, 1 \leq i \leq m$. Assume additionally that we have concept names V_i, W_i with $V_i^\mathcal{I}(\delta) = q \cdot 2^{-v_i}$ and $W_i^\mathcal{I}(\delta) = q \cdot 2^{-w_i}$. We want to ensure the existence of a node γ that encodes the concatenation of the words v, w with the i-th pair from $\mathcal{P}$; i.e. vv_i and ww_i. This is done through the TBox

$$\mathcal{T}_\mathcal{P}^i := \{\langle \top \sqsubseteq \exists r_i.\top \geq 1\rangle, \langle (V_i \sqcap A^{(s+1)^{|v_i|}}) \overset{r_i}{\rightsquigarrow} A\rangle, \langle (W_i \sqcap B^{(s+1)^{|w_i|}}) \overset{r_i}{\rightsquigarrow} B\rangle\}.$$

Recall that we are viewing words in Σ^* as natural numbers in base $s+1$. Thus, the concatenation of two words u, u' corresponds to the operation $u \cdot (s+1)^{|u'|} + u'$. Additionally, $A^\mathcal{I}(\delta) \leq q$ and hence

$$(A^{(s+1)^{|v_i|}})^\mathcal{I}(\delta) = \frac{q^{(s+1)^{|v_i|}} \cdot 2^{-v \cdot (s+1)^{|v_i|}}}{q^{(s+1)^{|v_i|}-1}} = q \cdot 2^{-v \cdot (s+1)^{|v_i|}}.$$

Since $V_i^\mathcal{I}(\delta) \leq q$, we then have

$$(V_i \sqcap A^{(s+1)^{|v_i|}})^\mathcal{I}(\delta) = \frac{(q \cdot 2^{-v_i}) \cdot (q \cdot 2^{-v \cdot (s+1)^{|v_i|}})}{q} = q \cdot 2^{-vv_i}.$$

Analogously, we get that $(W_i \sqcap B^{(s+1)^{|w_i|}})^\mathcal{I}(\delta) = q \cdot 2^{-ww_i}$.

If $\mathcal{I}$ is a witnessed model of $\mathcal{T}_{\mathcal{P}}^i$, then from the first axiom in $\mathcal{T}_{\mathcal{P}}^i$ it follows that $(\exists r_i.\top)^{\mathcal{I}}(\delta) = 1$, and according to **(wit1)**, there must exist a $\gamma \in \Delta^{\mathcal{I}}$ with $r^{\mathcal{I}}(\delta, \gamma) = 1$. The last two axioms then ensure that $A^{\mathcal{I}}(\gamma) = q \cdot 2^{-vv_i}$ and $B^{\mathcal{I}}(\gamma) = q \cdot 2^{-ww_i}$; thus, the concept names A and B encode, at node γ, the words vv_i and ww_i, as desired. If we want to use this construction to recursively construct all the pairs of concatenated words defined by $\mathcal{P}$, we need to ensure also that $V_j^{\mathcal{I}}(\gamma) = q \cdot 2^{-v_j}$, $W_j^{\mathcal{I}}(\gamma) = q \cdot 2^{-w_j}$ hold for every $j, 1 \leq j \leq m$. This can be done through the axioms

$$\mathcal{T}_{\mathcal{P}}^0 := \{\langle V_j \overset{r_i}{\rightsquigarrow} V_j\rangle, \langle W_j \overset{r_i}{\rightsquigarrow} W_j\rangle \mid 1 \leq i, j \leq m\}.$$

It only remains to ensure that there is a node δ_ε where

$$A^{\mathcal{I}}(\delta_\varepsilon) = B^{\mathcal{I}}(\delta_\varepsilon) = q = q \cdot 2^0,$$

that is, where A and B encode the empty word, and for every $j, 1 \leq i \leq m$, $V_j^{\mathcal{I}}(\delta_\varepsilon) = q \cdot 2^{-v_j}$ and $W_j^{\mathcal{I}}(\delta_\varepsilon) = q \cdot 2^{-w_j}$ hold. This condition is easily enforced through the ABox

$$\begin{aligned}\mathcal{A}_{\mathcal{P}}^0 := {} & \{\langle a : A = q\rangle, \langle a : B = q\rangle\} \cup \\ & \{\langle a : V_i = q \cdot 2^{-v_i}\rangle, \langle a : W_i = q \cdot 2^{-w_i}\rangle \mid 1 \leq i \leq m\}.\end{aligned}$$

Finally, we include a concept name H that must be interpreted as $q/2$ in every domain element reachable from a. This is enforced by the following axioms:

$$\begin{aligned}\mathcal{A}_0 &:= \{\langle a : H = q/2\rangle\}, \\ \mathcal{T}_0 &:= \{\langle H \overset{r_i}{\rightsquigarrow} H\rangle \mid 1 \leq i \leq m\}.\end{aligned}$$

The concept name H will later be used to detect whether $\mathcal{P}$ has a solution (see the proof of Theorem 6).

Let now $\mathcal{O}_{\mathcal{P}} := (\mathcal{A}_{\mathcal{P}}, \mathcal{T}_{\mathcal{P}})$ where $\mathcal{A}_{\mathcal{P}} = \mathcal{A}_{\mathcal{P}}^0 \cup \mathcal{A}_0$ and $\mathcal{T}_{\mathcal{P}} := \mathcal{T}_0 \cup \bigcup_{i=0}^m \mathcal{T}_{\mathcal{P}}^i$. We define the interpretation $\mathcal{I}_{\mathcal{P}} := (\Delta^{\mathcal{I}_{\mathcal{P}}}, \cdot^{\mathcal{I}_{\mathcal{P}}})$ as follows:

- $\Delta^{\mathcal{I}_{\mathcal{P}}} = \{1, \ldots, m\}^*$,
- $a^{\mathcal{I}_{\mathcal{P}}} = \varepsilon$,

for every $\mu \in \Delta^{\mathcal{I}_{\mathcal{P}}}$,

- $A^{\mathcal{I}_{\mathcal{P}}}(\mu) = q \cdot 2^{-v_\mu}$, $B^{\mathcal{I}_{\mathcal{P}}}(\mu) = q \cdot 2^{-w_\mu}$, $H^{\mathcal{I}_{\mathcal{P}}}(\mu) = q/2$,

and for all $j, 1 \leq j \leq m$,

- $V_j^{\mathcal{I}_{\mathcal{P}}}(\mu) = q \cdot 2^{-v_j}$, $W_j^{\mathcal{I}_{\mathcal{P}}}(\mu) = q \cdot 2^{-w_j}$, and
- $r_j^{\mathcal{I}_{\mathcal{P}}}(\mu, \mu j) = 1$ and $r_j^{\mathcal{I}_{\mathcal{P}}}(\mu, \mu') = 0$ if $\mu' \neq \mu j$.

It is easy to see that $\mathcal{I}_{\mathcal{P}}$ is in fact a model of $\mathcal{O}_{\mathcal{P}}$. This model is trivially witnessed since, for every $i, 1 \leq i \leq m$, every node has only one r_i successor with degree greater than 0. More interesting, however, is that every witnessed model $\mathcal{I}$ of $\mathcal{O}_{\mathcal{P}}$ "contains" $\mathcal{I}_{\mathcal{P}}$ in the sense stated in the following lemma.

Lemma 5. *Let $\mathcal{I}$ be a witnessed model of $\mathcal{O}_\mathcal{P}$. Then there exists a function $f : \Delta^{\mathcal{I}_\mathcal{P}} \to \Delta^\mathcal{I}$ such that, for every $\mu \in \Delta^{\mathcal{I}_\mathcal{P}}$, $C^{\mathcal{I}_\mathcal{P}}(\mu) = C^\mathcal{I}(f(\mu))$ holds for every concept name C and $r_i^\mathcal{I}(f(\mu), f(\mu i)) = 1$ holds for every $i, 1 \leq i \leq m$.*

Proof. The function f is built inductively on the length of μ. First, as $\mathcal{I}$ is a model of $\mathcal{A}_\mathcal{P}$, there must be a $\delta \in \Delta^\mathcal{I}$ such that $a^\mathcal{I} = \delta$. Notice that $\mathcal{A}_\mathcal{P}$ fixes the interpretation of all concept names on δ and hence $f(\varepsilon) = \delta$ satisfies the condition of the lemma.

Let now μ be such that $f(\mu)$ has already been defined. By induction, we can assume that $A^\mathcal{I}(f(\mu)) = q \cdot 2^{-v_\mu}, B^\mathcal{I}(f(\mu)) = q \cdot 2^{-w_\mu}, H^\mathcal{I}(f(\mu)) = q/2$, and for every j, $1 \leq j \leq m$, $V_j^\mathcal{I}(f(\mu)) = q \cdot 2^{-v_j}, W_j^\mathcal{I}(f(\mu)) = q \cdot 2^{-w_j}$. Since $\mathcal{I}$ is a witnessed model of $\langle \top \sqsubseteq \exists r_i.\top \geq 1 \rangle$, for all $i, 1 \leq i \leq m$, there exists a $\gamma \in \Delta^\mathcal{I}$ with $r_i^\mathcal{I}(f(\mu), \gamma) = 1$, and as $\mathcal{I}$ satisfies all the axioms of the form $\langle C \overset{r}{\rightsquigarrow} D \rangle$ in $\mathcal{T}_\mathcal{P}$, it follows that

$$A^\mathcal{I}(\gamma) = q \cdot 2^{-v_\mu v_i} = q \cdot 2^{-v_{\mu i}}, \quad B^\mathcal{I}(\gamma) = q \cdot 2^{-w_\mu w_i} = q \cdot 2^{-w_{\mu i}}, \quad H^\mathcal{I}(\gamma) = q/2,$$

and for all $j, 1 \leq j \leq m$, $V_j^\mathcal{I}(\gamma) = q \cdot 2^{-v_j}, W_j^\mathcal{I}(\gamma) = q \cdot 2^{-w_j}$. Setting $f(\mu i) = \gamma$ thus satisfies the required property. □

From this lemma it then follows that, if the PCP $\mathcal{P}$ has a solution μ for some $\mu \in \{1, \ldots, m\}^+$, then every witnessed model $\mathcal{I}$ of $\mathcal{O}_\mathcal{P}$ contains a node $\delta = f(\mu)$ such that $A^\mathcal{I}(\delta) = B^\mathcal{I}(\delta)$; i.e., where A and B encode the same word. Conversely, if every witnessed model contains such a node, then in particular $\mathcal{I}_\mathcal{P}$ does, and thus $\mathcal{P}$ has a solution. The question is now how to detect whether a node with this characteristics exists in every model. We will extend $\mathcal{O}_\mathcal{P}$ with axioms that further restrict $\mathcal{I}_\mathcal{P}$ to satisfy $A^{\mathcal{I}_\mathcal{P}}(\mu) \neq B^{\mathcal{I}_\mathcal{P}}(\mu)$ for every $\mu \in \{1, \ldots, m\}^+$. This ensures that the extended ontology has a model iff $\mathcal{P}$ has *no* solution.

In order to come up with the right axioms for achieving this, suppose for now that, for some $\mu \in \{1, \ldots, m\}^*$, it holds that

$$q \cdot 2^{-v_\mu} = A^{\mathcal{I}_\mathcal{P}}(\mu) > B^{\mathcal{I}_\mathcal{P}}(\mu) = q \cdot 2^{-w_\mu}.$$

We then have that $v_\mu < w_\mu$ and hence $w_\mu - v_\mu \geq 1$. It thus follows that

$$(A \to B)^{\mathcal{I}_\mathcal{P}}(\mu) = q \cdot (q \cdot 2^{-w_\mu})/(q \cdot 2^{-v_\mu}) = q \cdot 2^{-(w_\mu - v_\mu)} \leq q \cdot 2^{-1} = q/2$$

and thus $((A \to B) \sqcap (B \to A))^{\mathcal{I}_\mathcal{P}}(\mu) \leq q/2$. Likewise, if $A^{\mathcal{I}_\mathcal{P}}(\mu) < B^{\mathcal{I}_\mathcal{P}}(\mu)$, we also get $((A \to B) \sqcap (B \to A))^{\mathcal{I}_\mathcal{P}}(\mu) \leq q/2$. Additionally, if $A^{\mathcal{I}_\mathcal{P}}(\mu) = B^{\mathcal{I}_\mathcal{P}}(\mu)$, then it is easy to verify (see Lemma 1) that $((A \to B) \sqcap (B \to A))^{\mathcal{I}_\mathcal{P}}(\mu) = 1$. From all this it follows that, for every $\mu \in \{1, \ldots, m\}^*$,

$$A^{\mathcal{I}_\mathcal{P}}(\mu) \neq B^{\mathcal{I}_\mathcal{P}}(\mu) \quad \text{iff} \quad ((A \to B) \sqcap (B \to A))^{\mathcal{I}_\mathcal{P}}(\mu) \leq q/2. \tag{1}$$

Thus, the instance $\mathcal{P}$ has no solution iff for every $\mu \in \{1, \ldots, m\}^+$ it holds that $((A \to B) \sqcap (B \to A))^{\mathcal{I}_\mathcal{P}}(\mu) \leq q/2$.

We define now the ontology $\mathcal{O}'_\mathcal{P} := (\mathcal{A}_\mathcal{P}, \mathcal{T}'_\mathcal{P})$ where

$$\mathcal{T}'_\mathcal{P} := \mathcal{T}_\mathcal{P} \cup \{\langle \top \sqsubseteq \forall r_i.(((A \to B) \sqcap (B \to A)) \to H) \geq 1 \rangle \mid 1 \leq i \leq m\}.$$

Theorem 6. *The instance $\mathcal{P}$ of the PCP has a solution iff the ontology $\mathcal{O}'_{\mathcal{P}}$ is not witnessed consistent.*

Proof. Assume first that $\mathcal{P}$ has a solution $\mu = i_1 \cdots i_k$ and let $u = v_\mu = w_\mu$ and $\mu' = i_1 i_2 \cdots i_{k-1} \in \{1, \ldots, m\}^*$. Suppose there is a witnessed model $\mathcal{I}$ of $\mathcal{O}'_{\mathcal{P}}$. Since $\mathcal{O}_{\mathcal{P}} \subseteq \mathcal{O}'_{\mathcal{P}}$, $\mathcal{I}$ must also be a model of $\mathcal{O}_{\mathcal{P}}$. From Lemma 5 it then follows that there are nodes $\delta, \delta' \in \Delta^{\mathcal{I}}$ such that $A^{\mathcal{I}}(\delta) = A^{\mathcal{I}_{\mathcal{P}}}(\mu) = B^{\mathcal{I}_{\mathcal{P}}}(\mu) = B^{\mathcal{I}}(\delta)$, $H^{\mathcal{I}}(\delta) = H^{\mathcal{I}_{\mathcal{P}}}(\mu) = q/2$, and $r_{i_k}^{\mathcal{I}}(\delta', \delta) = 1$. Then we have $((A \to B) \sqcap (B \to A))^{\mathcal{I}}(\delta) = 1$, and hence

$$(((A \to B) \sqcap (B \to A)) \to H)^{\mathcal{I}}(\delta) = 1 \Rightarrow q/2 = q/2.$$

This then means that $(\forall r_{i_k}.(((A \to B) \sqcap (B \to A)) \to H))^{\mathcal{I}}(\delta') \leq q/2$, violating one of the axioms in $\mathcal{T}'_{\mathcal{P}} \setminus \mathcal{T}_{\mathcal{P}}$. Hence, $\mathcal{I}$ is cannot be a model of $\mathcal{O}'_{\mathcal{P}}$.

Conversely, assume that $\mathcal{O}'_{\mathcal{P}}$ is not witnessed consistent. Then $\mathcal{I}_{\mathcal{P}}$ is not a model of $\mathcal{O}'_{\mathcal{P}}$. Since it is a model of $\mathcal{O}_{\mathcal{P}}$, there must exist an $i, 1 \leq i \leq m$ such that $\mathcal{I}_{\mathcal{P}}$ violates the axiom $\langle \top \sqsubseteq \forall r_i.(((A \to B) \sqcap (B \to A)) \to H) \geq 1 \rangle$. This means that there is some $\mu \in \{1, \ldots, m\}^*$ such that

$$(\forall r_i.(((A \to B) \sqcap (B \to A)) \to H))^{\mathcal{I}_{\mathcal{P}}}(\mu) < 1.$$

Since $r_i^{\mathcal{I}_{\mathcal{P}}}(\mu, \mu') = 0$ for all $\mu' \neq \mu i$ and $r_i^{\mathcal{I}_{\mathcal{P}}}(\mu, \mu i) = 1$, this implies that

$$(((A \to B) \sqcap (B \to A)) \to H)^{\mathcal{I}_{\mathcal{P}}}(\mu i) < 1;$$

i.e. $((A \to B) \sqcap (B \to A))^{\mathcal{I}_{\mathcal{P}}}(\mu i) > q/2$. From the equivalence (1) above, it follows that $A^{\mathcal{I}_{\mathcal{P}}}(\mu i) = B^{\mathcal{I}_{\mathcal{P}}}(\mu i)$, and hence μi is a solution of $\mathcal{P}$. □

Corollary 7. *Witnessed consistency of $*$-$\mathcal{ALE}$ ontologies is undecidable if conjunction is interpreted using a t-norm that q-starts with Π for a rational number $q \in (0, 1]$.*

Notice that in the proofs of Lemma 5 and Theorem 6, the second condition of the definition of witnessed models was never used. Moreover, the witnessed interpretation $\mathcal{I}_{\mathcal{P}}$ is also weakly witnessed. We thus have the following corollary.

Corollary 8. *Weakly witnessed consistency and quasi-witnessed consistency of $*$-$\mathcal{ALE}$ ontologies are undecidable if conjunction is interpreted using a t-norm that q-starts with Π for a rational number $q \in (0, 1]$.*

5 Undecidability w.r.t. Strongly Witnessed Models

Unfortunately, the model $\mathcal{I}_{\mathcal{P}}$ constructed in the previous section is not a strongly witnessed model of $\mathcal{O}_{\mathcal{P}}$ since, for instance, $\inf_{\eta \in \Delta^{\mathcal{I}_{\mathcal{P}}}} (\top^{\mathcal{I}_{\mathcal{P}}}(\eta) \Rightarrow A^{\mathcal{I}_{\mathcal{P}}}(\eta)) = 0$, but there is no $\delta \in \Delta^{\mathcal{I}_{\mathcal{P}}}$ with $A^{\mathcal{I}_{\mathcal{P}}}(\delta) = 0$. Thus, the construction of $\mathcal{O}'_{\mathcal{P}}$ does not yield an undecidability result for strongly witnessed consistency in $*$-$\mathcal{ALE}$.

This means that we need a different reduction to prove undecidability of strongly witnessed consistency. This reduction will follow a similar idea to the

one from the previous section, in which models describe a search for a solution of the PCP $\mathcal{P}$. However, rather than building the whole search tree, models will describe only individual branches of this tree. The condition **(wit3)** will help ensure that at some point in this branch a solution is found. Conversely, the models constructed from solutions will be finite, and thus trivially strongly witnessed.

Before describing the reduction in detail, we recall a useful property of t-norms. Using a t-norm $\otimes$ and its associated residuum $\Rightarrow$, one can express the minimum and maximum operators as follows [15]:

- $\min(x, y) = x \otimes (x \Rightarrow y)$,
- $\max(x, y) = \min(((x \Rightarrow y) \Rightarrow y), ((y \Rightarrow x) \Rightarrow x))$.

We can thus introduce w.l.o.g. the $*$-$\mathcal{ALE}$ concept constructor max with the obvious semantics. We will use this constructor to simulate the non-deterministic choices in the search tree as described next.

Given an instance $\mathcal{P} = ((v_1, w_1), \ldots, (v_m, w_m))$ of the PCP, we define the ABox $\mathcal{A}^0_\mathcal{P}$ and the TBox $\mathcal{T}^0_\mathcal{P}$ as in the previous section, and for every $i, 1 \leq i \leq m$, we construct the TBox

$$\mathcal{T}^{s\,i}_\mathcal{P} := \{\langle C_i \sqsubseteq \exists r_i.\top \geq 1\rangle, \langle V_i \sqcap A^{(s+1)^{|v_i|}} \overset{r_i}{\rightsquigarrow} A\rangle, \langle W_i \sqcap B^{(s+1)^{|w_i|}} \overset{r_i}{\rightsquigarrow} B\rangle\}.$$

The only difference between the TBoxes $\mathcal{T}^i_\mathcal{P}$ and $\mathcal{T}^{s\,i}_\mathcal{P}$ is in the first axiom. Intuitively, the concept names C_i encode the choice of the branch in the tree to be expanded. Only if $C_i^\mathcal{I}(\delta) = 1$, there will be an r_i successor with degree 1, and the i-th branch of the tree will be explored. For this intuition to work, we need to ensure that at least one of the C_is is interpreted as 1 in every node. On the other hand, we can stop expanding the tree once a solution has been found. Using this intuition, we define the ontology $\mathcal{O}^s_\mathcal{P} := (\mathcal{A}^s_\mathcal{P}, \mathcal{T}^s_\mathcal{P})$ where

$$\mathcal{A}^s_\mathcal{P} := \mathcal{A}^0_\mathcal{P} \cup \{a : \max(C_1, \ldots, C_m) = 1\},$$
$$\mathcal{T}^s_\mathcal{P} := \mathcal{T}^0_\mathcal{P} \cup \bigcup_{i=1}^{m} \mathcal{T}^{s\,i}_\mathcal{P} \cup \{\langle (A \sqcap B) \rightarrow \bot \sqsubseteq \bot \geq 1\rangle\} \cup$$
$$\{\langle \top \sqsubseteq \forall r_i.\max((A \rightarrow B) \sqcap (B \rightarrow A), C_1, \ldots, C_m) \geq 1\rangle \mid 1 \leq i \leq m\}.$$

Theorem 9. *The instance $\mathcal{P}$ of the PCP has a solution iff the ontology $\mathcal{O}^s_\mathcal{P}$ is strongly witnessed consistent.*

Proof. Let $\nu = i_1 i_2 \cdots i_k$ be a solution of $\mathcal{P}$ and let $\mathsf{pre}(\nu)$ denote the set of all prefixes of ν. We build the finite interpretation $\mathcal{I}^s_\mathcal{P}$ as follows:

- $\Delta^{\mathcal{I}^s_\mathcal{P}} := \mathsf{pre}(\nu)$,
- $a^{\mathcal{I}^s_\mathcal{P}} = \varepsilon$,

for all $\mu \in \Delta^{\mathcal{I}^s_\mathcal{P}}$,

- $A^{\mathcal{I}^s_\mathcal{P}}(\mu) = q \cdot 2^{-v_\mu}$, $B^{\mathcal{I}^s_\mathcal{P}}(\mu) = q \cdot 2^{-w_\mu}$,

and for all $j, 1 \leq j \leq m$

- $V_j^{\mathcal{I}^s_\mathcal{P}}(\mu) = q \cdot 2^{-v_j}$, $W_j^{\mathcal{I}^s_\mathcal{P}}(\mu) = q \cdot 2^{-w_j}$,
- $C_j^{\mathcal{I}^s_\mathcal{P}}(\mu) = 1$ if $\mu j \in \mathsf{pre}(\nu)$ and $C_j^{\mathcal{I}^s_\mathcal{P}}(\mu) = 0$ otherwise, and
- $r_j^{\mathcal{I}^s_\mathcal{P}}(\mu, \mu j) = 1$ if $\mu j \in \mathsf{pre}(\nu)$ and $r_j^{\mathcal{I}^s_\mathcal{P}}(\mu, \mu') = 0$ if $\mu' \in \mathsf{pre}(\nu)$ and $\mu' \neq \mu j$.

We show now that $\mathcal{I}^s_\mathcal{P}$ is a model of $\mathcal{O}^s_\mathcal{P}$. Since $\mathcal{I}^s_\mathcal{P}$ is finite, it follows immediately that it is also strongly witnessed. Clearly $\mathcal{I}^s_\mathcal{P}$ satisfies all axioms in $\mathcal{A}^0_\mathcal{P}$; additionally, we have that $C_{i_1}^{\mathcal{I}^s_\mathcal{P}}(\varepsilon) = 1$ and thus, $\mathcal{I}^s_\mathcal{P}$ satisfies $\mathcal{A}^s_\mathcal{P}$. The axiom $\langle (A \sqcap B) \to \bot \sqsubseteq \bot \geq 1 \rangle$ expresses that $(A \sqcap B)^{\mathcal{I}^s_\mathcal{P}}(\mu) \Rightarrow 0 = 0$, and hence $(A \sqcap B)^{\mathcal{I}^s_\mathcal{P}}(\mu) > 0$ for all $\mu \in \mathsf{pre}(\nu)$, which clearly holds. We now show that the other axioms are also satisfied for every $\mu \in \mathsf{pre}(\nu)$.

Let $\mu \in \mathsf{pre}(\nu) \setminus \{\nu\}$. Then we know that there exists $i, 1 \leq i \leq m$, such that $C_i^{\mathcal{I}^s_\mathcal{P}}(\mu) = 1$ and $r_i^{\mathcal{I}^s_\mathcal{P}}(\mu, \mu i) = 1$; thus μ satisfies the axioms in $\mathcal{T}^{si}_\mathcal{P}$. Moreover, $C_j^{\mathcal{I}^s_\mathcal{P}}(\mu) = 0 = r_j^{\mathcal{I}^s_\mathcal{P}}(\mu, \mu')$ for all $j \neq i$ and all $\mu' \in \mathsf{pre}(\nu)$, which means that μ trivially satisfies all axioms in $\mathcal{T}^{sj}_\mathcal{P}$. If $\mu i = \nu$, then $((A \to B) \sqcap (B \to A))^{\mathcal{I}^s_\mathcal{P}}(\mu i) = 1$ since ν is a solution. Otherwise, there is a $j, 1 \leq j \leq m$ with $\mu i j \in \mathsf{pre}(\nu)$, and thus $C_j^{\mathcal{I}^s_\mathcal{P}}(\mu i) = 1$. Thus, we have in both cases that μ also satisfies the last axioms in $\mathcal{T}^s_\mathcal{P}$.

Finally, if $\mu = \nu$, then $r_i^{\mathcal{I}^s_\mathcal{P}}(\mu, \mu') = 0$ and $C_i(\mu) = 0$, for all $\mu' \in \mathsf{pre}(\nu)$ and all $i, 1 \leq i \leq m$, and thus the axioms are all trivially satisfied.

Conversely, let $\mathcal{I}$ be a strongly witnessed model of $\mathcal{O}^s_\mathcal{P}$. Then, there must be an element $\delta_0 \in \Delta^\mathcal{I}$ with $a^\mathcal{I} = \delta_0$. Since $\mathcal{I}$ must satisfy all axioms in $\mathcal{A}^s_\mathcal{P}$, there is an $i_1, 1 \leq i_1 \leq m$ such that $C_{i_1}^\mathcal{I}(\delta_0) = 1$. Since δ_0 must satisfy the axioms in $\mathcal{T}^{si_1}_\mathcal{P}$, there must exist a $\delta_1 \in \Delta^\mathcal{I}$ with $r_{i_1}^\mathcal{I}(\delta_0, \delta_1) = 1$, $A^\mathcal{I}(\delta_1) = q \cdot 2^{-v_{i_1}}$, and $B^\mathcal{I}(\delta_1) = q \cdot 2^{-w_{i_1}}$. If $A^\mathcal{I}(\delta_1) = B^\mathcal{I}(\delta_1)$, then i_1 is a solution of $\mathcal{P}$. Otherwise, from the last set of axioms in $\mathcal{T}^s_\mathcal{P}$, there must exist an $i_2, 1 \leq i_2 \leq m$ with $C_{i_2}^\mathcal{I}(\delta_1) = 1$. We can then iterate this construction to generate a sequence $i_3, i_4, \ldots$ of indices and $\delta_2, \delta_3, \ldots \in \Delta^\mathcal{I}$ where $A^\mathcal{I}(\delta_k) = q \cdot 2^{-v_{i_1} \cdots v_{i_k}}$, and $B^\mathcal{I}(\delta_k) = q \cdot 2^{-w_{i_1} \cdots w_{i_k}}$.

If there is some k such that $A^\mathcal{I}(\delta_k) = B^\mathcal{I}(\delta_k)$, then $i_1 \cdots i_k$ is a solution of $\mathcal{P}$. Assume now that no such k exists. We then have an infinite sequence of indices $i_1, i_2, \ldots$ and since, for every $i, 1 \leq i \leq m$, either $v_i \neq 0$ or $w_i \neq 0$, then at least one of the sequences $v_{i_1} \cdots v_{i_k}, w_{i_1} \cdots w_{i_k}$ increases as k gets larger. Thus, for every natural number n there is a k such that either $v_{i_1} \cdots v_{i_k} > n$ or $w_{i_1} \cdots w_{i_k} > n$; consequently $(A \sqcap B)^\mathcal{I}(\delta_k) < q \cdot 2^{-n}$. This implies that

$$\inf_{\eta \in \Delta^\mathcal{I}} (\top^\mathcal{I}(\eta) \Rightarrow (A \sqcap B)^\mathcal{I}(\eta)) = 0,$$

and since $\mathcal{I}$ is strongly witnessed, there must exist a $\gamma \in \Delta^\mathcal{I}$ with

$$0 = \top^\mathcal{I}(\gamma) \Rightarrow (A \sqcap B)^\mathcal{I}(\gamma) = (A \sqcap B)^\mathcal{I}(\gamma).$$

But from this it follows that $((A \sqcap B) \to \bot)^\mathcal{I}(\gamma) \Rightarrow 0 = 0$, contradicting the axiom $\langle (A \sqcap B) \to \bot \sqsubseteq \bot \geq 1 \rangle$ of $\mathcal{T}^s_\mathcal{P}$. Thus, $\mathcal{P}$ has a solution. □

Notice that, if $\mathcal{P}$ has no solution, then $\mathcal{O}^s_\mathcal{P}$ still has witnessed models, but no strongly witnessed models. It is also relevant to point out that $\mathcal{O}^s_\mathcal{P}$ has a strongly

witnessed model iff it has a finite model. In fact, the condition of strongly witnessed was only used for ensuring finiteness of the model, and hence, that a solution is indeed found.

Corollary 10. *For $*$-$\mathcal{ALE}$ ontologies, strongly witnessed consistency and consistency w.r.t. finite models are undecidable if conjunction is interpreted using a t-norm that q-starts with Π for a rational number $q \in (0, 1]$.*

6 Conclusions

We have shown that consistency of $*$-$\mathcal{ALE}$ ontologies w.r.t. several notions of models, ranging from finite models to weakly witnessed models, is undecidable if the t-norm used to interpret conjunction is a t-norm that q-starts with Π for a rational number $q \in (0, 1]$. Since, for every $q \in (0, 1]$, there exist uncountably many t-norms that q-start with Π, our results yield an uncountable family of t-norms for which reasoning in $*$-$\mathcal{ALE}$ becomes undecidable. Whether consistency in general (i.e., without restricting the class of interpretations) is also undecidable under these t-norms is still an open problem. The same is true if a t-norm that does not q-start with Π for a rational number $q \in (0, 1]$ is used. For the case of fuzzy DLs where disjunction and involutive negation is used in place of the residuum, we have an undecidability results for the product t-norm, but only for the case of witnessed models and with an extension of the TBox formalism to allow for the use of $>$ in fuzzy GCIs [2].

Since the results in [5,2] have shown that the tableau-based algorithms for fuzzy DLs with GCIs are actually incorrect, the only decidability results for fuzzy DLs with GCIs that are currently available are those that use a finite set of fuzzy membership degrees [11,12,10], or consider a rather simple t-norm (e.g. the Gödel t-norm) over the interval $[0, 1]$, where only finitely many membership degrees are relevant for reasoning [6]. In these cases, a black-box approach that calls a crisp DL reasoner can be used.

References

1. Baader, F., Calvanese, D., McGuinness, D., Nardi, D., Patel-Schneider, P.F. (eds.): The Description Logic Handbook: Theory, Implementation, and Applications. Cambridge University Press, Cambridge (2003)
2. Baader, F., Peñaloza, R.: Are fuzzy description logics with general concept inclusion axioms decidable? In: Proc. of Fuzz-IEEE 2011, pp. 1735–1742. IEEE Press, Los Alamitos (2011)
3. Baader, F., Sattler, U.: An overview of tableau algorithms for description logics. Studia Logica 69, 5–40 (2001)
4. Bobillo, F., Bou, F., Straccia, U.: On the failure of the finite model property in some fuzzy description logics. CoRR, abs/1003.1588 (2010)
5. Bobillo, F., Bou, F., Straccia, U.: On the failure of the finite model property in some fuzzy description logics. Fuzzy Sets and Systems 172(23), 1–12 (2011)

6. Bobillo, F., Delgado, M., Gómez-Romero, J., Straccia, U.: Fuzzy description logics under Gödel semantics. Int. J. of Approx. Reas. 50(3), 494–514 (2009)
7. Bobillo, F., Straccia, U.: A fuzzy description logic with product t-norm. In: Proc. of Fuzz-IEEE 2007, pp. 1–6. IEEE, Los Alamitos (2007)
8. Bobillo, F., Straccia, U.: On qualified cardinality restrictions in fuzzy description logics under Łukasiewicz semantics. In: Proc. of IPMU 2008, pp. 1008–1015 (2008)
9. Bobillo, F., Straccia, U.: Fuzzy description logics with general t-norms and datatypes. Fuzzy Sets and Systems 160(23), 3382–3402 (2009)
10. Bobillo, F., Straccia, U.: Reasoning with the finitely many-valued Łukasiewicz fuzzy description logic $\mathcal{SROIQ}$. Information Sciences 181, 758–778 (2011)
11. Borgwardt, S., Peñaloza, R.: Description logics over lattices with multi-valued ontologies. In: Proc. of IJCAI 2011 (to appear, 2011)
12. Borgwardt, S., Peñaloza, R.: Fuzzy ontologies over lattices with t-norms. In: Proc. of DL 2011, Barcelona, Spain. CEUR-WS, vol. 745 (2011)
13. Cerami, M., Esteva, F., Bou, F.: Decidability of a description logic over infinite-valued product logic. In: Proc. of KR 2010. AAAI Press, Menlo Park (2010)
14. García-Cerdaña, A., Armengol, E., Esteva, F.: Fuzzy description logics and t-norm based fuzzy logics. Int. J. of Approx. Reas. 51, 632–655 (2010)
15. Hájek, P.: Metamathematics of Fuzzy Logic Trends in Logic. Springer, Heidelberg (2001)
16. Hájek, P.: Making fuzzy description logic more general. Fuzzy Sets and Systems 154(1), 1–15 (2005)
17. Horrocks, I., Patel-Schneider, P.F., van Harmelen, F.: From SHIQ and RDF to OWL: The making of a web ontology language. Journal of Web Semantics 1(1), 7–26 (2003)
18. Lukasiewicz, T., Straccia, U.: Managing uncertainty and vagueness in description logics for the semantic web. Journal of Web Semantics 6(4), 291–308 (2008)
19. Molitor, R., Tresp, C.B.: Extending description logics to vague knowledge in medicine. In: Szczepaniak, P., Lisboa, P., Tsumoto, S. (eds.) Fuzzy Systems in Medicine. Studies in Fuzziness and Soft Computing, vol. 41, pp. 617–635. Springer, Heidelberg (2000)
20. Mostert, P.S., Shields, A.L.: On the structure of semigroups on a compact manifold with boundary. Annals of Mathematics 65, 117–143 (1957)
21. Post, E.: A variant of a recursively unsolvable problem. Bulletin of the American Mathematical Society 52, 264–268 (1946)
22. Stoilos, G., Stamou, G.B., Tzouvaras, V., Pan, J.Z., Horrocks, I.: The fuzzy description logic f-$\mathcal{SHIN}$. In: Proc. of URSW 2005, pp. 67–76 (2005)
23. Straccia, U.: Reasoning within fuzzy description logics. JAIR 14, 137–166 (2001)
24. Straccia, U.: Description logics with fuzzy concrete domains. In: Proc. of UAI 2005, pp. 559–567. AUAI Press (2005)
25. Straccia, U., Bobillo, F.: Mixed integer programming, general concept inclusions and fuzzy description logics. In: Proc. of EUSFLAT 2007, pp. 213–220. Universitas Ostraviensis (2007)
26. Tresp, C., Molitor, R.: A description logic for vague knowledge. In: Proc. of ECAI 1998, pp. 361–365. J. Wiley and Sons, Chichester (1998)

The Complexity of Reversal-Bounded Model-Checking*

Marcello M. Bersani[1,2] and Stéphane Demri[1]

[1] LSV, ENS Cachan, CNRS, INRIA, France
[2] Politecnico di Milano, Italy

Abstract. We study model-checking problems on counter systems when guards are quantifier-free Presburger formulae, the specification languages are LTL-like dialects with arithmetical constraints and the runs are restricted to reversal-bounded ones. We introduce a generalization of reversal-boundedness and we show the NExpTime-completeness of the reversal-bounded model-checking problem as well as for related reversal-bounded reachability problems. As a by-product, we show the effective Presburger definability for sets of configurations for which there is a reversal-bounded run verifying a given temporal formula. Our results generalize existing results about reversal-bounded counter automata and provides a uniform and more general framework.

1 Introduction

Reversal-Bounded Model-Checking. Given a counter system $\mathcal{S}$ and a linear-time property ϕ expressed in a logical formalism, a standard question in formal verification is to determine whether there is an infinite run ρ for $\mathcal{S}$ satisfying ϕ (written $\rho \models \phi$), or dually whether all runs satisfy ϕ. In full generality, existential model-checking problem is undecidable (as an immediate consequence of the undecidability of the halting problem for Minsky machines). A way to overcome this difficulty is to consider a subclass of runs for $\mathcal{S}$ for which decidability is regained; in that case, we answer a different question but in case of positive answer, starting from a subclass of runs does not harm. In the paper, we restrict the runs so that along an infinite run, for each counter the number of reversals is bounded by a given value r; a reversal is witnessed when a counter behaviour changes from increasing mode to decreasing mode, or vice-versa. We follow an approach similar to bounded model-checking (BMC), see e.g. [6], in which runs are built until positions of a bounded distance from the initial configuration. Analogously, in context-bounded model-checking, the number of segments of the computation during which only one thread is active is bounded in multithreaded programs, see e.g. [29]. As for bounded model-checking, in case of negative answer to the question, the value r can be incrementally augmented. Reversal-bounded counter systems have been first studied in [20] and several extensions have been considered in the literature, see e.g. [13]. A major property

* Work supported by Agence Nationale de la Recherche, grant ANR-06-SETIN-001 and by the European Commission, Project 227977-SMScom.

C. Tinelli and V. Sofronie-Stokkermans (Eds.): FroCoS 2011, LNAI 6989, pp. 71–86, 2011.

of such systems is that the sets of configurations reachable from a given initial configuration, are effectively Presburger-definable. However this does not entail that problems involving infinite runs are decidable, since infinite runs are not necessarily effectively representable in Presburger arithmetic, see e.g. [10]. In this paper, we study problems of the form: given a counter system $\mathcal{S}$, a bound $r \geq 0$ and a formula ϕ, is there an infinite r-reversal-bounded run ρ such that $\rho \models \phi$. To complete our analogy, it is fair to observe that BMC for finite systems benefits from nice properties on runs that allow the existence of an upper bound on the length of runs to be checked (a.k.a. *completeness threshold*). That is why, a finite amount of BMC instances needs to be checked in order to provide an answer to any instance of the model-checking problem. By contrast, since the reversal-boundedness detection problem on counter systems is undecidable [20], there is no guarantee that given an initialized counter system, there exists a r-reversal-bounded run, for some $r \geq 0$, satisfying a given temporal property.

Our Motivations. In order to test whether there is an infinite run satisfying a temporal property, we restrict ourselves to r-reversal-bounded runs for some $r \geq 0$ so that for a fixed r, the problem is decidable. In case of positive answer, we stop the process, otherwise we increment r and perform again a test. This incremental approximation approach is applied to counter systems that are more general than Minsky machines (counter automata with increments, decrements and zero-tests), typically by considering guards definable in quantifier-free fragment of Presburger arithmetic and update vectors in $\mathbb{Z}^n$. Moreover, we aim at expressing the temporal property in a rich LTL-like dialect, including arithmetical constraints and past-time operators (i.e., not only restricted to Boolean combinations of **GF**-formulae). Finally, not only we characterize the computational complexity of the existence of r-reversal-bounded runs but also our goal is to effectively express the set of configurations admitting such runs in Presburger arithmetic, which will allow us to use SMT solvers to perform verification tasks on counter systems (see e.g. [2,26]) or to take advantage of verification techniques developed in [5]. It is worth noting that the use of Presburger arithmetic for formal verification has been already advocated since the work [31].

Our Contributions. As far as the methodology is concerned, we reduce model-checking problems to reachability problems (first, by synchronization of the counter system and the automaton representing the temporal formula and, then, we reduce the model-checking problems to reachability problems). Let us quote the major results of the paper. (i) We introduce a new concept for reversal-boundedness that makes explicit the role of arithmetical terms and it captures previous notions on reversal-boundedness (see Section 2). (ii) We show that the reversal-bounded model-checking problem for counter systems with guards in QFP (quantifier-free fragment of Presburger arithmetic) and temporal formulae with atomic formulae in QFP is decidable and NExpTime-complete (see Theo. 4). The same complexity applies to reversal-bounded control state repeated reachability problem and reversal-bounded reachability problem (see Corollary 6). (iii) We show that the existence of reversal-bounded runs satisfying a temporal property implies the existence of reversal-bounded runs that are

ultimately periodic, i.e. the sequences of transitions are of the form $\pi_1 \cdot (\pi_2)^\omega$ where π_1 and π_2 are finite sequences. This type of properties has been already shown useful to implement verification methods following the BMC paradigm (see Corollary 3). (iv) Besides, our complexity results provide as by-products that reachability sets for reversal-bounded counter systems are effectively Presburger definable (see Corollary 4) and sets of configurations for which there is a reversal-bounded run verifying a temporal formula are also effectively Presburger definable (Theorem 5).

Related Works. Effective Presburger definability for reversal-bounded Minsky machines and more generally for reversal-bounded counter systems can be found in [20,19,13] whereas the NExpTime-completeness of the reversal-bounded reachability problem for Minsky machines has been shown in [17] (lower bound) and [14,17] (upper bound). The NExpTime upper bounds established in this paper for several extensions with richer classes of counter systems or with richer concepts of reversal-boundedness build on [14] and on [30] with adaptations to handle more complicated technical features. Decidability results for reversal-bounded counter systems augmented with familiar data structures such as stacks or queues (also with restricted behaviours) can be found in [18]. Our temporal language is very expressive and includes control states as well as arithmetical constraints in QFP. Moreover, in the paper we deal with model-checking involving linear-time temporal logics with past-time and future temporal operators and with arithmetical constraints on counter values. In [10], it is shown that $\exists$-Presburger-infinitely often problem for reversal-bounded counter automata (with guards made of Boolean combinations of the form $\mathtt{x}_i \sim k$) is decidable. Moreover, $\exists$-Presburger-always problem for reversal-bounded counter automata is undecidable [10]. Our decidability results on model-checking refine these results in order to obtain new decidability results, by allowing a full LTL specification language with arithmetical constraints and by proposing new concepts for reversal-boundedness. Finally, in [21, Theorem 22], ExpTime upper bound for LTL model-checking over reversal-bounded counter automata is shown but the logical language has no arithmetical constraint and the number of reversals r is encoded in unary (see also [32]). In our setting, our complexity results deal with instances in which all the integers are encoded in binary.

The recent work [16] is also closely related to our paper. We are grateful to an anonymous referee for pointing it to us. Our work and [16] have been done independently but most of our results can be reproved by extending [16]. In [16], operational models extending pushdown systems with counters and clocks are considered; a version of reversal-bounded LTL model-checking is shown to be co-NExpTime [16, Theorem 2]. A prototypical implementation and experimental results are also presented in [16]. LTL dialect contains only control states and guards are Boolean combinations of constraints of the form $\mathtt{x} \sim k$. By contrast, models are more general than ours. Theorem 2 in [16] is based on [16, Theorem 1] that also implies that reversal-bounded reachability problem considered in our paper is in NExpTime (assuming atomic guards of the form $\mathtt{x} \sim k$). Unlike [16], our LTL dialect contains control states, past-time operators but also arithmetical

constraints in QFP allowing non-trivial arithmetical properties like $\mathbf{GF}(\mathtt{Xx} = \mathtt{x} + \mathtt{y})$ (which may lead to undecidability in the general case). Similarly, even though our paper deals only with counter systems (no stack, no clocks), we allow general guards from QFP; we also introduce a new concept for reversal-boundedness. The proof of [16, Theorem 1] share common features with our proof of Theorem 2, at least in the use of counter modes. In both cases Presburger formulae are built: our proof is based on a run analysis whereas the proof in [16, Theorem 1] builds directly the formula. We believe our treatment is more uniform and it generalizes notions presented in [19,10]. Moreover, our run analysis for proving Theorem 2 is interesting for its own sake, see [4].

In general, omitted proofs can be found in [4] (submitted version).

2 Preliminaries

In this section, we introduce a language for arithmetical constraints, namely the quantifier-free fragment of Presburger arithmetic (over the set of natural numbers). This language serves two main purposes. Firstly, we define classes of operational models, namely counter systems, for which transitions are guarded by arithmetical constraints. Secondly, we introduce a version of linear-time temporal logic with past-time operators for which atomic formulae can state properties about counter values, i.e. arithmetical constraints.

Arithmetical Constraints. We write $\mathbb{N}$ (resp. $\mathbb{Z}$) for the set of natural (resp. integers) numbers and $[m, m']$ with $m, m' \in \mathbb{Z}$ to denote the set $\{j \in \mathbb{Z} : m \leq j \leq m'\}$. For $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{Z}^n$, we write $\boldsymbol{x}(1), \ldots, \boldsymbol{x}(n)$ for the entries of $\boldsymbol{x}$, $\boldsymbol{x} \preceq \boldsymbol{y} \stackrel{\text{def}}{\Leftrightarrow}$ for all $i \in [1, n]$, $\boldsymbol{x}(i) \leq \boldsymbol{y}(i)$ and $\boldsymbol{x} \prec \boldsymbol{y}$ when $\boldsymbol{x} \preceq \boldsymbol{y}$ and $\boldsymbol{x} \neq \boldsymbol{y}$.

Let $\mathrm{VAR} = \{\mathtt{x}_0, \mathtt{x}_1, \ldots\}$ be a countably infinite set of variables. We define below formulae from the quantifier-free theory of natural numbers with addition, also known as quantifier-free fragment of Presburger arithmetic. Terms $\mathtt{t}$ are defined from the grammar $\mathtt{t} := a\mathtt{x} \mid \mathtt{t} + \mathtt{t}$, where $\mathtt{x} \in \mathrm{VAR}$, $a \in \mathbb{Z}$ (encoded with a binary representation). A *valuation* $\mathbf{val}$ is a map $\mathbf{val} : \mathrm{VAR} \to \mathbb{N}$ and it can be extended to the set of all terms as follows: $\mathbf{val}(a\mathtt{x}) = a \times \mathbf{val}(\mathtt{x})$, $\mathbf{val}(\mathtt{t} + \mathtt{t}') = \mathbf{val}(\mathtt{t}) + \mathbf{val}(\mathtt{t}')$. It is worth noting that variables take values over $\mathbb{N}$ but terms take values over $\mathbb{Z}$. Formulae ξ of QFP are defined from the grammar $\xi ::= \top \mid \mathtt{t} \leq k \mid \mathtt{t} \geq k \mid \mathtt{t} \equiv_c k' \mid \xi \wedge \xi \mid \neg\xi$, where $\top$ is the truth constant, $c \in \mathbb{N} \setminus \{0, 1\}$, $k \in \mathbb{Z}$ and $k' \in \mathbb{N}$. The satisfaction relation $\models_{\mathrm{PA}}$ for QFP formulae is briefly recalled below:

- $\mathbf{val} \models_{\mathrm{PA}} \mathtt{t} \equiv_c k' \stackrel{\text{def}}{\Leftrightarrow}$ there is $n \in \mathbb{Z}$ such that $nc + \mathbf{val}(\mathtt{t}) = k'$,
- $\mathbf{val} \models_{\mathrm{PA}} \mathtt{t} \leq k \stackrel{\text{def}}{\Leftrightarrow} \mathbf{val}(\mathtt{t}) \leq k$ (and similarly with $\geq$),
- $\mathbf{val} \models_{\mathrm{PA}} \neg\phi \stackrel{\text{def}}{\Leftrightarrow} \mathbf{val} \not\models_{\mathrm{PA}} \phi$; $\mathbf{val} \models_{\mathrm{PA}} \phi \wedge \phi' \stackrel{\text{def}}{\Leftrightarrow} \mathbf{val} \models_{\mathrm{PA}} \phi$ and $\mathbf{val} \models_{\mathrm{PA}} \phi'$.

A valuation $\mathbf{val}$ restricted to variables in $V = \{\mathtt{x}_1, \ldots, \mathtt{x}_n\} \subseteq \mathrm{VAR}$ can be also represented by a vector $\boldsymbol{x} \in \mathbb{N}^n$, where $\mathbf{val}(\mathtt{x}_j) = \boldsymbol{x}(j)$ for $j \in [1, n]$. Hence, assuming that ϕ has n distinct variables, the satisfaction relation can be equivalently written with respect to a vector of values: $\boldsymbol{x} \models_{\mathrm{PA}} \phi$ (in place of $\mathbf{val} \models_{\mathrm{PA}} \phi$

with $\mathbf{val}(\mathtt{x}_i) = \boldsymbol{x}(i)$). Full Presburger arithmetic (i.e., with first-order quantification over natural numbers) has been shown decidable in [28] by means of quantifier elimination. Moreover, the satisfiability problem for QFP is known to be NP-complete, see e.g. [27].

We present below a few notations used in the sequel: QFP is also denoted by QFP$(<, \equiv)$ whereas its restriction without periodicity constraints is denoted by QFP$(<)$. Similarly, we write QFP$(<_1, \equiv)$ to denote the restriction of QFP$(<, \equiv)$ with atomic formulae involving at most one variable; QFP$(<_1, \equiv)$ without periodicity constraints is denoted by QFP$(<_1)$. Wlog., we can assume that the atomic formulae of QFP$(<_1, \equiv)$ are of one of the forms below: $\mathtt{x} \sim k$ with $k \in \mathbb{N}$ and $\sim \in \{<, \leq, >, \geq\}$ or $\mathtt{t} \equiv_c k'$ with $c > 1$ and $k' \in [0, c-1]$.

Counter Systems. In this paper, we consider *counter systems* to be finite-state automata equipped with a finite set of counters $\{1, \ldots, n\}$ with values over $\mathbb{N}$; a counter system is a tuple $\mathcal{S} = (Q, n, \delta)$ where Q is a finite set of control states, $n \geq 1$ is the number of counters and δ is a finite subset of $Q \times (\text{QFP} \times \mathbb{Z}^n) \times Q$ such that whenever $(q, (\phi, \boldsymbol{v}), q') \in \delta$ (also written $q \xrightarrow{(\phi, \boldsymbol{v})} q'$), ϕ is a formula in QFP with variables among $\mathtt{x}_1, \ldots, \mathtt{x}_n$ (a guard on the n counters) and $\boldsymbol{v} \in \mathbb{Z}^n$ is the *update vector*. Elements of δ are called *transitions*, i.e. rules acting on counters. A *configuration* of $\mathcal{S}$ is defined as a pair $(q, \boldsymbol{x}) \in Q \times \mathbb{N}^n$, where $\boldsymbol{x}$ is the vector of values for counters. The *one-step transition relation* $\rightarrow \subseteq Q \times \mathbb{N}^n \times Q \times \mathbb{N}^n$ is defined between a pair of configurations such that $((q, \boldsymbol{x}), (q', \boldsymbol{x'})) \in \rightarrow \stackrel{\text{def}}{\Leftrightarrow}$ there is a transition $t = q \xrightarrow{(\phi, \boldsymbol{v})} q'$ in δ, $\boldsymbol{x} \models_{\text{PA}} \phi$ and $\boldsymbol{x'} = \boldsymbol{x} + \boldsymbol{v}$ (in that case, we write $(q, \boldsymbol{x}) \xrightarrow{t} (q', \boldsymbol{x'})$). A *run* ρ is a (possibly infinite) sequence of configurations $(q_0, \boldsymbol{x}_0), (q_1, \boldsymbol{x}_1) \ldots$ such that two successive configurations agree with δ, i.e. for $i \geq 0$, we have $(q_i, \boldsymbol{x_i}) \xrightarrow{t} (q_{i+1}, \boldsymbol{x_{i+1}})$, for some $t \in \delta$. An *initialized* counter system is a pair $(\mathcal{S}, (q, \boldsymbol{x}))$ such that $\mathcal{S}$ is a counter system and $(q, \boldsymbol{x})$ is an *initial configuration* (with $\boldsymbol{x} \geq 0$).

Given a subset L of QFP, we write CS(L) to denote the class of counter systems for which transitions are restricted to guards in L. Clearly, Minsky machines (and also vector addition systems with states) are included in CS(QFP$(<_1)$). Then, most of all the reachability problems are already undecidable as soon as CS(L) contains CS(QFP$(<_1)$). For this reason, in order to get decidability for reachability and model-checking problems, some restrictions have to be imposed on the nature of the systems. The notion of reversal-boundedness introduced in [20] is based on a semantical restriction that entails the decidability of several reachability problems. Informally, a *reversal* for a counter occurs in a run when there is an alternation from nonincreasing to nondecreasing mode.

Below, we propose a slight generalization that captures the notion of reversal-boundedness from [20] and the notion of *strong* reversal-boundedness introduced in [19, Section 4.2.2]. In a few words, in our new definition below, reversal-boundedness applies to counters but *also* to terms occurring in guards. Let $\mathcal{S} = (Q, n, \delta)$ be a counter system and $\mathtt{T}$ be a finite set of terms including $\{\mathtt{x}_1, \ldots, \mathtt{x}_n\}$. Let us linearly order the terms in $\mathtt{T}$ with $\mathtt{x}_1, \ldots, \mathtt{x}_n, \mathtt{t}_1, \ldots, \mathtt{t}_{n'}$. So, $\text{card}(\mathtt{T}) = n + n'$ (n' can possibly be equal to 0). From a run $\rho = (q_0, \boldsymbol{x_0}), (q_1, \boldsymbol{x_1}), \ldots$ of

$\mathcal{S}$, in order to describe the behavior of counters and terms varying along ρ, we define a sequence of *mode vectors* $\boldsymbol{m_0}, \boldsymbol{m_1}, \ldots$ (of the same length as ρ) such that each $\boldsymbol{m_i}$ belongs to $\{\nearrow, \searrow\}^{n+n'}$. Intuitively, each value in a mode vector records whether a term is currently in an increasing phasis or in an decreasing phasis (this includes the counters themselves as in standard reversal-boundedness). Given a term $\mathtt{t} = \sum_k a_k \mathtt{x}_k$ and a counter vector $\boldsymbol{x}$, we write $\boldsymbol{x}(\mathtt{t})$ to denote the integer $\sum a_k \boldsymbol{x}(k)$. We are now ready to define the sequence $\boldsymbol{m_0}, \boldsymbol{m_1}, \ldots$

- By convention, m_0 is the unique vector in $\{\nearrow\}^{n+n'}$.
- For $j \geq 0$ and $i \in [1, n+n']$ with the ith term in $\mathtt{T}$ equal to $\mathtt{t}$, we have $\boldsymbol{m}_{j+1}(i) \stackrel{\text{def}}{=} \boldsymbol{m}_j(i)$ when $\boldsymbol{x}_j(\mathtt{t}) = \boldsymbol{x}_{j+1}(\mathtt{t})$, $\boldsymbol{m}_{j+1}(i) \stackrel{\text{def}}{=} \nearrow$ when $\boldsymbol{x}_{j+1}(\mathtt{t}) - \boldsymbol{x}_j(\mathtt{t}) > 0$ and $\boldsymbol{m}_{j+1}(i) \stackrel{\text{def}}{=} \searrow$ when $\boldsymbol{x}_{j+1}(\mathtt{t}) - \boldsymbol{x}_j(\mathtt{t}) < 0$.

It is worth noting that if $(q_j, \boldsymbol{x_j}) \xrightarrow{t} (q_{j+1}, \boldsymbol{x_{j+1}})$ with $t = q_j \xrightarrow{(\phi, \boldsymbol{v})} q_{j+1}$, then $\boldsymbol{x}_{j+1}(\mathtt{t}) - \boldsymbol{x}_j(\mathtt{t}) = \sum_k a_k \boldsymbol{v}(k)$. Now, let $Rev_i = \{j \in \mathbb{N} : \boldsymbol{m}_j(i) \neq \boldsymbol{m}_{j+1}(i)\}$; we say that ρ is r-$\mathtt{T}$-*reversal-bounded* for some $r \geq 0 \stackrel{\text{def}}{\Leftrightarrow}$ for all $i \in [1, n+n']$, $\text{card}(Rev_i) \leq r$. Given a counter system $\mathcal{S}$, we write $\mathtt{T}_{\mathcal{S}}$ to denote the finite set of terms $\mathtt{t}$ occurring in atomic guards of the form $\mathtt{t} \sim k$ with $\sim \in \{\leq, \geq\}$ and $k \in \mathbb{Z}$, plus the distinguished terms (counters) from $\{\mathtt{x}_1, \ldots, \mathtt{x}_n\}$. Note that terms occurring only in periodicity constraints are not taken into account; we shall deal with them separately (see Section 3). An initialized counter system $(\mathcal{S}, (q, \boldsymbol{x}))$ is *reversal-bounded* $\stackrel{\text{def}}{\Leftrightarrow}$ there is $r \geq 0$ such that every run from $(q, \boldsymbol{x})$ is r-$\mathtt{T}_{\mathcal{S}}$-reversal-bounded.

When $\mathtt{T}$ is reduced to $\{\mathtt{x}_1, \ldots, \mathtt{x}_n\}$, $\mathtt{T}$-reversal-boundedness is equivalent to reversal-boundedness from [20]. Hence, for $\mathcal{S} \in \text{CS}(\text{QFP}(<_1))$ and initial configuration $(q, \boldsymbol{x})$, $(\mathcal{S}, (q, \boldsymbol{x}))$ is reversal-bounded in the sense herein iff $(\mathcal{S}, (q, \boldsymbol{x}))$ is reversal-bounded in the sense from [20]. In strong reversal-boundedness [19, Sect. 4.2.2], a phasis can be either strictly increasing, or strictly decreasing or constant (mode vectors belong to $\boldsymbol{m_i} \in \{\nearrow, \searrow, \rightarrow\}^{n+n'}$). This provides more constraints on runs: the guards are more general (typically in QFP($<$)) and the update vectors are in $\{-1, 0, +1\}^n$. Again, our notion of $\mathtt{T}$-reversal-boundedness allows us to provide a uniform and more general treatment. Indeed, when a sequence of transitions has a unique update vector, the mode vector remains constant. When an initialized counter system from CS(QFP), involving guards with terms in $\mathtt{T}'$, is strongly reversal-bounded in the sense of [19, Sect. 4.2.2], then it is $(\mathtt{T}' \cup \{\mathtt{x}_1, \ldots, \mathtt{x}_n\})$-reversal-bounded, too.

Given a class $\mathcal{C}$ of counter systems, the *reversal-bounded reachability problem for* $\mathcal{C}$, written RB-REACH($\mathcal{C}$), is defined as follows (all integers are encoded in binary): given a counter system $\mathcal{S} \in \mathcal{C}$, configurations $(q_0, \boldsymbol{x_0})$ and $(q_f, \boldsymbol{x_f})$, $r \geq 0$, is there an r-$\mathtt{T}_{\mathcal{S}}$-reversal-bounded run from $(q_0, \boldsymbol{x_0})$ to $(q_f, \boldsymbol{x_f})$? Clearly, when $(\mathcal{S}, (q_0, \boldsymbol{x_0}))$ is reversal-bounded, reversal-bounded reachability corresponds exactly to reachability. Similarly, the *reversal-bounded control state repeated reachability problem for* $\mathcal{C}$, written RB-REP-REACH($\mathcal{C}$), is defined as follows: given a counter system $\mathcal{S} \in \mathcal{C}$, a configuration $(q_0, \boldsymbol{x_0})$, a control state q_f and $r \geq 0$, is there an infinite r-$\mathtt{T}_{\mathcal{S}}$-reversal-bounded run from $(q_0, \boldsymbol{x_0})$ such that q_f is repeated infinitely often? Both problems RB-REACH($\mathcal{C}$) and RB-REP-REACH($\mathcal{C}$)

restrict the set of runs witnessing a simple property (reaching $(q_f, \boldsymbol{x_f})$ or repeating infinitely often q_f). This makes sense in our incremental approximation approach, since removing the restriction leads to undecidability. However, it is worth noting that our new notion of T-reversal-boundedness is rich enough so that witness runs include standard reversal-bounded runs.

In the sequel, we show that RB-REACH(CS(QFP)) is NExpTime-complete. It is worth explaining why this is consistent with the fact that the reachability problem for (standard) reversal-bounded counter automata augmented with guards of the form $\mathtt{x}_i = \mathtt{x}_{i'}$ or $\mathtt{x}_i \neq \mathtt{x}_{i'}$ is undecidable [19]. Indeed, the presence of such guards entails the presence of terms of the form $\mathtt{x}_i - \mathtt{x}_{i'}$, that have to be reversal-bounded by definition of RB-REACH(CS(QFP)). However, it is not difficult to show that the undecidability proof in [19] produces enriched counter automata for which some terms of the form $\mathtt{x}_i - \mathtt{x}_{i'}$ are not reversal-bounded.

Reversal-Bounded Model-Checking Problems. We define below a linear-time temporal logic with future-time and past-time operators. Atomic formulae are either control states or arithmetical constraints about counter values at the current position and next position. *Counter variables* in $\mathrm{VAR} = \{\mathtt{x}_1, \mathtt{x}_2, \ldots\}$ are free variables, only interpreted by the counter values on configurations. As for defining QFP, *arithmetical terms* are defined by the grammar $\mathtt{t} ::= a\mathtt{x} \mid a\mathrm{X}\mathtt{x} \mid \mathtt{t} + \mathtt{t}$ with $\mathtt{x} \in \mathrm{VAR}$ and $a \in \mathbb{Z}$. Intuitively, $\mathtt{x}$ refers to the current value for counter $\mathtt{x}$, $\mathrm{X}\mathtt{x}$ refers to the counter value for $\mathtt{x}$ at the next position from the current one. Formulae of CLTL(QFP) are defined as follows:

$$\phi ::= \top \mid q \mid \mathtt{t} \sim k \mid \mathtt{t} \equiv_c k' \mid \neg\phi \mid \phi \wedge \phi \mid \mathbf{X}\phi \mid \phi\mathbf{U}\phi \mid \mathbf{Y}\phi \mid \phi\mathbf{S}\phi$$

with $q \in Q$, $\sim \in \{<, \leq, >, \geq, =\}$, $k \in \mathbb{Z}$, $c \in \mathbb{N} \setminus \{0, 1\}$ and $k' \in \mathbb{N}$. As usual, we pose $\mathbf{F}\phi \stackrel{\text{def}}{=} \top\mathbf{U}\phi$ and $\mathbf{G}\phi \stackrel{\text{def}}{=} \neg\mathbf{F}\neg\phi$. The formula $\mathbf{GF}(\mathtt{x}_1 - \mathtt{x}_2 = 3)$ states that infinitely often the value for counter 1 is equal to the value for counter 2 plus 3. Given a fragment $\mathrm{L} \subseteq \mathrm{QFP}$, we write CLTL(L) to denote the restriction of CLTL(QFP) with arithmetical constraints built from L.

Models of CLTL(QFP) are intended to be infinite runs of counter systems; hence they are of the form $\rho = (q_0, \boldsymbol{x_0}), (q_1, \boldsymbol{x_1}), (q_2, \boldsymbol{x_2}), \ldots$ with $\rho \in (Q \times \mathbb{N}^n)^\omega$. In order to deal with arithmetical constraints, we need to introduce a few notations. Given a term $\mathtt{t}$ from CLTL(QFP), we write $\tilde{\mathtt{t}}$ to denote the term in QFP obtained from $\mathtt{t}$ by replacing $\mathrm{X}\mathtt{x}_i$ by a fresh variable $\mathtt{x}'_i$. Then, satisfaction relation $\models$ is defined as follows (we omit obvious Boolean clauses):

- $\rho, i \models q \stackrel{\text{def}}{\Leftrightarrow} q = q_i$.
- $\rho, i \models \mathtt{t} \sim k \stackrel{\text{def}}{\Leftrightarrow} \mathbf{val} \models_{\mathrm{PA}} \tilde{\mathtt{t}} \sim k$ where for $j \in [1, n]$, $\mathbf{val}(\mathtt{x}_j) = \boldsymbol{x_i}(j)$ and $\mathbf{val}(\mathtt{x}'_j) = \boldsymbol{x_{i+1}}(j)$. Similarly, $\rho, i \models \mathtt{t} \equiv_c k' \stackrel{\text{def}}{\Leftrightarrow} \mathbf{val} \models_{\mathrm{PA}} \tilde{\mathtt{t}} \equiv_c k'$.
- $\rho, i \models \mathbf{X}\phi \stackrel{\text{def}}{\Leftrightarrow} \rho, i+1 \models \phi$; $\rho, i \models \mathbf{Y}\phi \stackrel{\text{def}}{\Leftrightarrow} \rho, i-1 \models \phi$ and $i \geq 1$.
- $\rho, i \models \phi\mathbf{U}\phi' \stackrel{\text{def}}{\Leftrightarrow}$ there is $j \geq i$ s.t. $\rho, j \models \phi'$ and for all $h \in [i, j-1]$, $\rho, h \models \phi$.
- $\rho, i \models \phi\mathbf{S}\phi' \stackrel{\text{def}}{\Leftrightarrow}$ there is $j \leq i$ s.t. $\rho, j \models \phi'$ and for all $h \in [j-1, i]$, $\rho, h \models \phi$.

Observe that $\mathbf{X}$ is a temporal operator whereas X is used to refer to next counter values and it does not admit nesting. Moreover, the syntax of CLTL(QFP) does

not allow terms that refer to counter values at the previous position; again, this can be easily simulated. For instance, current value for counter 1 is equal to the value of counter 2 at the previous position can be encoded by the formula $\mathbf{Y}(\mathtt{x}_2 = \mathrm{X}\mathtt{x}_1)$.

The basic idea behind the design of CLTL(QFP) is to allow comparisons between counter values at successive positions of the runs. Similar motivations can be found in the introduction of concrete domains in description logics, that are logic-based formalisms for knowledge representation [25]. Temporal logics with Presburger constraints have been developed, for instance, in [9,8,22]. Some of them have quite expressive decidable fragments. Undecidability of the existential model-checking problem for CLTL(QFP) can be shown using the undecidability of the halting problem for Minsky machines. SMT solvers can be used for checking bounded reachability problems, see e.g., [5].

Given an CLTL(QFP) formula ϕ, we write $\mathtt{T}_\phi$ to denote the finite set of terms of the form $\sum_k (a_k + b_k)\mathtt{x}_k$ when $\mathtt{t} = (\sum_k a_k \mathrm{X}\mathtt{x}_k) + (\sum_k b_k \mathtt{x}_k)$ is a term occurring in ϕ (modulo AC for the operator $+$) in an atomic formula of the form $\mathtt{t} \sim k$ with $\sim \in \{\leq, \geq, <, >, =\}$ and $k \in \mathbb{Z}$. Since the next value of counter k (denoted by $\mathrm{X}\mathtt{x}_k$) is equal to the current value of the counter plus some $b \in \mathbb{Z}$ (depending on the update vectors of the transitions), the value of the term $(\sum_k a_k \mathrm{X}\mathtt{x}_k) + (\sum_k b_k \mathtt{x}_k)$ is equal to the current value of $\sum_k (a_k + b_k)\mathtt{x}_k$ plus some constant depending on the next transition. This explains the current definition of $\mathtt{T}_\phi$ and more justifications can be found in Section 3.

Reversal-bounded model-checking problem. RBMC is defined as follows: given a counter system $\mathcal{S} \in \mathrm{CS(QFP)}$, a configuration $(q, \boldsymbol{x})$, a formula $\phi \in \mathrm{CLTL(QFP)}$ and bound $r \in \mathbb{N}$, is there an infinite run ρ from $(q, \boldsymbol{x})$ such that $\rho, 0 \models \phi$ and ρ is r-$\mathtt{T}$-reversal-bounded with $\mathtt{T} = \mathtt{T}_\mathcal{S} \cup \mathtt{T}_\phi$? The restriction of RBMC to counter systems in the class $\mathrm{CS(L_1)}$ and to formulae in $\mathrm{CLTL(L_2)}$ is denoted by $\mathrm{RBMC(CS(L_1),CLTL(L_2))}$ with $\mathrm{L_1, L_2} \subseteq \mathrm{QFP}$. If $\mathrm{L_1 = L_2 = QFP}(<_1)$, the witness run ρ should simply be reversal-bounded in the sense of [20] ($\mathtt{T}_\phi = \mathtt{T}_\mathcal{S} = \{\mathtt{x}_1, \ldots, \mathtt{x}_n\}$). Similarly, if $\mathrm{L_1 = L_2 = QFP}$, then the set of witness runs include the set of strongly reversal-bounded runs from $(q, \boldsymbol{x})$ in the sense of [19, Section 4.2.2]. We can impose that witness runs are exactly strongly reversal-bounded by adding the subformula $\bigvee_{q \xrightarrow{(\xi, \boldsymbol{v})} q' \in \delta} \mathbf{FG}(\bigwedge_{i \in [1,n]}((\mathrm{X}\mathtt{x}_i - \mathtt{x}_i) = \boldsymbol{v}(i)))$. Do note that a richer class of witness runs is allowed by our definition. The main result of the paper is the NExpTime-completeness of RBMC (with all integers admitting a binary representation). Observe also that both RB-REACH(CS(QFP)) and RB-REP-REACH(CS(QFP)) can be easily reduced to RBMC.

3 From Reversal-Bounded Model-Checking to Reachability

Herein, we show how to reduce RMBC into RB-REP-REACH(QFP), RB-REP-REACH(QFP) into RB-REP-REACH(QFP(<)) and RB-REP-REACH(QFP(<)) into RB-REACH(QFP(<)). In Section 4, we deal with RB-REACH(QFP(<)) complexity as well as with RMBC and RB-REP-REACH(QFP) complexity. The

two first reductions presented below use quite standard proof techniques but we have to perform them carefully since we shall reuse their complexity functions to establish the final complexity upper bound for RMBC, see e.g. [12] for the first reduction (see also [33]). It is worth nothing that each reduction below produces an exponential blow-up.

Towards Control State Repeated Reachability. In this section, we show how to reduce RBMC to RB-REP-REACH(QFP) by synchronizing counter systems with Büchi automata for temporal formulae, as done for LTL model-checking [34], see also developments for Petri nets in [12]. The definition of a synchronized product is motivated by the design of a unique counter system that captures the Büchi acceptance condition and the update of counters following the transitions of $\mathcal{S}$.

Let $\mathcal{S} = (Q, n, \delta) \in \mathrm{CS(QFP)}$, $(q, \boldsymbol{x})$, $\phi \in \mathrm{CLTL(QFP)}$ and $r \in \mathbb{N}$ be an instance of RBMC. The formula ϕ can be viewed as a standard LTL formula in which the atomic formulae of the form q, $\mathtt{t} \sim k$ and $\mathtt{t} \equiv_c k'$ are viewed as propositional variables. From [34], we know that we can represent the symbolic models of ϕ by a Büchi automaton $\mathcal{A}_\phi$ whose size is exponential in the size of ϕ. At the symbolic level, the counter values are disgarded. The instance we shall build for RB-REP-REACH(QFP) is obtained by synchronizing $\mathcal{A}_\phi$ with $\mathcal{S}$, providing the counter system $\mathcal{S}'$ such that $\mathtt{T}_{\mathcal{S}'} = \mathtt{T}_{\mathcal{S}} \cup \mathtt{T}_\phi$.

Let us be a bit more precise in the construction of $\mathcal{A}_\phi$. We write A to denote the set of atomic formulae of the form either q, or $\mathtt{t} \sim k$ or $\mathtt{t} \equiv_c k'$ occurring in ϕ, as well as their negations. Similarly, we write $cl(\phi)$ to denote the *closure* of ϕ, defined as the smallest set of formulae closed under subformulae, closed under negations (double negations are eliminated) and containing ϕ. The set of *atoms* for ϕ, written $Atoms(\phi)$, contains the subsets of $cl(\phi)$ that are maximally consistent and such that for every formula $\xi \in A$ then either ξ or $\neg\xi$ belongs to the set (but not both). States of $\mathcal{A}_\phi$ are in $Atoms(\phi) \times [0, m]$ where ϕ has m **U**-formulae and its alphabet is a subset of $Q \times \mathcal{P}(A)$ (details can be found in [4] with the standard construction for the synchronized product $\mathcal{S}'$). An instance of RBMC can be reduced to several instances of RB-REP-REACH(QFP) with the synchronized product $\mathcal{S}'$. In particular, RMBC can be solved by checking a finite number of instances of RB-REP-REACH(QFP) depending which initial states and accepting states are considered.

Lemma 1. *Let $\mathcal{S} = (Q, n, \delta) \in \mathrm{CS(QFP)}$, $(q, \boldsymbol{x})$, $\phi \in \mathrm{CLTL(QFP)}$ and $r \in \mathbb{N}$ be an instance of RBMC and $\mathcal{S}'$ be the counter system in* CS(QFP) *obtained by synchronizing $\mathcal{S}$ with $\mathcal{A}_\phi$. The propositions below are equivalent:* (I) *there is an infinite r-($\mathtt{T}_{\mathcal{S}} \cup \mathtt{T}_\phi$)-reversal-bounded run ρ of $\mathcal{S}$ from $(q, \boldsymbol{x})$ such that $\rho, 0 \models \phi$;* (II) *there is an infinite r-$\mathtt{T}_{\mathcal{S}'}$-reversal-bounded run from $((q, X_0, 0), \boldsymbol{x})$ such that $(q_f, X_f, 0)$ is repeated infinitely often for some* initial *atom $X_0 \in Atoms(\phi)$ and for some $(q_f, X_f) \in Q \times Atoms(\phi)$.*

Actually, thanks to the previous lemma, the following corollary holds:

Corollary 1. *There is a polynomial-space reduction from* RMBC *into* RB-REP-REACH(CS(QFP)).

The next section is devoted to show how to reduce RB-REP-REACH(QFP($<$,$\equiv$)) to RB-REP-REACH(QFP($<$)).

Removing Periodicity Constraints. In this section, we show that given L $\subseteq$ QFP using periodicity constraints of the form $\mathtt{t} \equiv_c k$, the reversal-bounded reachability problem for counter systems in CS(L) can be reduced to the corresponding problem restricted to counter systems in CS(L$'$), where L$'$ is the restriction of L without periodicity constraints.

Reduction. Let us consider the class of counter systems CS(L). The underlying idea to remove periodicity constraints consists in defining a new counter system $\mathcal{S}' \in$ CS(L$'$) from a given $\mathcal{S} \in$ CS(L), whose control states store counter values modulo C, where C is the lcm of all the constants c appearing in atomic formulae of the form $\mathtt{t} \equiv_c k$ in guards of $\mathcal{S}$ (see [4] for standard justifications about the value C). The number of control states in $\mathcal{S}'$ is equal to number of control states in $\mathcal{S}$ multiplied by C, which is in $\mathcal{O}(2^{N^2})$ (N is the size of $\mathcal{S}$ with some reasonably succinct encoding). This construction entails an exponential blow-up of the number of control states of the new counter system $\mathcal{S}'$. The transitions of $\mathcal{S}'$ are defined accordingly to the update operations on them in order to correctly represent the classes of modulo for each counter. Let $\mathcal{S}' = (Q', n, \delta')$ be the counter system where $Q' = Q \times [0, C-1]^n$. Given $\boldsymbol{x} \in \mathbb{N}^n$, we write $\tilde{\boldsymbol{x}}$ to denote the unique tuple in $[0, C-1]^n$ such that for $i \in [1, n]$, we have $\boldsymbol{x}(i) \equiv_C \tilde{\boldsymbol{x}}(i)$. Let config$_{ok}$ be the set of configurations for $\mathcal{S}'$ of the form $((q, \tilde{\boldsymbol{x}}), \boldsymbol{y})$ such that $\tilde{\boldsymbol{y}} = \tilde{\boldsymbol{x}}$. Let $f : (Q \times \mathbb{N}^n) \to$ config$_{ok}$ be the one-to-one map such that $f((q, \boldsymbol{x})) = ((q, \tilde{\boldsymbol{x}}), \boldsymbol{x})$. f and f^{-1} extend naturally to sequences (either finite or infinite ones). The transition relation δ' is defined as follows: if $q \xrightarrow{(\phi, \boldsymbol{b})} q' \in \delta$ then $(q, \tilde{\boldsymbol{x}}) \xrightarrow{(\phi', \boldsymbol{b})} (q', \tilde{\boldsymbol{y}}) \in \delta'$ for all tuples $\tilde{\boldsymbol{x}}$, $\tilde{\boldsymbol{y}}$, where ϕ' is defined from ϕ by substituting $\top$ in place of each $\mathtt{t} \equiv_c k$, with $\mathtt{t} = \sum_j a_j \mathtt{x}_j$, if $\sum_j a_j \tilde{\boldsymbol{x}}(j) \equiv_c k$; otherwise $\bot$. Moreover, we require that for $i \in [1, n]$, we have $\tilde{\boldsymbol{y}}(i) \equiv_C \tilde{\boldsymbol{x}}(i) + \boldsymbol{b}(i)$.

Lemma 2. *Let $\mathcal{S} = (Q, n, \delta)$ be in* CS(QFP) *and $\mathcal{S}' = (Q', n, \delta')$ be the counter system in* CS(QFP($<$)) *defined as above.* (I) *For every run ρ of $\mathcal{S}$, $f(\rho)$ is also a run of $\mathcal{S}'$.* (II) *For every run ρ of $\mathcal{S}'$ such that the first configuration belongs to* config$_{ok}$*, then all configurations in ρ belong to* config$_{ok}$ *and $f^{-1}(\rho)$ is also a run of $\mathcal{S}$.*

From the previous result, the following corollary can be drawn.

Corollary 2. *Let* L = QFP *[resp.* L = QFP($<_1, \equiv$)*] and* L$'$ = QFP($<$) *[resp.* L$'$ = QFP($<_1$)*].*

(I) *There is a polynomial-space reduction from* RB-REACH(CS(L)) *to* RB-REACH(CS(L$'$)). (II) *There is a polynomial-space reduction from* RB-REP-REACH(CS(L)) *to* RB-REP-REACH(CS(L$'$)).

Elimination of Büchi Acceptance Conditions. Let $\mathcal{S}$ be in CS(QFP($<$)), $(q_0, \boldsymbol{x_0})$ be an initial configuration, q_f be a control state and $r \geq 0$. We write $K_{min} \in \mathbb{Z}$ [resp. $K_{max} \in \mathbb{Z}$] to denote the minimal [resp. maximal] k occurring in atomic formulae of the form $\mathtt{t} \sim k$ in guards from $\mathcal{S}$. We show below how the existence of an *infinite* run can be characterized by the existence of a *finite*

run satisfying additional properties. The properties ($\star$) and ($\star\star$) below witness such an equivalence. This is comparable, but certainly a bit more technically involved, to the existence of infinite accepting runs in Büchi automata that is equivalent to conditions on finite runs. However, such a reduction is not possible with nondeterministic Minsky machines without the reversal-boundedness assumption. Indeed, the recurrence problem for nondeterministic Minsky machines is Σ_1^1-hard [1] whereas the halting problem for nondeterministic Minksy machines is in Σ_1^0. We show that the conditions below are equivalent.

($\star$) There is an infinite r-$\mathtt{T}_{\mathcal{S}}$-reversal-bounded run from $(q_0, \boldsymbol{x_0})$ such that q_f is repeated infinitely often.

($\star\star$) There exist a finite run $(q_0, \boldsymbol{x_0}), \ldots, (q_l, \boldsymbol{x_l})$, $l' < l$, $Z_{\rightarrow} \subseteq [1, n]$ and $\mathtt{T}_{\rightarrow}, \mathtt{T}_{\searrow}, \mathtt{T}_{\nearrow} \subseteq (\mathtt{T}_{\mathcal{S}} \setminus \{\mathtt{x}_1, \ldots, \mathtt{x}_n\})$ such that

1. $q_{l'} = q_l = q_f$ and $(q_0, \boldsymbol{x_0}), \ldots, (q_l, \boldsymbol{x_l})$ is r-$\mathtt{T}_{\mathcal{S}}$-reversal-bounded.
2. For $j \in [l'+1, l]$ and $i \in Z_{\rightarrow}$, $\boldsymbol{x_j}(i) - \boldsymbol{x_{j-1}}(i) = 0$.
3. For $j \in [l'+1, l]$ and $i \in [1, n] \setminus Z_{\rightarrow}$, $\boldsymbol{x_j}(i) - \boldsymbol{x_{j-1}}(i) \geq 0$.
4. For $i \in [1, n] \setminus Z_{\rightarrow}$, $\boldsymbol{x_{l'}}(i) \geq K_{max}$.
5. $\mathtt{T}_{\rightarrow}, \mathtt{T}_{\searrow}, \mathtt{T}_{\nearrow}$ is a partition of $(\mathtt{T}_{\mathcal{S}} \setminus \{\mathtt{x}_1, \ldots, \mathtt{x}_n\})$.
6. For $j \in [l'+1, l]$ and $\mathtt{t} \in \mathtt{T}_{\rightarrow}$, we have $\boldsymbol{x_j}(\mathtt{t}) - \boldsymbol{x_{j-1}}(\mathtt{t}) = 0$.
7. For $j \in [l'+1, l]$ and $\mathtt{t} \in \mathtt{T}_{\searrow}$, we have $\boldsymbol{x_j}(\mathtt{t}) - \boldsymbol{x_{j-1}}(\mathtt{t}) \leq 0$.
8. For $j \in [l'+1, l]$ and $\mathtt{t} \in \mathtt{T}_{\nearrow}$, $\boldsymbol{x_j}(\mathtt{t}) - \boldsymbol{x_{j-1}}(\mathtt{t}) \geq 0$.
9. For $\mathtt{t} \in \mathtt{T}_{\searrow}$, $\boldsymbol{x_{l'}}(\mathtt{t}) \leq K_{min}$; 10. For $\mathtt{t} \in \mathtt{T}_{\nearrow}$, $\boldsymbol{x_{l'}}(\mathtt{t}) \geq K_{max}$.

Lemma 3. *($\star$) is equivalent to ($\star\star$)*

Proof. ($\star$) implies ($\star\star$). Let $(q_0, \boldsymbol{x_0}), (q_1, \boldsymbol{x_1}), \ldots$ be an infinite r-$\mathtt{T}_{\mathcal{S}}$-reversal-bounded run from $(q_0, \boldsymbol{x_0})$ such that q_f is repeated infinitely often (with $\mathtt{T}_{\mathcal{S}} = \{\mathtt{x}_1, \ldots, \mathtt{x}_n\} \cup \{\mathtt{t}_1, \ldots, \mathtt{t}_{n'}\}$). All the atomic guards in $\mathcal{S}$ are of the form $\mathtt{t} \sim k$ with $\mathtt{t} \in \mathtt{T}_{\mathcal{S}}$ and $k \in [K_{min}, K_{max}]$. Let us make the following observations.

- Let $i \in [1, n]$. Because counter i has a bounded number of reversals, from some position, the value of counter i either remains constant or it is diverging to $+\infty$ and the update values (on counter i) are always greater than 0. Let $Z_{\rightarrow}$ be the subset of $[1, n]$ containing the counters whose values remain constant after some position. In the second case, there is a position j_1 such that for $j \geq j_1$, $\boldsymbol{x_j}(i) \geq K_{max}$, for all $i \in [1, n] \setminus Z_{\rightarrow}$.
- Let $i \in [1, n']$. Because the term $\mathtt{t}_i$ has a bounded number of reversals, one of the conditions below hold true (leading to the definition of $\mathtt{T}_{\rightarrow}, \mathtt{T}_{\searrow}, \mathtt{T}_{\nearrow}$).
  1. From some position, the value of the term $\mathtt{t}_i$ remains constant, i.e. there is $j_0 \in \mathbb{N}$, such that for $j \geq j_0$, $\boldsymbol{x_{j+1}}(\mathtt{t}_i) - \boldsymbol{x_j}(\mathtt{t}_i) = 0$.
  2. The value of the term $\mathtt{t}_i$ diverges to $-\infty$ and there is $j_0 \in \mathbb{N}$, such that for $j \geq j_0$, $\boldsymbol{x_{j+1}}(\mathtt{t}_i) - \boldsymbol{x_j}(\mathtt{t}_i) \leq 0$. In particular, there is a position $j_1 \geq j_0$ such that $\boldsymbol{x_{j_1}}(\mathtt{t}_i) \leq K_{min}$.
  3. The value of the term $\mathtt{t}_i$ diverges to $+\infty$ and there is $j_0 \in \mathbb{N}$, such that for $j \geq j_0$, $\boldsymbol{x_{j+1}}(\mathtt{t}_i) - \boldsymbol{x_j}(\mathtt{t}_i) \geq 0$. In particular, there is a position $j_1 \geq j_0$ such that $\boldsymbol{x_{j_1}}(\mathtt{t}_i) \geq K_{max}$.
- Since q_f is repeated infinitely often, there are two positions $l' < l$ satisfying the conditions (1)–(10).

($\star\star$) implies ($\star$). It remains to show that the existence of a finite run $(q_0, \boldsymbol{x_0})$, $(q_1, \boldsymbol{x_1})$, ..., $(q_l, \boldsymbol{x_l})$, $l' < l$, $Z_{\rightarrow} \subseteq [1, n]$ and $\mathtt{T}_{\rightarrow}, \mathtt{T}_{\searrow}, \mathtt{T}_{\nearrow} \subseteq (\mathtt{T}_{\mathcal{S}} \setminus \{\mathtt{x}_1, \ldots, \mathtt{x}_n\})$ such that (1)-(10) hold true implies that there is an infinite r-$\mathtt{T}_{\mathcal{S}}$-reversal-bounded run from $(q_0, \boldsymbol{x_0})$ such that q_f is repeated infinitely often. Let ρ be the run $(q_0, \boldsymbol{x_0}) \xrightarrow{t_1} (q_1, \boldsymbol{x_1}) \cdots \xrightarrow{t_{l'}} (q_{l'}, \boldsymbol{x_{l'}}) \cdots \xrightarrow{t_l} (q_l, \boldsymbol{x_l})$. For each transition t_i, we assume that the guard is ϕ_i and the update vector is $\boldsymbol{b_i}$. Let us consider the infinite sequence of configurations below

$$\rho' = (q_0, \boldsymbol{x_0}) \xrightarrow{t_1} (q_1, \boldsymbol{x_1}) \cdots \xrightarrow{t_{l'}} (q_{l'}, \boldsymbol{x_{l'}}) \cdots \xrightarrow{t_l} (q_l, \boldsymbol{x_l}) = (q_l, \boldsymbol{y_l}) \xrightarrow{t_{l'+1}} \cdots$$

$$\cdots (q_{l'+1}, \boldsymbol{y_{l'+(l-l')+1}}) \xrightarrow{t_l} (q_l, \boldsymbol{y_{l+(l-l')}}) \cdots$$

such that for $k \geq 0$ and $k' \in [0, l - l' - 1]$, we have $\boldsymbol{y_{l+k(l-l')+k'}} = \boldsymbol{x_{l+k'}} + k(\boldsymbol{x_l} - \boldsymbol{x_{l'}})$ and the sequence of transitions is $t_1 \cdots t_{l'}(t_{l'+1} \cdots t_l)^{\omega}$.

1. Obviously q_f is repeated infinitely often in ρ'.
2. ρ' is indeed a run as for $k \geq 0$ and $k' \in [0, l-l'-1]$, $\boldsymbol{y_{l+k(l-l')+k'}} \models \phi_{l'+1+k'}$ since $\boldsymbol{x_{l+k'}} \models \phi_{l'+1+k'}$ and after position l', atomic guards of the form $\mathtt{t} \sim k$ have a constant truth status. Indeed, $(\boldsymbol{x_l} - \boldsymbol{x_{l'}})$ is constant.
3. ρ' is r-$\mathtt{T}_{\mathcal{S}}$-reversal-bounded since after position l', no new reversal happens.

□

Theorem 1. *There is a polynomial-space many-one reduction from* RB-REP-REACH(CS(QFP(<))) *into* RB-REACH(CS(QFP(<))).

Proof. Let $\mathcal{S}$ be in CS(QFP(<)), $(q_0, \boldsymbol{x_0})$ be an initial configuration, q_f be a control state and $r \geq 0$. We write K_{min} [resp. K_{max}] to denote the minimal [resp. maximal] k occurring in atomic formulae of the form $\mathtt{t} \sim k$ in guards from $\mathcal{S}$. Let us build an instance of RB-REACH(CS(QFP(<))) which captures the condition ($\star\star$). We construct a counter automaton $\mathcal{S}' = (Q', n, \delta')$ such that ($\star\star$) iff there is an $(r+1)$-$\mathtt{T}_{\mathcal{S}'}$-reversal-bounded run from $(q_0, \boldsymbol{x_0})$ to $(q_{new}, \mathbf{0})$. $\mathcal{S}'$ is made of the original version of $\mathcal{S}$ (called below the *original copy*) augmented with copies of $\mathcal{S}$; each copy corresponds to a possible tuple $C = (Z_{\rightarrow}, \mathtt{T}_{\rightarrow}, \mathtt{T}_{\searrow}, \mathtt{T}_{\nearrow})$. By the *$C$-copy*, we mean the copy of $\mathcal{S}$ in which we keep only the transitions with update vector $\boldsymbol{b}$ such that for $i \in Z_{\rightarrow}$, $\boldsymbol{b}(i) = 0$; for $i \notin Z_{\rightarrow}$, $\boldsymbol{b}(i) \geq 0$. for $\mathtt{t} \in \mathtt{T}_{\rightarrow}$, $\boldsymbol{b}(\mathtt{t}) = 0$; for $\mathtt{t} \in \mathtt{T}_{\searrow}$, $\boldsymbol{b}(\mathtt{t}) \leq 0$; for $\mathtt{t} \in \mathtt{T}_{\nearrow}$, $\boldsymbol{b}(\mathtt{t}) \geq 0$.

In order to simulate the subrun $(q_{l'}, \boldsymbol{x_{l'}}) \cdots (q_l, \boldsymbol{x_l})$, from the original copy, nondeterministically we move from the original copy to some C-copy in $\mathcal{S}'$ (and therefore we choose the sets for C) and we test whether the counters in $[1, n] \setminus Z_{\rightarrow}$ have a value greater than K_{max} (with guards $\mathtt{x} \geq K_{max}$), the terms $\mathtt{t}$ in $\mathtt{T}_{\searrow}$ have a value smaller than K_{min} (with guards $\mathtt{t} \leq K_{min}$), the terms $\mathtt{t}$ in $\mathtt{T}_{\nearrow}$ have a value greater than K_{max} (with guards $\mathtt{t} \geq K_{max}$). Finally, in the C-copy, when q_f is reached again, nondeterministically we may jump to the new accepting control state q_{new}. Self-loops on q_{new} allows to decrement any counter. It is also worth noting that $\mathtt{T}_{\mathcal{S}'} = \mathtt{T}_{\mathcal{S}}$; $\mathcal{S}$ and $\mathcal{S}'$ have the same set of constants k occuring in atomic formulae of the form $\mathtt{t} \sim k$; the numbers of states of $\mathcal{S}'$ is bounded by $\mathrm{card}(Q) \times (1 + 2^n \times (2^{n'} \times 2^{n'}))) + 1$ (with $\mathrm{card}(\mathtt{T}_{\mathcal{S}}) = n + n'$). □

Given a counter system $\mathcal{S} = (Q, n, \delta)$ and an infinite run ρ, there exists at least one sequence of transitions $\pi \in \delta^\omega$ such that ρ is built from the successive firing of transitions from π. A sequence π is ultimately periodic if $\pi = \pi_1(\pi_2)^\omega$ for some finite sequences π_1 and π_2. The different reductions established in this section (see also their proofs) allow us to show the result below.

Corollary 3. *Let $\mathcal{S}$ be in* CS(QFP)*, $(q, \boldsymbol{x})$ be a configuration, ϕ be in* CLTL(QFP) *and $r \in \mathbb{N}$.* (I) *and* (II) *are equivalent:* (I) *there is an infinite run ρ from $(q, \boldsymbol{x})$ such that $\rho, 0 \models \phi$ and ρ is r-$\mathtt{T}$-reversal-bounded with $\mathtt{T} = \mathtt{T}_\mathcal{S} \cup \mathtt{T}_\phi$;* (II) *there exists an* ultimately periodic *run ρ satisfying the same properties as in (I).*

4 Complexity and Effective Presburger-Definability

In this section, we present the following results: RB-REACH(QFP), RB-REP-REACH(QFP) and RMBC are NExpTime-complete and the sets of initial configurations satisfying the properties related to these problems (witness run properties) are effectively definable in Presburger arithmetic, a key result for performing verification practically.

Theorem 2. RB-REACH(CS(QFP(<))) *is* NExpTime*-complete.*

The proof of Theorem 2 is the most involved part of the paper; it is presented in [4]. It generalizes the proof provided for [17, Theorem 3] and uses arguments that can be found also in [30] but in some other context (complexity upper bound for decision problems about Petri nets), see also [11]. It is essential to use the existence of small solutions for integer (inequality) systems [7]. Thanks to Theorem 2, we can improve [19, Theorem 4.4] by establishing that strong reversal-bounded reachability problem is in NExpTime (no complexity bound is provided in the proof of [19, Theorem 4.4]).

As a by-product of the previous result, we can show the following result.

Corollary 4. *Given $\mathcal{S}$ in* CS(QFP)*, $r \geq 0$ and control states q, q', one can effectively compute a Presburger formula $\phi_{q,q'}(\mathtt{x}_1, \ldots, \mathtt{x}_n, \mathtt{y}_1, \ldots, \mathtt{y}_n)$ such that for all valuations* $\mathbf{val}$*,* $\mathbf{val} \models_{\mathrm{PA}} \phi$ *iff there is an r-$\mathtt{T}_\mathcal{S}$-reversal-bounded run from $(q, (\mathbf{val}(\mathtt{x}_1), \ldots, \mathbf{val}(\mathtt{x}_n)))$ to $(q', (\mathbf{val}(\mathtt{y}_1), \ldots, \mathbf{val}(\mathtt{y}_n)))$.*

Consequently, when an initialized counter system is r-reversal-bounded for some $r \geq 0$, then the reachability set is effectively Presburger-definable. This captures the standard case when the counter system belongs to CS(QFP($<_1$)) [20,24] but Corollary 4 goes much beyond.

Theorem 3. RB-REP-REACH(CS(QFP(<))) *is* NExpTime*-complete.*

Corollary 5. *Given $\mathcal{S}$ in* CS(QFP)*, $r \geq 0$ and control states q, q_f, one can effectively compute a Presburger formula $\phi_{q,q_f}(\mathtt{x}_1, \ldots, \mathtt{x}_n)$ such that for all valuations* $\mathbf{val}$*,* $\mathbf{val} \models_{\mathrm{PA}} \phi$ *iff there is an infinite r-$\mathtt{T}_\mathcal{S}$-reversal-bounded run from $(q, (\mathbf{val}(\mathtt{x}_1), \ldots, \mathbf{val}(\mathtt{x}_n)))$ such that q_f is repeated infinitely often.*

We are now ready to state our main results (Theorem 4 and Theorem 5).

Theorem 4. RBMC *is* NExpTime*-complete.*

As a consequence, we obtain the following results since RB-REACH(QFP) and RB-REP-REACH(QFP) can be reduced in logarithmic space to RBMC.

Corollary 6. RB-REACH(QFP) *and* RB-REP-REACH(QFP) *are* NExpTime*-complete.*

Interestingly, vector addition systems with states (VASS) are elements of CS(QFP($<_1$)) and therefore RBMC(VASS,CLTLQFP($<_1, \equiv$))) is in NExpTime, which contrasts with the ExpSpace-completeness of the model-checking problem with LTL (the only atomic formulae are control states) restricted to VASS [15]. Unlike LTL, CLTL(QFP($<_1, \equiv$)) admits arithmetical constraints.

Theorem 5. *Let* $\mathcal{S}$ *be in* CS(QFP)*,* ϕ *be in* CLTL(QFP) $r \geq 0$ *and* q *be a control state. One can effectively build a Presburger formula* $\phi_q(\mathtt{x}_1, \ldots, \mathtt{x}_n)$ *such that for all* $\mathbf{val}$*,* $\mathbf{val} \models_{\text{PA}} \phi_q$ *iff there is an infinite run* ρ *from* $(q, (\mathbf{val}(\mathtt{x}_1), \ldots, \mathbf{val}(\mathtt{x}_n)))$ *such that* $\rho, 0 \models \phi$ *and* ρ *is* r*-*$\mathtt{T}$*-reversal-bounded with* $\mathtt{T} = \mathtt{T}_{\mathcal{S}} \cup \mathtt{T}_{\phi}$*.*

We are also able to improve Corollary 7 since we also have bounds on the lenght of reversal-bounded runs (see the proof of Theorem 4).

Corollary 7. *Let* $\mathcal{S}$ *be in* CS(QFP)*,* $(q, \boldsymbol{x})$ *be an initial configuration,* ϕ *be in* CLTL(QFP) *and* $r \in \mathbb{N}$*. Condition* (I) *in Corollary 3 is equivalent to* (II) *in Corollary 3 with the following additional condition: the sequence of transitions* $\pi_1(\pi_2)^\omega$ *verifies that the length of* $\pi_1\pi_2$ *is bounded by* $2^{2^{p_0(N)}}$*, for some polynom* $p_0(\cdot)$ *and* N *is the size of the instance of* RBMC*.*

Let us explain the benefits of these results from a practical point of view. From Theorem 5, given the formula $\phi_q(\mathtt{x}_1, \ldots, \mathtt{x}_n)$, we can check if an initial configuration verifies the existence of an infinite run satisfying a temporal formula. This can be done with a solver for Presburger arithmetic (tools handling first-order logics with linear arithmetic are for instance LIRA [3], TAPAS [23], CVC3 [2] and Z3 [26]). Hence, Theorem 5 is the final step in our investigations since verification problems are then reduced effectively to satisfiability in Presburger arithmetic. Moreover, our results on the computational complexity guarantee that we are optimal. Another approach arises from Corollary 7 which takes advantage of the method for checking bounded reachability problems as developed in [5]. Since an instance of RBMC can be transformed into an instance of RB-REACH(QFP) and by Theorem 2, one could solve the reversal-bounded model checking problem by looking for finite runs of length at most doubly exponential.

5 Conclusion

We have studied the model-checking problem RBMC over counter systems when runs are reversal-bounded and the specification language is an LTL-like dialect

with arithmetical constraints, past-time and future-time operators. A major result is the NExpTime-completeness of the problem RBMC. Even more importantly, in order to implement decision procedures, we have shown that given a counter system, a temporal formula ϕ and $r \geq 0$, one can build effectively a Presburger formula encoding the set of configurations $(q, \boldsymbol{x})$ such that there is an r-$(\mathtt{T}_\phi \cup \mathtt{T}_\mathcal{S})$-reversal-bounded infinite run ρ from $(q, \boldsymbol{x})$ such that ϕ is satisfied by ρ. Finally, we have also characterized the complexity of several reversal-bounded reachability problems and control state repeated reachability problem (obtaining NExpTime-completeness). It is worth noting that our proofs for NExpTime-easiness are obtained by an explicit run analysis that shortens the runs, as in [16] but in a different way.

Acknowledgment. We would like to thank the anonymous referees for their suggestions and constructive remarks; a special thank is due to the referee that pointed us to [16].

References

1. Alur, R., Henzinger, T.: A really temporal logic. In: FOCS 1989, pp. 164–169. IEEE, Los Alamitos (1989)
2. Barrett, C., Tinelli, C.: CVC3. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 298–302. Springer, Heidelberg (2007)
3. Becker, B., Dax, C., Eisinger, J., Klaedtke, F.: LIRA: Handling Constraints of Linear Arithmetics over the Integers and the Reals. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 307–310. Springer, Heidelberg (2007)
4. Bersani, M., Demri, S.: The complexity of reversal-bounded model checking. Tech. Rep. LSV-11-10, LSV, ENS Cachan, France (May 2011)
5. Bersani, M., Frigeri, A., Morzenti, A., Pradella, M., Rossi, M., San Pietro, P.: Bounded reachability for temporal logic over constraint systems. In: TIME 2010, pp. 43–50. IEEE, Los Alamitos (2010)
6. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Advances in Computers 58, 118–149 (2003)
7. Borosh, I., Treybig, L.: Bounds on positive integral solutions of linear diophantine equations. Proocedings of The American Mathematical Society 55, 299–304 (1976)
8. Bouajjani, A., Echahed, R., Habermehl, P.: On the verification problem of nonregular properties for nonregular processes. In: LICS 1995, pp. 123–133 (1995)
9. Čerāns, K.: Deciding properties of integral relational automata. In: Shamir, E., Abiteboul, S. (eds.) ICALP 1994. LNCS, vol. 820, pp. 35–46. Springer, Heidelberg (1994)
10. Dang, Z., Ibarra, O., San Pietro, P.: Liveness verification of reversal-bounded multicounter machines with a free counter. In: Hariharan, R., Mukund, M., Vinay, V. (eds.) FSTTCS 2001. LNCS, vol. 2245, pp. 132–143. Springer, Heidelberg (2001)
11. Demri, S.: On Selective Unboundedness of VASS. In: INFINITY 2010. EPTCS, vol. 39, pp. 1–15 (2010)
12. Esparza, J.: Decidability and complexity of Petri net problems — an introduction. In: Reisig, W., Rozenberg, G. (eds.) APN 1998. LNCS, vol. 1491, pp. 374–428. Springer, Heidelberg (1998)
13. Finkel, A., Sangnier, A.: Reversal-bounded counter machines revisited. In: Ochmański, E., Tyszkiewicz, J. (eds.) MFCS 2008. LNCS, vol. 5162, pp. 323–334. Springer, Heidelberg (2008)

14. Gurari, E., Ibarra, O.: The complexity of decision problems for finite-turn multicounter machines. In: Even, S., Kariv, O. (eds.) ICALP 1981. LNCS, vol. 115, pp. 495–505. Springer, Heidelberg (1981)
15. Habermehl, P.: On the complexity of the linear-time mu-calculus for Petri nets. In: Azéma, P., Balbo, G. (eds.) ICATPN 1997. LNCS, vol. 1248, pp. 102–116. Springer, Heidelberg (1997)
16. Hague, M., Lin, A.W.: Model checking recursive programs with numeric data types. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 743–759. Springer, Heidelberg (2011)
17. Howell, R., Rosier, L.: An analysis of the nonemptiness problem for classes of reversal-bounded multicounter machines. JCSS 34(1), 55–74 (1987)
18. Ibarra, O.H., Bultan, T., Su, J.: Reachability analysis for some models of infinite-state transition systems. In: Palamidessi, C. (ed.) CONCUR 2000. LNCS, vol. 1877, pp. 183–198. Springer, Heidelberg (2000)
19. Ibarra, O., Su, J., Dang, Z., Bultan, T., Kemmerer, R.: Counter Machines and Verification Problems. TCS 289(1), 165–189 (2002)
20. Ibarra, O.H.: Reversal-bounded multicounter machines and their decision problems. JACM 25(1), 116–133 (1978)
21. Kopczynski, E., To, A.: Parikh Images of Grammars: Complexity and Applications. In: LICS 2010, pp. 80–89. IEEE, Los Alamitos (2010)
22. Laroussinie, F., Meyer, A., Petonnet, E.: Counting LTL. In: TIME 2010, pp. 51–58. IEEE, Los Alamitos (2010)
23. Leroux, J., Point, G.: TaPAS: The Talence Presburger Arithmetic Suite. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 182–185. Springer, Heidelberg (2009)
24. Leroux, J., Sutre, G.: Flat counter automata almost everywhere! In: Peled, D.A., Tsay, Y.-K. (eds.) ATVA 2005. LNCS, vol. 3707, pp. 489–503. Springer, Heidelberg (2005)
25. Lutz, C.: NEXPTIME-complete description logics with concrete domains. ACM ToCL 5(4), 669–705 (2004)
26. de Moura, L., Björner, N.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
27. Papadimitriou, C.: On the complexity of integer programming. JACM 28(4), 765–768 (1981)
28. Presburger, M.: Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In: Comptes Rendus du premier congrès de mathématiciens des Pays Slaves, Warszawa, pp. 92–101 (1930)
29. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 93–107. Springer, Heidelberg (2005)
30. Rackoff, C.: The covering and boundedness problems for vector addition systems. TCS 6(2), 223–231 (1978)
31. Suzuki, N., Jefferson, D.: Verification Decidability of Presburger Array Programs. JACM 27(1), 191–205 (1980)
32. To, A.: Model Checking Infinite-State Systems: Generic and Specific Approaches. Ph.D. thesis, School of Informatics, University of Edinburgh (2010)
33. To, A., Libkin, L.: Algorithmic metatheorems for decidable LTL model checking over infinite systems. In: Ong, L. (ed.) FOSSACS 2010. LNCS, vol. 6014, pp. 221–236. Springer, Heidelberg (2010)
34. Vardi, M., Wolper, P.: Reasoning about infinite computations. I&C 115, 1–37 (1994)

Expressing Polymorphic Types in a Many-Sorted Language

François Bobot[1,2] and Andrei Paskevich[1,2]

[1] LRI, Université Paris-Sud 11, CNRS, Orsay F-91405
[2] INRIA Saclay-Île de France, ProVal, Orsay F-91893

Abstract. In this paper, we study translation from a first-order logic with polymorphic types à la ML (of which we give a formal description) to a many-sorted or one-sorted logic as accepted by mainstream automated theorem provers. We consider a three-stage scheme where the last stage eliminates polymorphic types while adding the necessary "annotations" to preserve soundness, and the first two stages serve to protect certain terms so that they can keep their original unannotated form. This protection allows us to make use of provers' built-in theories and operations. We present two existing translation procedures as sound and complete instances of this generic scheme. Our formulation generalizes over the previous ones by allowing us to protect terms of arbitrary monomorphic types. In particular, we can benefit from the built-in theory of arrays in SMT solvers such as Z3, CVC3, and Yices. The proposed methods are implemented in the Why3 tool and we compare their performance in combination with several automated provers.

1 Introduction

Polymorphic types are a means of abstraction over families of different types; a polymorphic definition or proposition stands for a potentially infinite number of its type-specific instances. Type systems employing polymorphism arise naturally in programming languages and they are a prominent feature of interactive proof assistants such as Coq [17] or Isabelle/HOL [16].

However, a proof task written in a language with polymorphic types is today a difficult subject for automation. This is not because polymorphism handling in a prover is complicated or inefficient *per se*. As was demonstrated by the Alt-Ergo project [3], this only requires a straightforward extension of the unification procedure and does not impose any significant overhead. The fact is, advanced type systems have not yet become mainstream in automated deduction: SMT solvers use many-sorted languages such as SMT-LIB [1], and TPTP provers are content with one-sorted first-order language. Thus, to apply a mainstream prover to a problem expressed in a polymorphic language, we have to translate it into an equivalent monomorphic or even one-sorted problem.

The challenge is not new and a number of solutions is known, ranging from adding per-variable "type guards" (also known as "relativisation of quantifiers", see [12] and [11, Sect. 3.0]), to throughout decoration of terms with their types

C. Tinelli and V. Sofronie-Stokkermans (Eds.): FroCoS 2011, LNAI 6989, pp. 87–102, 2011.

[9,6], to various flavours of type erasure [10,14,11]. The latter method is logically unsound, though adding type annotations can prevent certain unsound inference steps (see [14, Sect. 2.5,2.6] and [11, Sect. 3.1]).

An important feature of a polymorphism encoding method is special treatment of types and operations that are directly handled by provers' built-in decision procedures, e.g., for linear arithmetic or bit-vectors. The idea is to prevent the terms that can be interpreted by a prover from being modified by translation, to preserve their original form [6,11]. In what follows, we call this "type protection" to emphasize that we are interested in terms of particular types.

In this work, we aim to lift (or at least work around) several limitations we perceive in the previous approaches. Firstly, the existing type protection techniques only handle "simple" types, like integers or booleans, but not instances of polymorphic types, like lists of integers or arrays of reals. Yet decision procedures for such "complex" types are implemented in some SMT solvers; for example, Z3 [15], CVC3 [2], and Yices [7] have a built-in support for arrays. Secondly, type protection, as defined in [6], cannot be used to protect finite types such as booleans: given an axiom "every boolean is equal either to 'true' or to 'false'", one can derive that there are only two values in any encoded type, which can easily lead to a contradiction. Thirdly, while translation by type erasure with addition of type arguments to polymorphic symbols [11, Sect. 3.1] is less intrusive and more efficient than full term decoration [6], the former method is unsound and, according to [11], is only applicable in combination with provers that use trigger-based rather than unification-based instantiation. Such a requirement excludes the superposition-based provers and may be difficult to test when a third-party prover is used.

We begin with a formal presentation of first-order logic with polymorphic types (Section 2). In particular, we show that complete monomorphisation is undecidable, that is, we cannot effectively compute a finite set of monomorphic instances of a polymorphic formula F that is equisatisfiable to F. Then we introduce a generic three-stage scheme of polymorphism encoding (Section 3). In this scheme, we start by replacing interpreted polymorphic symbols (such as operations of access and update in arrays) with selected monomorphic instances. The translation proceeds then to type protection, which we consider as a separate transformation, and concludes with polymorphism elimination proper.

We present a sound and complete method of type instantiation with symbol discrimination for the first stage (Section 3.1). Furthermore, we give a generalized formulation of the type protection method from [6], free from the aforesaid restrictions (Section 3.2). As third-stage transformations, we consider full term decoration from [9,6] (Section 3.3) and type erasure with added type annotations from [11, Sect. 3.1] (Section 3.4). We show the latter method to be sound on problems that admit models with infinite domains for every non-protected type and we discuss how this condition can be handled in practice.

We conclude by comparing the described techniques in combination with the SMT solvers Z3, CVC3, and Yices [7] on a set of about 4100 proof obligations in the Why3 tool [4] (Section 4).

2 First-Order Logic with Polymorphic Types

The logic $\mathbf{FOL}_{\mathrm{T}}$, presented below, is an extension of classical first-order many-sorted logic. In $\mathbf{FOL}_{\mathrm{T}}$, types are built from type constants (such as "integer"), type functions (such as "list of"), and type variables that stand for arbitrary monomorphic types. We do not admit quantifiers over type variables, neither in types, nor in formulas: a polymorphic formula is rather seen as a scheme, a potentially infinite conjunction of its monomorphic type instances. In other words, every type variable that occurs in a formula is bound by an implicit prenex universal quantifier. Basically, we use type polymorphism as a convenient way to write a set of polymorphic axioms — say, for lists or arrays — once, instead of copying them for every particular instance of these types.

The principal purpose of $\mathbf{FOL}_{\mathrm{T}}$ is to help specify and prove programs and its type system can be seen as the first-order fragment of the ML type system. The Why3 verification tool [4] is based upon $\mathbf{FOL}_{\mathrm{T}}$ with some extensions such as algebraic types. The papers [6] and [11] work in a similar setting, though the latter employs explicit quantifiers over type variables in logic formulas.

Syntax. We define types as syntactical expressions built from *type constructors* of fixed arity (denoted with capital sans-serif letters) and *type variables* (denoted α, β, γ). For example, β, I, $\mathsf{F}(\mathsf{I}, \gamma)$ are well-formed types. Type constructors of arity 0 are called *type constants*. A type that contains no type variables is called *monomorphic type* or *sort*. A vector of types $\langle T_1, \ldots, T_n \rangle$ is called *type signature*.

A *type substitution* is a mapping from type variables to types. A *monomorphic* type substitution maps every type variable either to itself or to a sort. A type T is said to *match* another type T' whenever there is a type substitution that instantiates T to T'. This notion is trivially extended to type signatures.

We use letters S and T for types, boldface letters $\mathbf{S}$, $\mathbf{T}$ for type signatures, and Greek letters τ, θ, and π for type substitutions. We denote the set of available type variables with $\mathbb{V}_{\mathrm{T}}$, the set of type constructors with $\mathbb{F}_{\mathrm{T}}$, the set of all types built from $\mathbb{V}_{\mathrm{T}}$ and $\mathbb{F}_{\mathrm{T}}$ with $\mathcal{T}(\mathbb{F}_{\mathrm{T}}, \mathbb{V}_{\mathrm{T}})$, and the set of all sorts with $\mathcal{T}(\mathbb{F}_{\mathrm{T}})$. We presume that $\mathbb{V}_{\mathrm{T}}$ is infinite and $\mathbb{F}_{\mathrm{T}}$ contains at least one type constant.

We use traditional first-order terms and formulas, built from variable symbols (denoted u, v, w), function symbols (denoted f, g, h), and predicate symbols (denoted p, q), with the following additions:

- Every term carries an explicit type, e.g.: $w : \mathsf{C}(\mathsf{I})$, $f(u : \alpha, v : \mathsf{L}(\alpha)) : \mathsf{L}(\alpha)$. We denote terms with letters s and t, and, by abuse of notation, we sometimes write the type of a term to the right of the letter: $s : T_1$, $t : T_2$, and so on.
- A *variable* is a variable symbol with a type, and we treat $w : \mathsf{C}(\mathsf{I})$ and $w : \mathsf{C}(\alpha)$ as two distinct variables even though they share the same variable symbol.
- To each function symbol of arity n we assign a type signature of length $n+1$. A term of the form $f(t_1 : T_1, \ldots, t_n : T_n) : T$ is well-formed if and only if the type signature of f matches $\langle T_1, \ldots, T_n, T \rangle$.
- To each predicate symbol of arity n we give a type signature of length n. An atomic formula of the form $p(t_1 : T_1, \ldots, t_n : T_n)$ is well-formed if and only if the type signature of p matches $\langle T_1, \ldots, T_n \rangle$.

- An atomic equality formula of the form $t_1 \approx t_2$ is well-formed if and only if the terms t_1, t_2 have the same type.
- Quantifiers bind variables, i.e., typed variable symbols: $\forall(u:\alpha)\, p(u:\alpha, u:\mathsf{C})$. Here, the first argument of p is bound, but the second one is free.

We treat equality ($\approx$), negation ($\neg$), conjunction ($\wedge$), and the universal quantifier ($\forall$) as logical symbols and we treat disjunction ($\vee$), implication ($\supset$), equivalence ($\equiv$), disequality ($\not\approx$), and the existential quantifier ($\exists$) as abbreviations.

We use letters x, y, z for variables, letters F, G, H for formulas, and Greek letters Γ, Δ for sets of formulas. We denote the (infinite) set of variable symbols with $\mathbb{V}$, the set of function symbols with $\mathbb{F}$, and the set of predicate symbols with $\mathbb{P}$. Given a term or a formula e, the set of type variables occurring in e is denoted $\mathrm{FV_T}(e)$ and the set of free variables of e is denoted $\mathrm{FV}(e)$. If $\mathrm{FV_T}(e)$ is empty, we call e *monomorphic*. If $\mathrm{FV}(e)$ is empty, we call e *closed* or *ground*.

Substitutions, denoted with letters σ and δ, apply to a term or a formula e, replacing free variables with terms of the same type (denoted $e\sigma$). The symbol $\circ$ denotes the composition of two (type) substitutions: $x(\sigma \circ \delta) \triangleq x\sigma\delta$.

Type substitutions apply only to closed formulas and ground terms; also, we require type instantiation to rename every bound variable symbol to some fresh variable symbol. In this way, we avoid variable collisions: for example, the type substitution $[\mathsf{I}/\alpha]$ would not instantiate the formula $\forall(u:\alpha)\forall(u:\mathsf{I})\, p(u:\alpha, u:\mathsf{I})$ to $\forall(u:\mathsf{I})\forall(u:\mathsf{I})\, p(u:\mathsf{I}, u:\mathsf{I})$, but to $\forall(u':\mathsf{I})\forall(u'':\mathsf{I})\, p(u':\mathsf{I}, u'':\mathsf{I})$. In our subsequent examples, we will not use a variable symbol in two different variables in the same formula to avoid confusion.

In what follows, we illustrate our transformations on the following simple polymorphic formula (for the sake of readability, we omit the most obvious type annotations): $\forall(m:\mathsf{M}(\alpha,\mathsf{I}))\,\forall(c:\alpha)\,\mathtt{get}(\mathtt{set}(m,c,6):\mathsf{M}(\alpha,\mathsf{I}),c):\mathsf{I}*7 \approx 42$. Here, the type I represents integers and the type $\mathsf{M}(\alpha,\beta)$ is that of polymorphic α-to-β maps. The function symbol `get` is of type signature $\langle \mathsf{M}(\alpha,\beta), \alpha, \beta\rangle$ and `set` is of type signature $\langle \mathsf{M}(\alpha,\beta), \alpha, \beta, \mathsf{M}(\alpha,\beta)\rangle$.

Satisfiability. Given sets $\mathbb{F}_\mathrm{T}$, $\mathbb{F}$, $\mathbb{P}$, an interpretation $\mathfrak{I}$ is defined by three maps:

- to each sort $S \in \mathcal{T}(\mathbb{F}_\mathrm{T})$, we assign a non-empty *domain* $\mathcal{D}^\mathfrak{I}_S$;
- to each symbol $f \in \mathbb{F}$ and each vector of sorts $\mathbf{S} = \langle S_1, \ldots, S_n, S\rangle$ matched by the type signature of f, we assign a function $f^\mathfrak{I}_\mathbf{S} : \mathcal{D}^\mathfrak{I}_{S_1} \times \cdots \times \mathcal{D}^\mathfrak{I}_{S_n} \to \mathcal{D}^\mathfrak{I}_S$;
- to each symbol $p \in \mathbb{P}$ and each vector of sorts $\mathbf{S} = \langle S_1, \ldots, S_n\rangle$ matched by the type signature of p, we assign a function $p^\mathfrak{I}_\mathbf{S} : \mathcal{D}^\mathfrak{I}_{S_1} \times \cdots \times \mathcal{D}^\mathfrak{I}_{S_n} \to \{\top, \bot\}$, where $\top$ and $\bot$ stand for Boolean constants "true" and "false", respectively.

We call *type valuation* a type substitution that instantiates every type variable in $\mathbb{V}_\mathrm{T}$ to a sort. Given a type valuation π, we call *variable valuation under* π a function that maps every variable $u:T$ to some element of $\mathcal{D}^\mathfrak{I}_{T\pi}$. We simply say *variable valuation* when the implied type valuation is known from the context or in a purely monomorphic setting, where every type is already closed.

Let π be a type valuation and ξ be a variable valuation under π. We evaluate terms and formulas according to the following equalities, where $\mathbf{t}:\mathbf{T}$ stands for

a vector of terms $t_1 : T_1, \ldots, t_n : T_n$ and $\xi[u : T \mapsto a]$ is a valuation that coincides with ξ everywhere except $u : T$, which is mapped to a.

$$\begin{aligned}
\mathfrak{I}_{\pi,\xi}(u : T) &\triangleq \xi(u : T) & \mathfrak{I}_{\pi,\xi}(t_1 \approx t_2) &\triangleq (\mathfrak{I}_{\pi,\xi}(t_1) = \mathfrak{I}_{\pi,\xi}(t_2)) \\
\mathfrak{I}_{\pi,\xi}(f(\mathbf{t} : \mathbf{T})) : T) &\triangleq f^{\mathfrak{I}}_{\langle \mathbf{T}, T \rangle \pi}(\mathfrak{I}_{\pi,\xi}(\mathbf{t})) & \mathfrak{I}_{\pi,\xi}(\neg F) &\triangleq \neg\, \mathfrak{I}_{\pi,\xi}(F) \\
\mathfrak{I}_{\pi,\xi}(p(\mathbf{t} : \mathbf{T})) &\triangleq p^{\mathfrak{I}}_{\mathbf{T}\pi}(\mathfrak{I}_{\pi,\xi}(\mathbf{t})) & \mathfrak{I}_{\pi,\xi}(F \wedge G) &\triangleq \mathfrak{I}_{\pi,\xi}(F) \wedge \mathfrak{I}_{\pi,\xi}(G) \\
& & \mathfrak{I}_{\pi,\xi}(\forall (u : T) F) &\triangleq \bigwedge_{a \in \mathcal{D}^{\mathfrak{I}}_{T\pi}} \mathfrak{I}_{\pi,\xi[u : T \mapsto a]}(F)
\end{aligned}$$

It is easy to see that evaluation of a term or a formula e under $\mathfrak{I}_{\pi,\xi}$ does not depend on the values of π and ξ on (type) variables that do not occur in e. In what follows, when we evaluate closed or monomorphic expressions, we often omit the variable valuation or the type valuation, respectively.

Lemma 1. *For any closed formula F and type valuation π, $\mathfrak{I}_\pi(F) = \mathfrak{I}(F\pi)$.*

As we said above, we treat type variables as implicitly universally quantified at the top of a polymorphic formula. Thus, a closed formula F is *satisfied* by $\mathfrak{I}$ if and only if $\mathfrak{I}_\pi(F)$ is true for every type valuation π. A closed formula is *satisfiable* if and only if it is satisfied by some interpretation, called a *model* of this formula. These definitions are trivially extended to sets of closed formulas. To prove a polymorphic formula G in a polymorphic context Γ, we take a type substitution τ that replaces all type variables in G with fresh type constants and show that the set $\Gamma, \neg G\tau$ is unsatisfiable. Generally speaking, the semantics of polymorphic formulas in $\mathbf{FOL}_{\mathrm{T}}$ is quite similar to that of first-order clauses, where the free variables are also implicitly universally quantified.

On monomorphic terms and formulas, our definitions correspond to the traditional many-sorted logic with disjoint sorts. Moreover, a trivial corollary of Lemma 1 is that F is satisfiable if and only if the set of all monomorphic type instances of F is satisfiable.

Computing monomorphic instances? A polymorphic formula can have infinitely many monomorphic type instances. But can't we find out, in finite time, all sorts that are potentially relevant to the problem and deal with a finite subset of instances, produced just with these sorts? On one hand, this resembles an attempt to pre-compute the relevant ground instances in a set of first-order formulas — a problem well known to be undecidable. On the other hand, type handling does not need to be as hard as proof search in general, and complete monomorphisation is often possible in programming languages (e.g., C++ templates).

Theorem 1. *There is no computable function that maps an arbitrary closed formula F to an equisatisfiable finite set of monomorphic type instances of F (notice that such a set always exists by compactness).*

Proof. It turns out that our type system is expressive enough to encode an undecidable theory, namely, combinatory logic. Consider the following signature:

$$\mathbb{F}_{\mathrm{T}} = \{\, \mathsf{A}(\cdot,\cdot),\ \mathsf{S},\ \mathsf{K} \,\} \quad \mathbb{F} = \{\, \mathtt{A} : \langle \alpha, \beta, \mathsf{A}(\alpha,\beta) \rangle,\ \mathtt{S} : \langle \mathsf{S} \rangle,\ \mathtt{K} : \langle \mathsf{K} \rangle \,\} \quad \mathbb{P} = \{\, \mathtt{R} : \langle \alpha, \beta \rangle \,\}$$

along with five axioms (for brevity, we omit some type annotations):

$$\forall (u:\alpha)\, \forall (v:\beta)\, \forall (w:\gamma)\ ((\mathtt{R}(u,v) \wedge \mathtt{R}(v,w)) \supset \mathtt{R}(u,w))$$

$$\forall (u:\alpha)\, \forall (v:\beta)\, \forall (w:\gamma)\ (\mathtt{R}(u,v) \supset \mathtt{R}(\mathtt{A}(u,w) : \mathsf{A}(\alpha,\gamma), \mathtt{A}(v,w) : \mathsf{A}(\beta,\gamma)))$$

$$\forall (u:\alpha)\, \forall (v:\beta)\, \forall (w:\gamma)\ (\mathtt{R}(u,v) \supset \mathtt{R}(\mathtt{A}(w,u) : \mathsf{A}(\gamma,\alpha), \mathtt{A}(w,v) : \mathsf{A}(\gamma,\beta)))$$

$$\forall (u:\alpha)\, \forall (v:\beta)\ \mathtt{R}(\mathtt{A}(\mathtt{A}(\mathtt{K},u),v) : \mathsf{A}(\mathsf{A}(\mathsf{K},\alpha),\beta),\ u:\alpha)$$

$$\forall (u:\alpha)\, \forall (v:\beta)\, \forall (w:\gamma)\ \mathtt{R}(\mathtt{A}(\mathtt{A}(\mathtt{A}(\mathtt{S},u),v),w) : \mathsf{A}(\mathsf{A}(\mathsf{A}(\mathsf{S},\alpha),\beta),\gamma),\ \mathtt{A}(\mathtt{A}(u,w),\mathtt{A}(v,w)) : \mathsf{A}(\mathsf{A}(\alpha,\gamma),\mathsf{A}(\beta,\gamma)))$$

Here the binary function symbol $\mathtt{A}$ stands for term application, and the binary predicate symbol $\mathtt{R}$ for CL-reducibility. Notice that every ground combinatory term is reflected in its type.

Now, if we were able to compute a finite set of potentially relevant *closed types* for an arbitrary reducibility problem in this theory, this would readily let us decide the problem itself, as we would thus obtain the set of potentially relevant *ground terms*. Since ground reducibility in CL is undecidable, complete monomorphisation in $\mathbf{FOL}_{\mathrm{T}}$ is undecidable, too. □

3 Eliminating Polymorphic Types

Being unable to select just a relevant monomorphic subset of a polymorphic problem, we have to resort to some form of encoding, converting the polymorphic problem to an equisatisfiable monomorphic one. Such conversion inevitably implies merging many types into few sorts or just a single sort. This is undesirable if we target an automated prover equipped with special techniques (decision procedures, unification modulo, etc.) for particular types, such as integers, booleans or arrays. These types ought to be separated from the rest, protected against this "type fusion", expelled from polymorphism in the problem.

To this purpose, we slightly extend our language in order to be able to select the terms that will keep their (monomorphic) type through polymorphism elimination. To every sort S in $\mathcal{T}(\mathbb{F}_{\mathrm{T}})$ we associate a new *protected sort* $\bar{S}$. The use of protected sorts is restricted: a protected sort can appear in the type signature of a symbol or as a type of a term, but it cannot occur under a type constructor or in the range of a type substitution. In other words, the only type that matches a protected sort $\bar{S}$ is $\bar{S}$ itself.

For example, $\mathtt{get}(v : \overline{\mathsf{M}(\mathsf{I},\mathsf{I})}, c : \mathsf{I}) : \mathsf{I}$ is a malformed term, since the type signature of $\mathtt{get}$ is $\langle \mathsf{M}(\alpha,\beta), \alpha, \beta \rangle$ and $\mathsf{M}(\alpha,\beta)$ does not match $\overline{\mathsf{M}(\mathsf{I},\mathsf{I})}$. Similarly, the term $\mathtt{get}(v : \mathsf{M}(\mathsf{I},\bar{\mathsf{I}}), c : \mathsf{I}) : \bar{\mathsf{I}}$ is malformed, because β does not match $\bar{\mathsf{I}}$ and also because $\mathsf{M}(\mathsf{I},\bar{\mathsf{I}})$ is a malformed type expression. However, if we consider a "protected specialization" of $\mathtt{get}$, denoted $\overline{\mathtt{get}}$, with the type signature $\langle \overline{\mathsf{M}(\mathsf{I},\mathsf{I})}, \bar{\mathsf{I}}, \bar{\mathsf{I}} \rangle$, the application $\overline{\mathtt{get}}(v : \overline{\mathsf{M}(\mathsf{I},\mathsf{I})}, c : \bar{\mathsf{I}}) : \bar{\mathsf{I}}$ is a well-formed term.

Concerning interpretation, every protected sort $\bar{S}$ has its proper non-empty domain $\mathcal{D}^{\mathcal{J}}_{\bar{S}}$. As with any type substitution, we restrict type valuations to non-protected sorts. Thus, a set $\{\forall(u:\alpha)\,\forall(v:\alpha)\,u \approx v,\ \exists(a:\bar{\mathrm{I}})\,\exists(b:\bar{\mathrm{I}})\,a \not\approx b\}$ is satisfiable. Indeed, the first formula requires the domain of every non-protected sort to be a singleton, but does not constrain the domains of protected sorts.

Using protected sorts, we can define a general three-stage scheme of encoding of polymorphic formulas, explained below from the end to the beginning.

The final, "type-fusing" stage takes a set of polymorphic formulas with protected sorts and converts it into an equisatisfiable set of monomorphic formulas. A common requirement to the methods on this stage is preservation of terms with protected types: monomorphic protected fragments of the problem, e.g., arithmetic expressions, must be sent to a prover as is. We present two "type-fusing" transformations, Dec and Exp, in Sections 3.3 and 3.4. Both methods have been previously described in the literature [9,10,14,6,11]. Our presentation is more general in that it permits to protect arbitrarily complex monomorphic types, such as "list of integers" or "integer-to-real map". The ability to preserve such sorts is of more than purely theoretical interest: as we have already mentioned, Z3, CVC3, and Yices provide built-in support for access and update operations on integer-indexed arrays.

The intermediate, "type-protecting" stage takes a set of polymorphic formulas without protected sorts and converts it into an equisatisfiable set of formulas with protection. The methods on this stage take as a parameter the set of sorts that we wish to protect; we expect them to put protection over every occurrence of every sort from this set in the problem. We present a type-protecting transformation called Tw in Section 3.2. This method was introduced in [6]; we generalize it to complex sorts.

The first stage can be figuratively called "type-revealing". Even if our type-protecting and type-fusing transformations are not limited to sort constants and can protect arbitrarily complex sorts, say, arrays of integers, we cannot readily benefit from this capacity. In an initial $\mathbf{FOL}_{\mathrm{T}}$-problem, arrays are most probably formalized as a polymorphic type, with premises that apply to arrays of any type and with polymorphic function symbols for access and update. In order to produce interpreted monomorphic operations for Z3, CVC3, or Yices in the end, we must start by replacing, wherever possible, these function symbols with their monomorphic specializations. This is the purpose of the Dis transformation, presented in Section 3.1. We show in Section 4 that this "type revealing" brings a considerable improvement to provers' results.

To fit the page limit, we omit the proofs of our theorems. The reader is referred to the extended technical report [5].

3.1 Symbol Discrimination

The Dis transformation involves producing a sufficient number of type instances of formulas in an initial problem Γ with subsequent discrimination of function and predicate symbols.

Let f be a function symbol of type signature $\mathbf{S}$ in the initial problem Γ. Let τ be a type substitution in the type variables of $\mathbf{S}$. A fresh function symbol f_τ with the type signature $\mathbf{S}\tau$ is called a *specialization* of f. We call f_τ a *monomorphic specialization* if $\mathbf{S}\tau$ is monomorphic. Specializations of predicate symbols are defined in the same way.

Let W be a set of monomorphic specializations of function and predicate symbols in Γ. These are the instances that we want to "reveal" in the problem. The set W is fixed for the rest of this section; the Dis transformation is implicitly parametrized by it.

First of all, the Dis transformation modifies the signature of Γ:

1. For every variable symbol u and type T, we add a new variable symbol u_T.
2. We add every function and predicate symbol from W.

Given an arbitrary type substitution θ, the discriminating transformation $\textsc{Dis}_\theta$ instantiates and converts terms and formulas into the new signature:

1. Given a variable $u : T$, $\textsc{Dis}_\theta(u : T) \triangleq u_T : T\theta$.
2. Consider a term $t = f(t_1 : T_1, \ldots, t_n : T_n) : T$. Let τ be the type substitution that instantiates the type signature of the symbol f to $\langle T_1\theta, \ldots, T_n\theta, T\theta\rangle$. If f_τ belongs to W, then $\textsc{Dis}_\theta(t) \triangleq f_\tau(\textsc{Dis}_\theta(t_1) : T_1\theta, \ldots, \textsc{Dis}_\theta(t_n) : T_n\theta) : T\theta$. Otherwise, $\textsc{Dis}_\theta(t) \triangleq f(\textsc{Dis}_\theta(t_1) : T_1\theta, \ldots, \textsc{Dis}_\theta(t_n) : T_n\theta) : T\theta$.
3. Consider an atomic formula $F = p(t_1 : T_1, \ldots, t_n : T_n)$. Let τ be the type substitution that instantiates the type signature of p to $\langle T_1\theta, \ldots, T_n\theta\rangle$. If p_τ is in W, then $\textsc{Dis}_\theta(F) \triangleq p_\tau(\textsc{Dis}_\theta(t_1) : T_1\theta, \ldots, \textsc{Dis}_\theta(t_n) : T_n\theta)$. Otherwise, $\textsc{Dis}_\theta(F) \triangleq p(\textsc{Dis}_\theta(t_1) : T_1\theta, \ldots, \textsc{Dis}_\theta(t_n) : T_n\theta)$.

Equalities and complex formulas are converted in a natural way:

$$\textsc{Dis}_\theta(t_1 \approx t_2) \triangleq \textsc{Dis}_\theta(t_1) \approx \textsc{Dis}_\theta(t_2) \qquad \textsc{Dis}_\theta(F \wedge G) \triangleq \textsc{Dis}_\theta(F) \wedge \textsc{Dis}_\theta(G)$$
$$\textsc{Dis}_\theta(\neg F) \triangleq \neg\, \textsc{Dis}_\theta(F) \qquad \textsc{Dis}_\theta(\forall x\, F) \triangleq \forall(\textsc{Dis}_\theta(x))\, \textsc{Dis}_\theta(F)$$

Now, let F be a closed formula. The set of monomorphic type substitutions $\Theta(F)$ is defined as follows:

$$\begin{aligned}\Theta(F) \triangleq \{\, \theta \mid\ & F \text{ contains a term } f(t_1 : T_1, \ldots, t_n : T_n) : T \text{ such that} \\ & \theta \text{ only instantiates the variables of } \mathbf{T} = \langle T_1, \ldots, T_n, T\rangle \text{ and} \\ & W \text{ contains a specialization of } f \text{ with the type signature } \mathbf{T}\theta \,\} \\ \cup \{\, \theta \mid\ & F \text{ contains an atomic formula } p(t_1 : T_1, \ldots, t_n : T_n) \text{ such that} \\ & \theta \text{ only instantiates the variables of } \mathbf{T} = \langle T_1, \ldots, T_n\rangle \text{ and} \\ & W \text{ contains a specialization of } p \text{ with the type signature } \mathbf{T}\theta \,\}\end{aligned}$$

We call two monomorphic type substitutions *compatible* if they do not substitute two different sorts for the same type variable. The *union* of two compatible monomorphic type substitutions is their composition (the order is irrelevant). We define $\Theta^\star(F)$ as the closure of $\Theta(F)$ with respect to finite unions of compatible type substitutions. The empty union, i.e., the identity type substitution, also belongs to $\Theta^\star(F)$.

Finally, Dis translates a closed formula F into a set of formulas:

$$\textsc{Dis}(F) \triangleq \{\, \textsc{Dis}_\theta(F) \mid \theta \in \Theta^\star(F) \,\}$$

On our running example, assuming $W = \{\mathtt{get}_{[\mathsf{I}/\alpha,\mathsf{I}/\beta]}, \mathtt{set}_{[\mathsf{I}/\alpha,\mathsf{I}/\beta]}\}$, Dis produces the following two formulas:

$$\forall(m_{\mathsf{M}(\alpha,\mathsf{I})} : \mathsf{M}(\alpha,\mathsf{I}))\, \forall(c_\alpha : \alpha)\ \mathtt{get}(\mathtt{set}(m_{\mathsf{M}(\alpha,\mathsf{I})}, c_\alpha, 6), c_\alpha) * 7 \approx 42$$
$$\forall(m_{\mathsf{M}(\alpha,\mathsf{I})} : \mathsf{M}(\mathsf{I},\mathsf{I}))\, \forall(c_\alpha : \mathsf{I})\ \mathtt{get}_{[\mathsf{I}/\alpha,\mathsf{I}/\beta]}(\mathtt{set}_{[\mathsf{I}/\alpha,\mathsf{I}/\beta]}(m_{\mathsf{M}(\alpha,\mathsf{I})}, c_\alpha, 6), c_\alpha) * 7 \approx 42$$

The new symbol $\mathtt{get}_{[\mathsf{I}/\alpha,\mathsf{I}/\beta]}$ has the monomorphic type signature $\langle \mathsf{M}(\mathsf{I},\mathsf{I}), \mathsf{I}, \mathsf{I}\rangle$.

Lemma 2. *Let θ be a type substitution, t a term of type T, and F a formula. Then* $\textsc{Dis}_\theta(t)$ *is a well-formed term of type $T\theta$ and* $\textsc{Dis}_\theta(F)$ *is a well-formed formula such that* $\mathrm{FV}(\textsc{Dis}_\theta(F)) = \{\textsc{Dis}_\theta(x) \mid x \in \mathrm{FV}(F)\}$ *and* $\mathrm{FV}_\mathrm{T}(\textsc{Dis}_\theta(F)) = \mathrm{FV}_\mathrm{T}(F) \backslash \mathrm{dom}(\theta)$.

Theorem 2. *A set of closed formulas Γ is equisatisfiable to* $\textsc{Dis}(\Gamma)$.

The definition of Dis can be generalized to a case where W admits polymorphic specializations. This requires W to be closed with respect to unification of type signatures, so that we can always choose the most refined specialization symbol during discrimination. The substitutions in the set $\Theta(F)$ must be considered modulo renaming of type variables in the signatures of specialization symbols. Finally, the union of two substitutions would be their most general common refinement. However, since our transformations target monomorphic theorem provers, we find this generalization of lesser practical interest and do not pursue it in this paper.

3.2 Twin Sorts

The Tw transformation converts a set of formulas into an equisatisfiable set with protected sorts. It applies a pair of conversion functions to pass, wherever necessary, from a protected sort to a non-protected one and vice versa.

Let U be a set of sorts that we want to preserve across our type-eliminating transformations. The set U is fixed for the rest of the section and the Tw transformation is parametrized by it. Given a type T, the transformed type $[T]$ is $\overline{T}$ if $T \in U$, and T otherwise. Then Tw modifies the signature of a transformed theory as follows:

1. We replace every function symbol f of type signature $\langle S_1, \ldots, S_n, S\rangle$ with a symbol $\bar{f}$ of type signature $\langle [S_1], \ldots, [S_n], [S]\rangle$.
2. We replace every predicate symbol p of type signature $\langle S_1, \ldots, S_n\rangle$ with a symbol $\bar{p}$ of type signature $\langle [S_1], \ldots, [S_n]\rangle$.
3. For every sort $T \in U$, we add a pair of "bridge" function symbols $\mathtt{to}_T : \langle \overline{T}, T\rangle$ and $\mathtt{from}_T : \langle T, \overline{T}\rangle$.

Then we convert terms and atomic formulas into the new signature. Our aim is to forbid a polymorphic type in a symbol's type signature being instantiated into a type from U. Whenever such instantiation takes place, a bridge function is applied. In more precise terms:

1. Given a variable $u : T$, $\mathrm{Tw}(u : T) \triangleq u : [T]$.
2. Consider a term $t = f(t_1 : T_1, \ldots, t_n : T_n) : T$ and let $\langle S_1, \ldots, S_n, S \rangle$ be the type signature of f.

$$\text{For every } t_i,\ t'_i \triangleq \begin{cases} \mathtt{to}_{T_i}(\mathrm{Tw}(t_i)) : T_i & \text{if } T_i \in U \text{ and } S_i \notin U, \\ \mathrm{Tw}(t_i) & \text{if } T_i \notin U \text{ or } S_i \in U. \end{cases}$$

$$\text{Then } \mathrm{Tw}(t) \triangleq \begin{cases} \bar{f}(t'_1, \ldots, t'_n) : [T] & \text{if } T \notin U \text{ or } S \in U, \\ \mathtt{from}_T(\bar{f}(t'_1, \ldots, t'_n) : T) : \overline{T} & \text{if } T \in U \text{ and } S \notin U. \end{cases}$$

3. Consider a formula $p(t_1 : T_1, \ldots, t_n : T_n)$ and let $\langle S_1, \ldots, S_n \rangle$ be the type signature of p. For every argument t_i, we define t'_i as in the previous case. Then, $\mathrm{Tw}(p(t_1 : T_1, \ldots, t_n : T_n)) \triangleq \bar{p}(t'_1, \ldots, t'_n)$.

Equalities and complex formulas are converted in a natural way:

$$\mathrm{Tw}(t_1 \approx t_2) \triangleq \mathrm{Tw}(t_1) \approx \mathrm{Tw}(t_2) \qquad \mathrm{Tw}(F \wedge G) \triangleq \mathrm{Tw}(F) \wedge \mathrm{Tw}(G)$$
$$\mathrm{Tw}(\neg F) \triangleq \neg\, \mathrm{Tw}(F) \qquad \mathrm{Tw}(\forall x\, F) \triangleq \forall (\mathrm{Tw}(x))\, \mathrm{Tw}(F)$$

Finally, we convert the formulas in Γ and add axioms for the bridge functions:

$$\begin{aligned} \mathrm{Tw}(\Gamma) \triangleq & \{\, \mathrm{Tw}(F) \mid F \in \Gamma \,\} \\ & \cup \{\, \forall (v : \overline{T})\ \mathtt{from}_T(\mathtt{to}_T(v : \overline{T})) \approx v : \overline{T} \mid T \in U \,\} \\ & \cup \{\, \forall (u : T)\ \mathtt{to}_T(\mathtt{from}_T(u : T)) \approx u : T \mid T \in U \,\} \end{aligned}$$

Assuming $U = \{\mathsf{I}\}$, the running example is transformed as follows. Notice that `6`, `7`, and `42` have the type $\overline{\mathsf{I}}$ and the type signature of $*$ is $\langle \overline{\mathsf{I}}, \overline{\mathsf{I}}, \overline{\mathsf{I}} \rangle$.

$$\forall (m : \mathsf{M}(\alpha, \mathsf{I}))\, \forall (c : \alpha)\ \mathtt{from}_{\mathsf{I}}(\mathtt{get}(\mathtt{set}(m, c, \mathtt{to}_{\mathsf{I}}(\mathtt{6}) : \mathsf{I}), c) : \mathsf{I}) * \mathtt{7} \approx \mathtt{42}$$

Lemma 3. *For every term t of type T, $\mathrm{Tw}(t)$ is a well-formed term of type $[T]$, and for every formula F, $\mathrm{Tw}(F)$ is a well-formed formula with the same free variables (modulo conversion of their types) and type variables.*

Theorem 3. *A set of closed formulas Γ is equisatisfiable to $\mathrm{Tw}(\Gamma)$.*

3.3 Decorated Terms

The Dec transformation converts a polymorphic problem with protected sorts into an equisatisfiable monomorphic problem. Roughly speaking, in order to preserve type information, it decorates every term with its type, which itself is transformed to a term of a special sort.

First of all, we introduce three fresh sort constants U, D, and T. The first one is assigned to undecorated terms, the second one to decorated terms, and the third one to the terms representing types. To transform the type signatures of function and predicate symbols, we use the following operations on types:

$$[T]^- \triangleq \begin{cases} T & \text{if } T \text{ is protected,} \\ \mathsf{D} & \text{otherwise} \end{cases} \qquad [T]^+ \triangleq \begin{cases} T & \text{if } T \text{ is protected,} \\ \mathsf{U} & \text{otherwise} \end{cases}$$

Now, the signature of the resulting theory is defined as follows:

1. The set of type constructors is extended with U, D, T.
2. We replace every function symbol f of type signature $\langle S_1, \ldots, S_n, S\rangle$ with a symbol $\hat{f}$ with the monomorphic type signature $\langle [S_1]^-, \ldots, [S_n]^-, [S]^+\rangle$.
3. We replace every predicate symbol p of type signature $\langle S_1, \ldots, S_n\rangle$ with a symbol $\hat{p}$ with the monomorphic type signature $\langle [S_1]^-, \ldots, [S_n]^-\rangle$.
4. For every variable symbol u and type T, we add a new variable symbol u_T.
5. For every type variable $\alpha \in \mathbb{V}_\mathrm{T}$, we add a new variable symbol v^α.
6. For every type constructor $\mathsf{F} \in \mathbb{F}_\mathrm{T}$, we add a new function symbol $\mathtt{F}$ of the same arity and with type signature $\langle \mathsf{T}, \ldots, \mathsf{T}, \mathsf{T}\rangle$.
7. We add a new "decoration" function symbol $\mathtt{deco}{:}\langle \mathsf{T}, \mathsf{U}, \mathsf{D}\rangle$.

The Dec transformation applies to non-protected types, translating them to terms of type T:

$$\textsc{Dec}(\alpha) \triangleq v^\alpha : \mathsf{T} \qquad \textsc{Dec}(\mathsf{F}(T_1, \ldots, T_n)) \triangleq \mathtt{F}(\textsc{Dec}(T_1), \ldots, \textsc{Dec}(T_n)) : \mathsf{T}$$

In the next definition, $\mathbf{t}$ stands for a vector of terms, $\bar{S}$ for a protected sort, and T for a non-protected type. The Dec transformation applies to terms:

$$\begin{aligned} \textsc{Dec}(u : \bar{S}) &\triangleq u_{\overline{S}} : \bar{S} \\ \textsc{Dec}(u : T) &\triangleq \mathtt{deco}(\textsc{Dec}(T), u_T : \mathsf{U}) : \mathsf{D} \\ \textsc{Dec}(f(\mathbf{t}) : \bar{S}) &\triangleq \hat{f}(\textsc{Dec}(\mathbf{t})) : \bar{S} \\ \textsc{Dec}(f(\mathbf{t}) : T) &\triangleq \mathtt{deco}(\textsc{Dec}(T), \hat{f}(\textsc{Dec}(\mathbf{t})) : \mathsf{U}) : \mathsf{D} \end{aligned}$$

and formulas (here, $\{\alpha_1, \ldots, \alpha_m\} = \mathrm{FV}_\mathrm{T}(H)$):

$$\begin{aligned} \textsc{Dec}(p(\mathbf{t})) &\triangleq \hat{p}(\textsc{Dec}(\mathbf{t})) & \textsc{Dec}(\neg F) &\triangleq \neg\, \textsc{Dec}(F) \\ \textsc{Dec}(t_1 \approx t_2) &\triangleq \textsc{Dec}(t_1) \approx \textsc{Dec}(t_2) & \textsc{Dec}(\forall(u : \bar{S})F) &\triangleq \forall(u_{\overline{S}} : \bar{S})\, \textsc{Dec}(F) \\ \textsc{Dec}(F \wedge G) &\triangleq \textsc{Dec}(F) \wedge \textsc{Dec}(G) & \textsc{Dec}(\forall(u : T)F) &\triangleq \forall(u_T : \mathsf{U})\, \textsc{Dec}(F) \end{aligned}$$

$$\textsc{Dec}^\circ(H) \triangleq \forall(v^{\alpha_1} : \mathsf{T}) \ldots \forall(v^{\alpha_m} : \mathsf{T})\, \textsc{Dec}(H)$$

On the running example, assuming $U = \{\mathsf{I}\}$, the transformations Tw and $\textsc{Dec}^\circ$ produce the following monomorphic formula:

$$\begin{aligned} \forall(v^\alpha : \mathsf{T})\, \forall(m_{\mathsf{M}(\alpha,\mathsf{I})} : \mathsf{U})\, \forall(c_\alpha : \mathsf{U})\, \mathtt{from}_\mathsf{I}\big(\mathtt{deco}\big(\mathtt{I}, \mathtt{get}(\mathtt{deco}(\mathtt{M}(v^\alpha, \mathtt{I}), \\ \mathtt{set}(\mathtt{deco}(\mathtt{M}(v^\alpha, \mathtt{I}), m_{\mathsf{M}(\alpha,\mathsf{I})}), \mathtt{deco}(v^\alpha, c_\alpha), \mathtt{deco}(\mathtt{I}, \mathtt{to}_\mathsf{I}(6)))), \\ \mathtt{deco}(v^\alpha, c_\alpha))\big)\big) * 7 \approx 42 \end{aligned}$$

The second axiom of bridge functions $\mathtt{to_l}$ and $\mathtt{from_l}$ becomes

$$\forall(u_\mathsf{l} : \mathsf{U})\, \mathtt{deco}(\mathtt{I}, \mathtt{to_l}(\mathtt{from_l}(\mathtt{deco}(\mathtt{I}, u_\mathsf{l})))) \approx \mathtt{deco}(\mathtt{I}, u_\mathsf{l})$$

Due to the outer application of deco on the both sides of equality, our translation is sound even when we protect finite types, such as booleans. Without this additional decoration (as in [6, Eq. (8)]), the finiteness of a protected sort implies the finiteness of the whole sort U.

Lemma 4. *For every term t of type T,* $\textsc{Dec}(t)$ *is a well-formed monomorphic term of type* $[T]^-$*. For every formula F,* $\textsc{Dec}(F)$ *is a well-formed monomorphic formula. Also,* $\mathrm{FV}(\textsc{Dec}(t)) = \{u_T : [T]^+ \mid u : T \in \mathrm{FV}(t)\} \cup \{v^\alpha : \mathsf{T} \mid \alpha \in \mathrm{FV_T}(t)\}$ *and* $\mathrm{FV}(\textsc{Dec}(F)) = \{u_T : [T]^+ \mid u : T \in \mathrm{FV}(F)\} \cup \{v^\alpha : \mathsf{T} \mid \alpha \in \mathrm{FV_T}(F)\}$*. For every closed formula F,* $\textsc{Dec}^\circ(F)$ *is a well-formed closed monomorphic formula.*

Theorem 4. *A set of closed formulas with protected sorts Γ is satisfiable if and only if* $\textsc{Dec}^\circ(\Gamma)$ *is satisfiable.*

3.4 Explicit Polymorphism

The Exp transformation is similar to Dec except that instead of attaching an explicit type annotation to every term, we add type-representing arguments to polymorphic symbols. This allows for much lighter modifications in the original problem. However, the method is only sound on problems that admit a model where every non-protected sort has an infinite domain.

We introduce fresh sort constants U and T. The first one replaces non-protected types and the second one, as in Dec, is the sort of type-representing terms. For any type T, we define $[T]$ to be T if T is protected, and U otherwise. Then Exp modifies the signature of a transformed theory in the following way:

1. The set of type constructors is extended with U and T.
2. Let f be a function symbol of signature $\mathbf{S} = \langle S_1, \ldots, S_n, S\rangle$ and $\alpha_1, \ldots, \alpha_r$ be the free type variables of $\mathbf{S}$. We replace f with a function symbol $\hat{f}$ of arity $r + n$ with monomorphic type signature $\langle \mathsf{T}, \ldots, \mathsf{T}, [S_1], \ldots, [S_n], [S]\rangle$.
3. Let p be a predicate symbol of signature $\mathbf{S} = \langle S_1, \ldots, S_n\rangle$ and $\alpha_1, \ldots, \alpha_r$ be the free type variables of $\mathbf{S}$. We replace p with a predicate symbol $\hat{p}$ of arity $r + n$ with monomorphic type signature $\langle \mathsf{T}, \ldots, \mathsf{T}, [S_1], \ldots, [S_n]\rangle$.
4. For every variable symbol u and type T, we add a new variable symbol u_T.
5. For every type variable $\alpha \in \mathbb{V}_\mathrm{T}$, we add a new variable symbol v^α.
6. For every type constructor $\mathsf{F} \in \mathbb{F}_\mathrm{T}$, we add a new function symbol F of the same arity and with type signature $\langle \mathsf{T}, \ldots, \mathsf{T}, \mathsf{T}\rangle$.

The Exp transformation applies to non-protected types, translating them to terms of type T, exactly as Dec:

$$\textsc{Exp}(\alpha) \triangleq v^\alpha : \mathsf{T} \qquad \textsc{Exp}(\mathsf{F}(T_1, \ldots, T_n)) \triangleq \mathtt{F}(\textsc{Exp}(T_1), \ldots, \textsc{Exp}(T_n)) : \mathsf{T}$$

The Exp transformation applies to terms and formulas. In the definition below, $\mathbf{t}$ stands for a list of terms; $\alpha_1, \ldots, \alpha_r$ are the type variables of the type signature

of f and p; the type signature of f and p is instantiated with a type substitution τ; and $\beta_1, \ldots, \beta_m$ are the type variables of H:

$$\begin{aligned}
\textsc{Exp}(u:T) &\triangleq u_T:[T] \\
\textsc{Exp}(f(\mathbf{t}):T) &\triangleq \hat{f}(\textsc{Exp}(\alpha_1\tau), \ldots, \textsc{Exp}(\alpha_r\tau), \textsc{Exp}(\mathbf{t})):[T] \\
\textsc{Exp}(p(\mathbf{t})) &\triangleq \hat{p}(\textsc{Exp}(\alpha_1\tau), \ldots, \textsc{Exp}(\alpha_r\tau), \textsc{Exp}(\mathbf{t})) \\
\textsc{Exp}(t_1 \approx t_2) &\triangleq \textsc{Exp}(t_1) \approx \textsc{Exp}(t_2) \\
\textsc{Exp}(\neg F) &\triangleq \neg\, \textsc{Exp}(F) \\
\textsc{Exp}(F \wedge G) &\triangleq \textsc{Exp}(F) \wedge \textsc{Exp}(G) \\
\textsc{Exp}(\forall x\, F) &\triangleq \forall(\textsc{Exp}(x))\, \textsc{Exp}(F) \\
\textsc{Exp}^\circ(H) &\triangleq \forall(v^{\beta_1}:\mathsf{T}) \ldots \forall(v^{\beta_m}:\mathsf{T})\, \textsc{Exp}(H)
\end{aligned}$$

On the running example, assuming $U = \{\mathsf{I}\}$, the transformations Tw and $\textsc{Exp}^\circ$ produce the following formula:

$$\forall(v^\alpha:\mathsf{T})\, \forall(m_{\mathsf{M}(\alpha,\mathsf{I})}:\mathsf{U})\, \forall(c_\alpha:\mathsf{U})\, \mathtt{from}_{\mathsf{I}}(\mathtt{get}(v^\alpha, \mathtt{I}, \mathtt{set}(v^\alpha, \mathtt{I}, m_{\mathsf{M}(\alpha,\mathsf{I})}, c_\alpha, \mathtt{to}_{\mathsf{I}}(6)), c_\alpha)) * 7 \approx 42$$

Lemma 5. *For every term t of type T, $\textsc{Exp}(t)$ is a well-formed monomorphic term of type $[T]$. For every formula F, $\textsc{Exp}(F)$ is a well-formed monomorphic formula. Also, $\mathrm{FV}(\textsc{Exp}(t)) = \{u_T:[T] \mid u:T \in \mathrm{FV}(t)\} \cup \{v^\alpha:\mathsf{T} \mid \alpha \in \mathrm{FV}_{\mathrm{T}}(t)\}$ and $\mathrm{FV}(\textsc{Exp}(F)) = \{u_T:[T] \mid u:T \in \mathrm{FV}(F)\} \cup \{v^\alpha:\mathsf{T} \mid \alpha \in \mathrm{FV}_{\mathrm{T}}(F)\}$. Finally, for every closed formula F, $\textsc{Exp}^\circ(F)$ is a well-formed closed monomorphic formula.*

Theorem 5. *Let Γ be a set of closed formulas with protected sorts. If Γ is satisfiable so that every non-protected sort has an infinite domain in the model, then $\textsc{Exp}^\circ(\Gamma)$ is satisfiable. Conversely, if $\textsc{Exp}^\circ(\Gamma)$ is satisfiable then Γ is satisfiable.*

From a practical point of view, the soundness part of Theorem 5 is not comforting. Given a $\mathbf{FOL}_{\mathrm{T}}$-problem Γ, we cannot effectively decide which sorts admit infinite models and which do not (one can postulate a bijection between a given sort and the domain of a partial-recursive function). A practical way out could consist in a small language extension: for every type/sort, we specify explicitly whether it is finite or infinite. We proceed from the assumption that the author of any given problem knows the intended model of every type.

In Why3 [4], every type is declared either as abstract or algebraic (i.e., a sum of products). We postulate that abstract types are all infinite and we analyse the definitions of algebraic types to find out which of their monomorphic instances admit infinite models. For example, given the standard algebraic definitions of booleans (B), lists ($\mathsf{L}(\alpha)$), and pairs ($\mathsf{P}(\alpha, \beta)$), we can conclude that the sorts B and $\mathsf{P}(\mathsf{B}, \mathsf{B})$ are finite and I, $\mathsf{L}(\mathsf{B})$, and $\mathsf{P}(\mathsf{I}, \mathsf{B})$ are infinite.

Once we know the finite sorts, can we transform a problem to eliminate them, so that Exp (or a similar method) can be applied? Meng and Paulson propose to filter out the premises implying the finiteness of sorts [14, Sect. 2.8]; however,

we need an infallible filter to ensure the soundness of type erasure. We have implemented an alternative solution which consists in putting a special "projection" function $\mathtt{proj}_T$ over every variable and function symbol of a finite type T. Thus, the premise $\forall(x:\mathsf{B})(x \approx \mathtt{True} \vee x \approx \mathtt{False})$ becomes $\forall(x:\mathsf{B})(\mathtt{proj}_{\mathsf{B}}(x) \approx \mathtt{proj}_{\mathsf{B}}(\mathtt{True}) \vee \mathtt{proj}_{\mathsf{B}}(x) \approx \mathtt{proj}_{\mathsf{B}}(\mathtt{False}))$ and the domain of B does not need to be finite anymore, as we confine ourselves to the range of $\mathtt{proj}_{\mathsf{B}}$. This method is still potentially unsound, as one can state the finiteness of a sort with a polymorphic axiom, where no projection would apply. Precisely, let $\mathtt{isUnit}:\langle\alpha\rangle$ be a unary predicate. Then the formulas $\forall(x:\alpha)(\mathtt{isUnit}(x) \supset \forall(y:\alpha)(y \approx x))$ and $\forall(x:\mathsf{A})\,\mathtt{isUnit}(x)$ imply that the sort A has a single inhabitant. Today, we know of no way to use Exp soundly on polymorphic problems with finite sorts.

The last remark to make is that Exp provides a path towards one-sorted languages. Indeed, in a monomorphic setting, Theorem 5 comes to: "if every sort admits an infinite domain, then we can safely erase the sort annotations". Thus, if we want to use a TPTP prover such as Vampire or SPASS, we start by translating a proof task to the many-sorted language, using any of the methods described above. Then we eliminate the protected finite sorts (if any) using projections; in absence of polymorphism, this is a sound and complete transformation. And finally, we apply Exp assuming that all types are non-protected, which amounts to simply erasing all sorts.

4 Experiments and Conclusion

In our experiments, we wanted to compare the impact of different "paths" of polymorphism encoding on the performance of three well-established SMT solvers. We add the classical type encoding technique with per-variable "type guards" [12]. Our implementation of this method (denoted Grd below) closely follows the description given in [11, Sect. 3.0].

We run our tests on 4123 verification conditions generated by the Why platform from 166 programs, which originate from Caduceus [8], Jessie [13], or directly from Why. Translated tasks were sent to Z3, CVC3, and Yices with a time limit of 60 seconds. On the whole, 3993 proof obligations were proved by at least one prover. The initial Why3 files and our results are available at `http://why3.lri.fr/download/polyfol_encoding.tar.gz`.

We have tested the encodings Tw+Dec, Tw+Exp, and Tw+Grd, both with and without Dis. In the latter case, these methods correspond to what is described in [6] and [11]; the set U of sorts to protect in Tw is set to contain only integers and reals, which are natively supported by the three provers. In presence of Dis, we put in the set W every monomorphic specialization that occurs in the goal formula along with the specializations of access and update operations on every monomorphic array type in the goal; we also protect every sort in the goal (as well as integers and reals). This configuration of Dis and Tw gives better results comparing to other configurations that we tried, e.g., collect the specializations and the sorts to protect from the whole proof task.

Z3 (3809)	Tw+Grd		Tw+Exp		Tw+Dec		Dis+Tw+Grd		Dis+Tw+Exp	
Dis+Tw+Dec	+203	−36	+20	−49	+66	−37	+18	−5	+26	−30
Dis+Tw+Exp	+191	−20	+13	−38	+63	−30	+35	−18		
Dis+Tw+Grd	+195	−41	+11	−53	+59	−43				
Tw+Dec	+157	−19	+15	−73						
Tw+Exp	+211	−15								

CVC3 (3756)	Tw+Grd		Tw+Exp		Tw+Dec		Dis+Tw+Grd		Dis+Tw+Exp	
Dis+Tw+Dec	+269	−20	+0	−26	+84	−19	+66	−4	+0	−6
Dis+Tw+Exp	+272	−17	+0	−20	+88	−17	+69	−1		
Dis+Tw+Grd	+204	−17	+1	−89	+46	−43				
Tw+Dec	+188	−4	+0	−91						
Tw+Exp	+275	−0								

Yices (3717)	Tw+Grd		Tw+Exp		Tw+Dec		Dis+Tw+Grd		Dis+Tw+Exp	
Dis+Tw+Dec	+882	−6	+13	−276	+379	−79	+204	−2	+3	−272
Dis+Tw+Exp	+1149	−4	+39	−33	+574	−5	+472	−1		
Dis+Tw+Grd	+684	−10	+6	−471	+241	−143				
Tw+Dec	+577	−1	+5	−568						
Tw+Exp	+1140	−1								

Our results are given in the table above. To the right of the prover's name, we put the number of goals proved by at least one encoding method. In every cell we specify the number of goals proved by one encoding but not by the other one. For example, with CVC3, the encoding by Dis+Tw+Dec allows us to prove 84 goals that were not proved by Tw+Dec. On the other hand, with Tw+Dec, CVC3 proves 19 goals that were not proved with Dis+Tw+Dec.

On the average, symbol discrimination increases the number of premises by a factor of 1.8 (ranging from 1 to 10 on some examples). Nevertheless, adding the Dis phase allows us to prove more goals in every case except for Z3 and CVC3 with Exp. In particular, the Grd transformation is remarkably helped by Dis. Apart from the possibility to use the built-in support for arrays, the effectiveness of Dis is also explained by the fact that we protect the sorts that occur in the selected monomorphic specializations. Thus, the new premises generated by Dis are not only instantiated to the relevant sorts, they are also liberated from decorations imposed by the third, type-fusing, stage. This effect is less important in the case of Exp, because this transformation, unlike Dec and Grd, adds very little clutter to the encoded formulas in the first place.

Also notice that type protection, Tw, is crucially important: if we protect no types at all, we prevent provers from using their built-in theories, and the total number of goals proved (using only Exp, Dec, or Grd) drops to 1861.

The comparison between Exp, Dec, and Grd shows that Exp is generally more efficient than Dec which in its turn is more efficient than Grd. This is quite different from the results given in [11], where Exp and Grd have roughly the same performance. We have not yet identified whether this discrepancy comes from the difference in our test cases or in our implementations.

Conclusion. In the present paper, we described first-order logic with polymorphic types and introduced generic notions to define and reason about practical

methods of polymorphism elimination. Using these notions, we generalized and proved two translation techniques known from literature. We also proposed to combine type protection with symbol discrimination. As our experiments show, this improves the performance of automated proof search and allows us to use built-in theories of complex types, such as arrays, in SMT solvers. One interesting problem we would like to resolve in the future is protection of polymorphic types, allowing to merge all monomorphic instances of a given complex type in a single protected sort. We also believe that better heuristics to choose the sets W and U can be devised, and further experiments are in order.

References

1. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: Version 2.0. Tech. rep., Department of Computer Science, The University of Iowa (2010)
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. Bobot, F., Conchon, S., Contejean, E., Lescuyer, S.: Implementing polymorphism in SMT solvers. In: SMT 2008. ACM ICPS, vol. 367, pp. 1–5 (2008)
4. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Why3: Shepherd your herd of provers. In: Boogie 2011 (co-loc. with CADE-23), Wrocław, Poland (August 2011)
5. Bobot, F., Paskevich, A.: Expressing polymorphic types in a many-sorted language (extended version) (July 2011), http://hal.inria.fr/inria-00591414/en/
6. Couchot, J.F., Lescuyer, S.: Handling polymorphism in automated deduction. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 263–278. Springer, Heidelberg (2007)
7. Dutertre, B., de Moura, L.: The YICES SMT solver. Tech. rep., SRI International (2006)
8. Filliâtre, J.C., Marché, C.: Multi-prover verification of C programs. In: Davies, J., Schulte, W., Barnett, M. (eds.) ICFEM 2004. LNCS, vol. 3308, pp. 15–29. Springer, Heidelberg (2004)
9. Hurd, J.: An LCF-style interface between HOL and first-order logic. In: Voronkov, A. (ed.) CADE 2002. LNCS (LNAI), vol. 2392, pp. 134–138. Springer, Heidelberg (2002)
10. Hurd, J.: First-order proof tactics in higher-order logic theorem provers. In: Design and Application of Strategies/Tactics in Higher Order Logics. NASA Technical Report NASA/CP-2003-212448, pp. 56–68 (2003)
11. Leino, K.R.M., Rümmer, P.: A polymorphic intermediate verification language: Design and logical encoding. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 312–327. Springer, Heidelberg (2010)
12. Manzano, M.: Extensions of First-Order Logic, Cambridge Tracts in Theoretical Computer Science, vol. 19. Cambridge University Press, Cambridge (1996)
13. Marché, C., Moy, Y.: Jessie plug-in (2010), http://frama-c.com/jessie.html
14. Meng, J., Paulson, L.C.: Translating higher-order clauses to first-order clauses. Journal of Automated Reasoning 40(1), 35–60 (2008)
15. 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)
16. Nipkow, T., Paulson, L.C., Wenzel, M.T.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002)
17. The Coq Development Team: The Coq Proof Assistant Reference Manual – Version V8.0 (2004), http://coq.inria.fr

A Combination of Rewriting and Constraint Solving for the Quantifier-Free Interpolation of Arrays with Integer Difference Constraints

Roberto Bruttomesso[1], Silvio Ghilardi[2], and Silvio Ranise[3]

[1] Università della Svizzera Italiana, Lugano, Switzerland
[2] Università degli Studi di Milano, Milan, Italy
[3] FBK (Fondazione Bruno Kessler), Trento, Italy

Abstract. The use of interpolants in model checking is progressively gaining importance. The application of encodings based on the theory of arrays, however, is limited by the impossibility of deriving quantifier-free interpolants in general. To overcome this problem, we have recently proposed a quantifier-free interpolation solver for a natural variant of the theory of arrays. However, arrays are usually combined with fragments of arithmetic over indexes in applications, especially those related to software verification. In this paper, we propose a quantifier-free interpolation solver for the variant of arrays considered in previous work when combined with integer difference logic over indexes.

1 Introduction

Arrays are essential data-structures in computer science. The problem of verifying functional correctness of software and hardware components using symbolic model-checking techniques often boils down to the problem of checking properties over arrays and arithmetic, expressed as quantifier-free first order logic formulæ. Consider for example the following pseudo-code fragment

$$\begin{array}{l}\text{for (int } i = 0 \text{ ; } i \leq n-1 \text{ ; } i = i+1 \text{)}\\ \quad \text{if (} a[i] > a[i+1] \text{)}\\ \quad\quad \text{swap(} a[i], a[i+1] \text{);}\end{array}$$

This loop is used, e.g., in bubble-sort to move the maximum element in the range $[0, n]$ of the array a to position n. It thus satisfies the postcondition

$$\forall i.\ 0 \leq i \leq n-1 \implies a[i] \leq a[n].$$

A possible approach to model-check such property can be established by taking its negation $(\exists i.\ 0 \leq i \leq n-1 \ \wedge\ a[i] > a[n])$, which is an "unsafety condition", and by running a symbolic reachability procedure. State-of-the-art methods for reachability are based on an abstraction-refinement loop, where the refinement phase is handled by means of the computation of interpolants [10].

In order to apply this method it is necessary to provide procedure that computes quantifier-free interpolants for unsatisfiable quantifier-free formulæ in the

C. Tinelli and V. Sofronie-Stokkermans (Eds.): FroCoS 2011, LNAI 6989, pp. 103–118, 2011.

theories under consideration. For instance a symbolic encoding of our example above would be naturally defined in the combination of the theory of arrays ($\mathcal{AX}$) and integer difference logic ($\mathcal{IDL}$). However it is known that, already in $\mathcal{AX}$, quantifier-free interpolants cannot be produced in general.

In a recent work [5], we have shown an extension of $\mathcal{AX}$ with a further functional symbol `diff`, called $\mathcal{AX}_{\texttt{diff}}$, in which quantifier-free interpolants can be computed. In this paper, we extend that result by augmenting $\mathcal{AX}_{\texttt{diff}}$ to a theory $\mathcal{AX}_{\texttt{diff}} \cup \mathcal{IDL}$ for which quantifier-free interpolants can also be computed. In particular, we achieve this result via an ad-hoc combination of the procedure for $\mathcal{AX}_{\texttt{diff}}$ outlined in [5] (based on rewriting) and standard methods for solving $\mathcal{IDL}$ constraints (based on a reduction to finding negative cycles in a graph). To the best of our knowledge this is the first successful attempt of combining arrays and (a subset of) arithmetic for obtaining interpolants without quantifiers. The resulting interpolating procedure for $\mathcal{AX}_{\texttt{diff}} \cup \mathcal{IDL}$ may be applied for the verification of programs over arrays, such as sorting algorithms.

The paper is structured as follows. In Section 2, we recall some basic notions of rewriting and first order logic. In Section 3, we introduce the important notion of "modularized constraint". In Section 4, we outline a satisfiability solver for the combination of IDL and a subtheory $\mathcal{BAX}_{\texttt{diff}}$ of $\mathcal{AX}_{\texttt{diff}}$, and we extend it first to produce interpolants (in Section 5), and then to support full $\mathcal{AX}_{\texttt{diff}}$ (in Section 6). We conclude in Section 7. Omitted proofs can be found in the extended version, available at `http://homes.dsi.unimi.it/~ghilardi`.

Related Work. The research on algorithms for computing quantifier-free interpolants for a number of first-order theories has been an active area in the latest years. McMillan proposed in [12] a set of interpolating inference rules to compute interpolants for linear rational arithmetic ($\mathcal{LRA}$), uninterpreted functions ($\mathcal{EUF}$), and their combinations. Alternative approaches, targeted towards efficiency w.r.t. established decision procedures, are based on the lazy framework of [15], and can be found in [9] (for $\mathcal{EUF}$), and in [14,7] (for $\mathcal{LRA}$ and $\mathcal{LRA} \cup \mathcal{EUF}$). The latter also presents algorithms specific for difference logics ($\mathcal{IDL}$) and unit-two-variables-per-inequality ($\mathcal{UTVPI}$) constraints.

As far as the (classical) theory of arrays ($\mathcal{AX}$) is concerned, it is known [11] that quantifier-free interpolants cannot be computed in general. The same paper suggests a reduction approach to compute interpolants (with quantifiers) for arrays via reduction to uninterpreted functions and linear integer arithmetic ($\mathcal{LIA}$). Unlike [11], our approach is not based on a reduction to other theories. Following the same reduction approach, [3,4] present an interpolating calculus for computing (in general quantified) interpolants in linear integer arithmetic and some extensions, such as the combination of $\mathcal{LIA}$ with $\mathcal{EUF}$ or $\mathcal{LIA}$ with $\mathcal{AX}$. In contrast, our approach computes quantifier-free interpolants, as we rely on $\mathcal{AX}_{\texttt{diff}}$ instead of $\mathcal{AX}$ as the background theory for modelling arrays. Unlike [4], our approach uses a combination of rewriting and constraint solving as opposed to a sequent calculus, and interpolants are retrieved by means of the application of a set of metarules that record basic transformation steps on the set of constraints. Also, our combination method is *ad hoc*, and it is not based on

a Nelson-Oppen framework as in [15], as the theory of $\mathcal{IDL}$ is non-convex and it thus requires specific treatment. $\mathcal{AX}_{\mathtt{diff}}$ was shown in [5] to have the quantifier-free interpolation property. Here we show that the latter property still holds when combining $\mathcal{AX}_{\mathtt{diff}}$ with $\mathcal{IDL}$.

2 Background and Preliminaries

We assume the usual syntactic (e.g., signature, variable, term, atom, literal, formula, and sentence) and semantic (e.g., structure, truth, satisfiability, and validity) notions of first-order logic. The equality symbol "=" is included in all signatures considered below. For clarity, we shall use "≡" in the meta-theory to express the syntactic identity between two symbols or two strings of symbols.

Rewriting. We recall some notions and results about term rewriting (see, e.g., [2]) used in the paper. A total ordering $\succ$ on a signature Σ is called a 'precedence' relation. The Lexicographic Path Ordering (LPO) orients equalities by using a given precedence relation; usually, abusing notation, the same symbol is used for the precedence and the associated LPO. Given an equality $s = t$, we write $s \to t$ (called an oriented equality or rewriting rule) when $s \succ t$ for a given precedence $\succ$. Given a set E of oriented equalities, the reduction relation $t \to^* u$ holds when u is obtained by repeatedly rewriting subterms of t by using instances of the rules in E. We say that u is in normal form (w.r.t. a set E of rules) when no rule in E can be applied to u. A set E of rules is *ground irreducible* iff for every ground rule $l \to r$ from E, it is not possible to rewrite neither l nor r by using rules different from $l \to r$ itself. A set E of rules is *convergent* iff every term t has a unique *normal form*, denoted with $\hat{t}$, i.e., $t \to^* \hat{t}$ by using using the rules from E. If the rules in E are all ground and E is ground irreducible, then E is also convergent (because it has no critical pairs, see [2]).

Theories and Constraints. A *theory* T is a pair (Σ, Ax_T), where Σ is a signature and Ax_T is a set of Σ-sentences, called the axioms of T (we shall sometimes write directly T for Ax_T). The Σ-structures in which all sentences from Ax_T are true are the *models* of T. A Σ-formula ϕ is *T-satisfiable* if there exists a model $\mathcal{M}$ of T such that ϕ is true in $\mathcal{M}$ under a suitable assignment $\mathtt{a}$ to the free variables of ϕ (in symbols, $(\mathcal{M}, \mathtt{a}) \models \phi$); it is *$T$-valid* (in symbols, $T \vdash \varphi$) if its negation is T-unsatisfiable or, equivalently, iff φ is provable from the axioms of T in a complete calculus for first-order logic. A formula φ_1 *T-entails* a formula φ_2 if $\varphi_1 \to \varphi_2$ is *T-valid*; the notation used for such T-entailment is $\varphi_1 \vdash_T \varphi_2$ or simply $\varphi_1 \vdash \varphi_2$, if T is clear from the context. The *satisfiability modulo the theory* T ($SMT(T)$) *problem* amounts to establishing the T-satisfiability of quantifier-free Σ-formulae.

Let T be a theory in a signature Σ; a *T-constraint* (or, simply, a constraint) A is a set of ground literals in a signature Σ' obtained from Σ by adding a set of free constants. A finite constraint A can be equivalently seen as a single formula,

represented by the conjunction of its elements; thus, when we say that a constraint A is T-*satisfiable* (or just "satisfiable" if T is clear from the context), we mean that the associated formula (also called A) is satisfiable in a Σ'-structure which is a model of T.

Two finite constraints A and B are *logically equivalent* (modulo T) iff $T \vdash A \leftrightarrow B$. The notion of logical equivalence is often too strong when checking T-satisfiability of constraints, as we do in this paper. To overcome this problem, we introduce the notion of $\exists$-equivalence which is weaker than logical equivalence and still implies equisatisfiability of constraints. Let A be a first-order sentence, $A^{\exists}$ is the formula obtained from A by replacing free constants with variables and then existentially quantifying them out.

Definition 1. *Two finite constraints A and B (or, more generally, first order sentences) are* $\exists$-equivalent *(modulo T) iff $T \vdash A^{\exists} \leftrightarrow B^{\exists}$.*

Obviously, the preservation of $\exists$-equivalence is an important requirement for T-satisfiability procedures based on constraint transformations. As an example of such equisatisfiability-preserving transformations based on $\exists$-equivalence, we consider the renaming of terms by constants which will be used in our procedures below. This transformation takes a constraint A and replaces all the occurrences of one of its terms, say t, with a fresh constant a (i.e., a does not occur in A) so to obtain a new constraint A' such that $A' \cup \{a = t\}$ is $\exists$-equivalent to A, where the equality $a = t$ is called the *explicit definition of* t.

Theories of Arrays. Let $\mathcal{AX}$ denote the McCarthy theory of arrays with extensionality whose signature contains three sort symbols $\mathtt{ARRAY}, \mathtt{ELEM}, \mathtt{INDEX}$ and two function symbols rd of type $\mathtt{ARRAY} \times \mathtt{INDEX} \longrightarrow \mathtt{ELEM}$ and wr of type $\mathtt{ARRAY} \times \mathtt{INDEX} \times \mathtt{ELEM} \longrightarrow \mathtt{ARRAY}$. The set $\mathcal{AX}$ of axioms contains the following three sentences:

$$\forall y, i, e. \quad rd(wr(y, i, e), i) = e \tag{1}$$

$$\forall y, i, j, e. \quad i \neq j \Rightarrow rd(wr(y, i, e), j) = rd(y, j) \tag{2}$$

$$\forall x, y. \quad x \neq y \Rightarrow (\exists i.\ rd(x, i) \neq rd(y, i)). \tag{3}$$

It is known [11] that quantifier-free interpolants may not exist for two unsatisfiable quantifier-free formulae in $\mathcal{AX}$. To overcome this problem, in [5], we have introduced the following variant of $\mathcal{AX}$—called the theory of arrays with $\mathtt{diff}$ and denoted with $\mathcal{AX}_{\mathtt{diff}}$—whose signature is that of $\mathcal{AX}$ extended with the function symbol $\mathtt{diff}$ of type $\mathtt{ARRAY} \times \mathtt{ARRAY} \longrightarrow \mathtt{INDEX}$. The set $\mathcal{AX}_{\mathtt{diff}}$ of axioms contains (1), (2), and the following Skolemization of (3):

$$\forall x, y. \quad x \neq y \Rightarrow rd(x, \mathtt{diff}(x, y)) \neq rd(y, \mathtt{diff}(x, y)), \tag{4}$$

which constrains the interpretation of $\mathtt{diff}$ to be a (binary) function returning an index at which the input arrays store different values, if such an index exists; otherwise (i.e., when the arrays are identical) $\mathtt{diff}$ returns an arbitrary value. Quantifier-free interpolants can be computed for mutually unsatisfiable quantifier-free formulae of $\mathcal{AX}_{\mathtt{diff}}$ [5].

In this paper, we first consider a sub-theory $\mathcal{BAX}_{\mathtt{diff}}$ of $\mathcal{AX}_{\mathtt{diff}}$ whose signature is that of $\mathcal{AX}_{\mathtt{diff}}$ except for the function symbol wr, which is omitted. The set of axioms of $\mathcal{BAX}_{\mathtt{diff}}$ is the singleton containing just the Skolemization of extensionality, i.e., (4).

Integer Difference Logic. Following [8], we define $\mathcal{IDL}$ as the mono-sorted theory whose signature contains just one sort symbol, that we call `INDEX` (in preparation to "combine" this theory with $\mathcal{BAX}_{\mathtt{diff}}$ or $\mathcal{AX}_{\mathtt{diff}}$), the constant 0, the binary predicate $\leq$, and two unary function symbols *succ* and *pred*. The axioms of $\mathcal{IDL}$ are all the sentences which are true in the usual structure $\mathbb{Z}$ of the integers when interpreting the constant 0 as the number zero, $\leq$ as the natural ordering, *succ* as the successor $(\lambda x.(x+1))$, and *pred* as the predecessor $(\lambda x.(x-1))$ functions. Ground atoms of $\mathcal{IDL}$ are equivalent to formulæ of the form $i \bowtie S^n(j)$ (where $\bowtie \in \{=, \leq\}$, $S \in \{succ, pred\}$, and i, j are either free constants or 0) and are written as $i - j \bowtie n$ or as $i \bowtie j + n$, for $n \in \mathbb{Z}$. We use also obvious abbreviations like $i \bowtie j$ (for $i - 0 \bowtie j$), $i < j$ (for $i \leq j - 1$), or $i \geq j + n$ (for $j - i \leq -n$), etc. Using these abbreviations, it is easy to see that the theory just defined is usually referred to as "integer difference logic" in the literature.

3 Modularized Constraints for the Theory $\mathcal{BAX}_{\mathtt{diff}} \cup \mathcal{IDL}$

In this paper, we consider the composed theories $\mathcal{BAX}_{\mathtt{diff}} \cup \mathcal{IDL}$ and $\mathcal{AX}_{\mathtt{diff}} \cup \mathcal{IDL}$ (where $\cup$ denotes the union of the signatures and the axioms of the component theories) and design algorithms for the computation of (quantifier-free) interpolants. We do this in two steps. First, we describe an *ad hoc* combination of rewriting (for $\mathcal{BAX}_{\mathtt{diff}}$)—along the lines of [2]—and constraint solving (for $\mathcal{IDL}$) to build a satisfiability procedure and (on top of this) an interpolating algorithm for $\mathcal{BAX}_{\mathtt{diff}} \cup \mathcal{IDL}$. Then, we show how this can be lifted to compute quantifier-free interpolants also for $\mathcal{AX}_{\mathtt{diff}} \cup \mathcal{IDL}$.

Methodologically, this may appear surprising, but the following observations should clarify our choice. On the one hand, the component theories satisfy the hypotheses of the Nelson-Oppen combination method [13] for satisfiability checking. Furthermore, the Nelson-Oppen method has been extended to combine interpolating procedures in [15] for component theories which are "*equality interpolating*" in order to restrict the formulae to be propagated between interpolating procedures. On the other hand, for simplicity, the definition of equality interpolating theory in [15] applies only to convex theories; unfortunately, $\mathcal{IDL}$ is not convex and we were not able to extend the notion of equality interpolating to the non-convex case in a form that applies to our case. Our experience suggests that finding the right generalization of this notion is far from being trivial and explains why we preferred to design the *ad hoc* method in this paper. More precisely, after a pre-processing phase, we separate constraints in two parts: one pertaining to $\mathcal{BAX}_{\mathtt{diff}}$ and one to $\mathcal{IDL}$. Literals in $\mathcal{BAX}_{\mathtt{diff}}$ are transformed by ground rewriting (along the lines of [5]) while for those in $\mathcal{IDL}$, we adapt

available constraint solving techniques for integer difference logic. The goal of these transformations is to derive a (so-called) *modularized* constraint whose satisfiability is trivial to establish.

Let $a, b, \ldots$ denote free constants of sort `ARRAY`, $i, j, \ldots$ free constants of sort `INDEX`, $d, e, \ldots$ free constants of sort `ELEM`, and $\alpha, \beta, \ldots$ free constants of any sort. A *(ground) flat* literal is a literal of the form $i \bowtie j + n, rd(a, i) = e, \texttt{diff}(a, b) = i, \alpha = \beta, \alpha \neq \beta$. By replacing literals of the form $i \not\leq j$ with $j \leq i - 1$ and renaming terms with constants as explained in Section 2, given a constraint A it is always possible to produce an $\exists$-equivalent (*flat*) constraint A' such that A' contains only flat literals. We first analyze in detail flat constraints; we introduce some notions, aiming at defining, so called, "modularized" constraints for which satisfiability can be easily assessed both in isolation and combination.

Separation. We split a flat constraint A in two: the *index* part A_I and the *main* part A_M, where A_I contains the literals of the form $i = j + n, i \leq j + n, i \neq j, \texttt{diff}(a, b) = i$ and A_M contains the remaining literals, i.e., those of the forms $a = b, a \neq b, rd(a, i) = e, e = d, e \neq d$.

Rewriting for $\mathcal{BAX}_{\texttt{diff}}$. We fix the precedence $\succ$ to be such that $\leq \succ a \succ rd \succ \texttt{diff} \succ i \succ succ \succ pred \succ 0 \succ e$, for every a, i, e of the corresponding sorts. The LPO extension of $\succ$ allows us to orient all the equalities in the main part A_M of a constraint (in particular, we have that $rd(a, i) = e$ is oriented as $rd(a, i) \rightarrow e$, and $\alpha = \beta$ is oriented as $\alpha \rightarrow \beta$ when $\alpha \succ \beta$).

Constraint Solving for $\mathcal{IDL}$. Equalities in the index part A_I are classified as follows: a `diff`-*explicit definition* is an equality having the form $\texttt{diff}(a, b) = i$ and an $\mathcal{IDL}$-*explicit definition* is an equality of the form $i = j + n$ (with $i \not\equiv j, i \not\equiv 0$). Each equality in A_I can be rewritten as an $\mathcal{IDL}$-explicit definition, unless it is a `diff`-explicit definition, or a tautology (such as $i = i + 0$), or it is unsatisfiable (as $i = i + 4$). In an $\mathcal{IDL}$-explicit definition $i = j + n$, we say that "i is explicitly defined by $j + n$" (notice that j can be 0). As it is customary in solvers for difference logic (see, e.g. [1]), we associate the integer difference logic literals of the form $j \leq i + n$ with a weighted directed graph. More precisely, let $G(V, E)$ be a (finite, integer-weighted, directed) graph: the notation $i \xrightarrow{n} j$ means that there is an edge from i to j with weight n. We can associate the tuple $\langle G_A(V_A, E_A), \mathcal{D}_A, d_A, n_A \rangle$ to the index part A_I, where $\mathcal{D}_A$ is a set of `diff`-explicit definitions, d_A is a set of $\mathcal{IDL}$-explicit definitions, n_A is a set of negated equalities, and $G_A(V_A, E_A)$ has an edge $i \xrightarrow{n} j \in E_A$, for each $j - i \leq n \in A_I$.

Constraints in $\mathcal{BAX}_{\texttt{diff}} \cup \mathcal{IDL}$. We write a constraint A as follows:

$$A = \langle G_A, \mathcal{D}_A, d_A, n_A, A_M \rangle \ . \tag{5}$$

By abusing notation, we confuse the graph G_A with the corresponding set of inequalities and leave implicit both the set V_A of vertices and E_A of edges.

Definition 2 (Modularized Constraint). *A* modularized *constraint is a flat constraint of the form* (5) *such that the following conditions are satisfied:*

(i) d_A *is* appropriate *in the sense that: (a) each free constant has at most one definition; (b) different constants have different definitions; (c) no constant is defined as another constant (i.e., as* $j+0$*); (d) 0 is not defined; (e) defined constants do not occur as vertices in* G_A*;*
(ii) *the graph* G_A *is* acyclic*;*
(iii) *the rewrite system formed by the equalities in* A_M *is convergent and ground irreducible (below, the normal form of a term* t *w.r.t.* A_M *is denoted by* $\hat{t}$*); no* A_M *does not contain array inequalities* $a \neq b$*;*
(iv) $\{\mathtt{diff}(a,b) = i, \mathtt{diff}(a',b') = i'\} \subseteq \mathcal{D}_A$*,* $\hat{a} \equiv \widehat{a'}$*, and* $\hat{b} \equiv \widehat{b'}$ *imply* $i \equiv i'$*;*
(v) $\mathtt{diff}(a,b) = i \in \mathcal{D}_A$ *and* $\widehat{rd(a,i)} \equiv \widehat{rd(b,i)}$ *imply* $\hat{a} \equiv \hat{b}$*.*

Notice that no index equality $i = j$ may occur in a modularized constraint because of condition (i)(c) above. Also notice that condition (iii) is much stronger than what is usually required for satisfiability. The reason is that we want the satisfiability of modularized constraints to be invariant under addition of (implicit) inequalities.

Proposition 1. *Suppose that* A *is modularized and that there is no element inequality* $e \neq d$ *in* A_M *such that* $\hat{e} \equiv \hat{d}$*. Then* $A \cup \{\alpha \neq \beta\}_{\alpha,\beta}$ *(varying* α, β *among the different pairs of constants in normal form of the same sort occurring in* A*) is* $(\mathcal{BAX}_{\mathtt{diff}} \cup \mathcal{IDL})$*-satisfiable.*

Finally, conditions (iv) and (v) of Definition 2 deal with `diff`-explicit definitions: the former requires `diff` to be "well-defined" and the latter is a "conditional" reformulation of the contrapositive of axiom (4).

Combining Modularized Constraints. Let A, B be two constraints in the signatures Σ^A, Σ^B obtained from the signature Σ by adding some free constants and let $\Sigma^C := \Sigma^A \cap \Sigma^B$. Given a term, a literal, or a formula φ we call it:

- AB*-common* iff it is defined over Σ^C;
- A*-local* (resp. B*-local*) if it is defined over Σ^A (resp. Σ^B);
- A*-strict* (resp. B*-strict*) iff it is A-local (resp. B-local) but not AB-common;
- AB*-mixed* if it contains symbols in both $(\Sigma^A \setminus \Sigma^C)$ and $(\Sigma^B \setminus \Sigma^C)$;
- AB*-pure* if it does not contain symbols in both $(\Sigma^A \setminus \Sigma^C)$ and $(\Sigma^B \setminus \Sigma^C)$.

(Sometimes in the literature about interpolation, "A-local" and "B-local" are used to denote what we call here "A-strict" and "B-strict"). As we will see below, the following modularity result is crucial for interpolation.

Proposition 2. *Let* Σ *be the signature of* $\mathcal{BAX}_{\mathtt{diff}} \cup \mathcal{IDL}$*,* $A = \langle G_A, \mathcal{D}_A, d_A, n_A, A_M\rangle$*, and* $B = \langle G_B, \mathcal{D}_B, d_B, n_B, B_M\rangle$ *be modularized constraints in the expanded signatures* Σ^A, Σ^B*. We have that* $A \cup B$ *is modularized in case the four conditions below are all satisfied:*

(O) *the restriction of* A *and* B *to the common subsignature* $\Sigma^C := \Sigma^A \cap \Sigma^B$ *coincide;*
(I) *for each* $\mathcal{IDL}$*-explicit definition* $i = j + n \in d_A \cup d_B$*, if* $i \in \Sigma^C$ *then* $j \in \Sigma^C$*;*

(II) *given* $c, c' \in \Sigma_C$*, there is a path in* G_A *leading from* c *to* c' *iff there is a path in* G_B *leading from* c *to* c'*;*

(III) *for each equation (or rule) of the form* $\alpha = t$ *in* $A \cup B$*, if the term* t *is* AB*-common, then so is the constant* α*.*

4 A Satisfiability Solver for $\mathcal{BAX}_{\mathtt{diff}} \cup \mathcal{IDL}$

Given a (finite) constraint A in $\mathcal{BAX}_{\mathtt{diff}} \cup \mathcal{IDL}$, we present a sequence of transformations for deriving a set $\{A_i \mid 1 \leq i \leq n\}$ of modularized constraints such that $\bigvee_{1 \leq i \leq n} A_i$ is $\exists$-equivalent to A. Since the satisfiability of each A_i is easy to check, we can thus establish the satisfiability of the original constraint A. The transformations are closely related to those in [5] for the theory $\mathcal{AX}_{\mathtt{diff}}$. This approach has two advantages. First, it allows us to use the same method of [5] to lift the satisfiability solver to an interpolating solver. Second, as we will see in Section 6, it is easy to lift the interpolating solver for $\mathcal{BAX}_{\mathtt{diff}} \cup \mathcal{IDL}$ to the theory $\mathcal{AX}_{\mathtt{diff}} \cup \mathcal{IDL}$.

The satisfiability solver for $\mathcal{BAX}_{\mathtt{diff}} \cup \mathcal{IDL}$ consists of the three groups of transformations below applied to the input constraint A.

4.1 Preprocessing

This group consists of the following steps to be executed sequentially.

Step 1 Flatten A, by replacing sub-terms with fresh constants and adding the related defining equalities; replace also literals of the form $i \not\leq j$ with $j \leq i - 1$.

Step 2 Replace array inequalities $a \neq b$ by the following literals

$$\mathtt{diff}(a, b) = i, \quad rd(b, i) = e, \quad rd(a, i) = d, \quad d \neq e,$$

where i, e, d are fresh constants.

Step 3 Guess a total ordering on the index constants occurring in the constraint obtained from the application of the previous two steps. That is, for each pair (i, j) of indexes, add either $i = j$, $i \leq j - 1$, or $j \leq i - 1$. Remove the positive literals $i = j$ by replacing all occurrences of i with j if $i \succ j$ (according to the symbol precedence); otherwise, replace each occurrence of j with i. Now, if an unsatisfiable literal $i \neq i$ is derived, then try another guess. If all guesses produce an unsatisfiable literal, then return `unsat`isfiability; otherwise, each negative literal $i \neq j$ (for each $i \not\equiv j$) is now redundant and can be removed.

Step 4 For each pair (a, i) of constants such that $rd(a, i) = e$ does not occur in the current constraint, add the literal $rd(a, i) = e$ with e fresh constant.

It is easy to see that these four steps terminate and that we obtain a finite set $\{A_i \mid 1 \leq i \leq n\}$ of flat constraints, whose disjunction is $\exists$-equivalent to the

original constraint A. If there exists $i \in \{1, ..., n\}$ such that the exhaustive applications of the transformations in the next group (Completion) does not exit with a `failure`, then return `satisfiable`; otherwise (i.e., for each $i \in \{1, ..., n\}$, the transformations in the next group halt with a `failure`), report `unsatisfiable`.

4.2 Completion

Let $\langle G_A, \mathcal{D}_A, d_A, n_A, A_M \rangle$ be one of the n flat constraints obtained from Preprocessing. We exhaustively apply to this constraint the following group of rules, which are organized in three sub-groups to clarify their purposes. All the transformations below can be interleaved arbitrarily; they are all deterministic with the exception of (G2) which might introduce further case-splits.

(I) Graph Completion. The transformations in this sub-group aim at satisfying conditions (i) and (ii) of Definition 2. We view G_A in a constraint as both a graph and a set of inequalities (recall that an arc $i \xrightarrow{n} j$ represents the inequality $j - i \leq n$): when some inequalities in the constraint are modified, the graph is updated accordingly.

(G1) Suppose we have an $\mathcal{IDL}$-explicit definition $i = j + n$. If $i \equiv j$, then the literal is either trivially true and can be removed, or false and `failure` can be reported. Otherwise, i.e., if $i \succ j$ (when $j \succ i$, we can rewrite it to $j = i - n$), keep the equality $i = j + n$ and also the equalities of the kind $\texttt{diff}(a, b) = i$, but replace every other occurrence of i in the index part of the current constraint by $j + n$ (it is easy to see that the constraint remains flat, after normalization of ground atoms, if needed).

(G2) Suppose we have a cycle $i_1 \xrightarrow{n_1} i_2 \xrightarrow{n_2} i_3 \cdots i_k \xrightarrow{n_k} i_1$ in G_A. If $n_1 + \cdots + n_k < 0$, then report `failure`. If $k = 1$, then since $n_1 \geq 0$, the arc represents a tautology and can be removed. In case $k > 1$, we can assume that i_1 is the $\succ$-biggest node in the cycle (if this is not the case, a permutation of the cycle is sufficient to satisfy this assumption) and that i_1 does not have an $\mathcal{IDL}$-explicit definition (otherwise instruction (G2) is not applied, (G1) should be applied instead). Let $m := n_2 + \cdots + n_k$; the cycle entails that i_1 lies in the integer interval $[i_2 - n_1, i_2 + m]$ (i.e., it entails $i_1 = i_2 - n_1 \vee i_1 = i_2 - n_1 + 1 \vee \cdots \vee i_1 = i_2 + m$) and we can add to the current constraint an $\mathcal{IDL}$-explicit definition for i_1 via a disjunctive guessing.

(II) Knuth-Bendix Completion. The transformations in this sub-group aim at satisfying condition (iii) of Definition 2 by using a Knuth-Bendix completion process (see, e.g., [2]). In particular, (K1)-(K3) remove critical pairs.

(K1) $\boxed{d \ {}_*\!\leftarrow rd(b, i) \leftarrow rd(a, i) \rightarrow e' \rightarrow_* e}$

Remove the parent rule $rd(a, i) \rightarrow e'$ and keep the other parent rule $a \rightarrow b$. If $d > e$ (resp. $e > d$), then add the rule $d \rightarrow e$ (resp. $e \rightarrow d$); otherwise (i.e., when $d \equiv e$), do nothing. (Notice that terms of the form $rd(b, i)$ are always reducible to an element constant because of Step 4 in the pre-processing phase.)

(K2) $\boxed{e \ {}_*\!\leftarrow e' \leftarrow rd(a,i) \rightarrow d' \rightarrow_* d}$
If $e \not\equiv d$, then orient the critical pair, add it as a new rule, and remove one of the parent rules.

(K3) $\boxed{\alpha \ {}_*\!\leftarrow \alpha' \leftarrow \beta \rightarrow \beta'_1 \rightarrow_* \beta_1}$
If $\beta \not\equiv \beta_1$, then orient the critical pair, add it as a new rule, and remove one of the parent rules; here $\alpha, \alpha', \beta, \beta_1, \beta'_1$ are all either of sort `ARRAY` or of sort `ELEM`.

(K4) If the right-hand side of a current ground rewrite rule can be reduced, then reduce it as much as possible, remove the old rule, and replace it with the newly obtained reduced rule.

(K5) If there exists a negative literal $e \neq d \in A_M$ such that $e \rightarrow_* e' \ {}_*\!\leftarrow d$, then report `failure`.

(III) Handling `diff`**.** The transformations in this group take care of condition (iv)-(v) of Definition 2 (we write $t \downarrow t'$ to mean that $t \rightarrow_* u \ {}_*\!\leftarrow t'$ for some u).

(S) If $\mathtt{diff}(a,b) = i \in A_I$, $rd(a,i) \downarrow rd(b,i)$ and $a \succ b$, then add the rule $a \rightarrow b$ and replace $\mathtt{diff}(a,b) = i$ by $\mathtt{diff}(b,b) = i$ (this is needed for termination, it prevents the rule from being indefinitely applied).

(U) If $\{\mathtt{diff}(a,b) = i, \mathtt{diff}(a',b') = i'\} \subseteq A_I$, $a \downarrow a'$ and $b \downarrow b'$ for $i \not\equiv i'$, then report `failure` and backtrack to Step 3 of the pre-processing phase.

It can be proved that the algorithm using the groups of transformations described above terminates and computes modularized constraints. We are now in the position to derive the main result of this section:

Theorem 1. *Every constraint in* $\mathcal{BAX}_{\mathtt{diff}} \cup \mathcal{IDL}$ *is* $\exists$*-equivalent to a disjunction of modularized constraints. The algorithm described above decides* $(\mathcal{BAX}_{\mathtt{diff}} \cup \mathcal{IDL})$*-satisfiability.*

5 An Interpolating Solver for $\mathcal{BAX}_{\mathtt{diff}} \cup \mathcal{IDL}$

Our design of the interpolating solver (as in [5]) is based on an abstract framework, in which we focus on the *basic operations* necessary to derive an interpolating refutation, independently of the underlying satisfiability procedure.

5.1 Interpolating Metarules

Let A, B be constraints in signatures Σ^A, Σ^B expanded with free constants and $\Sigma^C := \Sigma^A \cap \Sigma^B$. Recall the definitions of AB-common, A-local, B-local, A-strict, B-strict, AB-mixed, AB-pure terms, literals and formulae given in Section 3. The goal is to compute a ground AB-common sentence ϕ such that $A \vdash_{\mathcal{BAX}_{\mathtt{diff}} \cup \mathcal{IDL}} \phi$ and $\phi \wedge B$ is $(\mathcal{BAX}_{\mathtt{diff}} \cup \mathcal{IDL})$-unsatisfiable, whenever $A \wedge B$ is $(\mathcal{BAX}_{\mathtt{diff}} \cup \mathcal{IDL})$-unsatisfiable.

The basic operations needed to re-design the solver of Section 4 in order to add the capability of computing interpolants are called *metarules* and are shown

in Table 1 (in this section, we use $\phi \vdash \psi$ for $\phi \vdash_{\mathcal{BAX}_{\mathrm{diff}} \cup \mathcal{IDL}} \psi$). The metarules are the same as those introduced in [5] to which the reader is pointed for more details. The correctness of the procedure explained in Subsection 5.2 relies on

Table 1. Interpolating Metarules: each rule has a proviso *Prov.* and an instruction *Instr.* for recursively computing the new interpolant ϕ' from the old one(s) $\phi, \phi_1, \ldots, \phi_k$. Metarules are applied *bottom-up* and interpolants are computed *top-down*.

Close1	Close2	Propagate1	Propagate2
$\dfrac{}{A \mid B}$	$\dfrac{}{A \mid B}$	$\dfrac{A \mid B \cup \{\psi\}}{A \mid B}$	$\dfrac{A \cup \{\psi\} \mid B}{A \mid B}$
Prov.: A is unsat. *Instr.*: $\phi' \equiv \bot$.	*Prov.*: B is unsat. *Instr.*: $\phi' \equiv \top$.	*Prov.*: $A \vdash \psi$ and ψ is AB-common *Instr.*: $\phi' \equiv \phi \wedge \psi$.	*Prov.*: $B \vdash \psi$ and ψ is AB-common *Instr.*: $\phi' \equiv \psi \rightarrow \phi$.

Define0	Define1	Define2
$\dfrac{A \cup \{a = t\} \mid B \cup \{a = t\}}{A \mid B}$	$\dfrac{A \cup \{a = t\} \mid B}{A \mid B}$	$\dfrac{A \mid B \cup \{a = t\}}{A \mid B}$
Prov.: t is AB-common, a fresh. *Instr.*: $\phi' \equiv \phi(t/a)$.	*Prov.*: t is A-local and a is fresh. *Instr.*: $\phi' \equiv \phi$.	*Prov.*: t is B-local and a is fresh. *Instr.*: $\phi' \equiv \phi$.

Disjunction1	Disjunction2
$\dfrac{\cdots \quad A \cup \{\psi_k\} \mid B \quad \cdots}{A \mid B}$	$\dfrac{\cdots \quad A \mid B \cup \{\psi_k\} \quad \cdots}{A \mid B}$
Prov.: $\bigvee_{k=1}^{n} \psi_k$ is A-local and $A \vdash \bigvee_{k=1}^{n} \psi_k$. *Instr.*: $\phi' \equiv \bigvee_{k=1}^{n} \phi_k$.	*Prov.*: $\bigvee_{k=1}^{n} \psi_k$ is B-local and $B \vdash \bigvee_{k=1}^{n} \psi_k$. *Instr.*: $\phi' \equiv \bigwedge_{k=1}^{n} \phi_k$.

Redplus1	Redplus2	Redminus1	Redminus2
$\dfrac{A \cup \{\psi\} \mid B}{A \mid B}$	$\dfrac{A \mid B \cup \{\psi\}}{A \mid B}$	$\dfrac{A \mid B}{A \cup \{\psi\} \mid B}$	$\dfrac{A \mid B}{A \mid B \cup \{\psi\}}$
Prov.: $A \vdash \psi$ and ψ is A-local. *Instr.*: $\phi' \equiv \phi$.	*Prov.*: $B \vdash \psi$ and ψ is B-local. *Instr.*: $\phi' \equiv \phi$.	*Prov.*: $A \vdash \psi$ and ψ is A-local. *Instr.*: $\phi' \equiv \phi$.	*Prov.*: $B \vdash \psi$ and ψ is B-local. *Instr.*: $\phi' \equiv \phi$.

ConstElim1	ConstElim2	ConstElim0
$\dfrac{A \mid B}{A \cup \{a = t\} \mid B}$	$\dfrac{A \mid B}{A \mid B \cup \{b = t\}}$	$\dfrac{A \mid B}{A \cup \{c = t\} \mid B \cup \{c = t\}}$
Prov.: a is A-strict and does not occur in A, t. *Instr.*: $\phi' \equiv \phi$.	*Prov.*: b is B-strict and does not occur in B, t. *Instr.*: $\phi' \equiv \phi$.	*Prov.*: c, t are AB-common, c does not occur in A, B, t. *Instr.*: $\phi' \equiv \phi$.

Proposition 3 below. Before stating the proposition, we need to introduce the following formal notion. An *interpolating metarules refutation* for A, B is a labeled tree having the following properties: (i) nodes are labeled by pairs of finite sets of constraints; (ii) the root is labeled by A, B; (iii) the leaves are labeled by a pair $\tilde{A}, \tilde{B}$ such that $\bot \in \tilde{A} \cup \tilde{B}$; (iv) each non-leaf node is the conclusion of a rule from Table 1 and its successors are the premises of that rule.

Proposition 3 ([5]). *If there exists an interpolating metarules refutation for A, B then there is a quantifier-free interpolant for A, B (i.e., there exists a quantifier-free AB-common sentence ϕ such that $A \vdash \phi$ and $B \wedge \phi \vdash \bot$). The interpolant ϕ is recursively computed by applying the relevant interpolating instructions from Table 1.*

Metarules are useful to design an algorithm manipulating pairs of constraints based on transformation instructions. Each of the transformation instructions is derived from the satisfiability solver of Section 4 and is *justified* by a metarule (or by a sequence of metarules): in this way, if our instructions form a complete and terminating algorithm, we can use Proposition 3 to get the desired interpolants. The main advantage of this approach is that we just need to take care of the completeness and termination of the algorithm, while ignoring interpolants. Here "completeness" means that our transformations should be able to bring a pair (A, B) of constraints into an $\exists$-equivalent set of pairs of constraints (A_i, B_i) that either match the requirements of Proposition 2 or are trivially unsatisfiable (i.e., $\bot \in A_i \cup B_i$). By Theorem 1, the latter happens iff the original pair (A, B) is $(\mathcal{BAX}_{\mathtt{diff}} \cup \mathcal{IDL})$-unsatisfiable or, equivalently, we get an interpolating metarules refutation.

5.2 The Interpolating Solver

The key idea for lifting the satisfiability solver of Section 4 to an interpolating solver is that of invoking it separately on A and B, and propagating equalities involving AB-common terms. We shall use *a precedence in which AB-common constants are smaller than A-strict or B-strict constants of the same sort.* Unfortunately, this is not sufficient to prevent the instances of the satisfiability solver from generating literals and rules violating one or more of the hypotheses of Proposition 2. This is the reason for introducing further correcting instructions (γ)-(δ) below. The interpolating solver consists of two groups of instructions, detailed below and called pre-processing and completion, derived from those in Sections 4.1 and 4.2. In the following, the A-component and the B-component of the constraints under consideration will be called A and B.

Pre-processing. This group of transformations contains those in Section 4.1. They are performed on both A and B. To justify these transformations, we need metarules (Define0,1,2), (Redplus1,2), (Redminus1,2), (Disjunction1,2), (ConstElim0,1,2), and (Propagate1,2) in Table 1. The last two are required because when i and j are AB-common, the case-splitting on $i = j$, $i < j$, or $j < i$ of Step 3 can be done—say—in A and then propagated to B. After the applications of these transformations, the following invariants (to be maintained also by the next group of transformations) hold:

(i1) A (resp. B) entails (modulo $\mathcal{BAX}_{\texttt{diff}} \cup \mathcal{IDL}$) either $i < j$ or $j < i$, for each A-local (resp. B-local) constants i, j of sort `INDEX` in A (resp. B);
(i2) if the constants a, i occur in A (resp. in B), then $rd(a, i)$ reduces to an A-local (resp. B-local) constant of sort `ELEM`.

Completion. This group of transformations are executed non-deterministically until no more rules can be applied.

(α) Apply any instruction of Section 4.2 to A or B.
(β) If there is an AB-common literal that belongs to A but not to B (or vice versa), copy it to B (resp. A).
(γ) "Repair" (see below for a precise description) those literals violating conditions (I) or (III) of Proposition 2, called *undesired literals* below.
(δ) If G_A (or G_B) contains a path $i_1 \xrightarrow{n_1} i_2 \xrightarrow{n_2} i_3 \cdots i_k \xrightarrow{n_k} i_{k+1}$ between AB-common constants i_1 and i_{k+1} and there is no path from i_1 to i_{k+1} in $G_A \cap G_B$, then add the inequality $i_{k+1} - i_1 \leq n_1 + \cdots + n_k$ to both A and B (so that an arc from i_1 to i_{k+1} will be created in $G_A \cap G_B$).

Instructions in (α) deleting an AB-common literal should be performed *simultaneously* in A and B. It can be easily checked (the check is done within the proof of Theorem 2) that this is always possible by inspecting the transformations in Section 4.2. An easy way to guarantee this is to give higher priority to the rules in (β) and (γ).

Preliminary to describing how to "repair" literals—i.e., the instructions in (γ)—we need to introduce a technique that we call *Term Sharing*. Suppose that A contains a literal $\alpha = t$ where the term t is AB-common but the free constant α is only A-local. It is possible to "make α AB-common" as follows. First, introduce a fresh AB-common constant α' with the explicit definition $\alpha' = t$ (to be inserted both in A and in B, as justified by metarule (Define0)). Then, replace the literal $\alpha = t$ with $\alpha = \alpha'$ and α with α' everywhere else in A. Finally, delete $\alpha = \alpha'$. The result is a pair (A, B) of constraints which is almost identical to the original pair except for the fact that α has been renamed to an AB-common constant α'. These transformations can be justified by metarules (Define0), (Redplus1), (Redminus1), (ConstElim1). This concludes the description of Term Sharing.

We are now in the position to explain the instructions in (γ): notice that literals violating conditions (I) or (III) of Proposition 2 are all of the form $\alpha = t$, where t is AB-common and α is, say, just A-local (this applies also to the literals $i = j + n$ violating (I), because they can be rewritten as $j = i - n$). Clearly, Term Sharing can replace them by literals of the form $\alpha' = t$, where α' is AB-common too. There is however a subtlety to take care of in case α is of sort `INDEX`: since α' is AB-common whereas α is only A-local, we might need to perform some guessing to maintain invariant (i1). In other words, we need to repeat Step 3 from Section 4.1 until invariant (i1) is restored (α' must be compared with the other B-local constants of sort `INDEX`).

By exhaustively applying the transformations in the two groups above (namely, Pre-processing and Completion) on a pair (A, B) of constraints, we can produce a tree whose nodes are labelled by pairs of constraints and such that

the successor nodes are labelled by pairs of constraints that are obtained by applying an instruction. We call such a tree an *interpolating tree* for (A, B). The key observation is that interpolating trees are interpolating metarules refutation trees when input pairs of constraints are mutually unsatisfiable.

Theorem 2. *Any interpolating tree for (A, B) is finite and it is an interpolating metarules refutation iff $A \wedge B$ is $(\mathcal{BAX}_{\mathtt{diff}} \cup \mathcal{IDL})$-unsatisfiable.*

By using this theorem and recalling Proposition 3, a (quantifier-free) interpolant can be recursively computed by using the metarules of Table 1. In other words, we have designed a quantifier-free interpolating solver for $\mathcal{BAX}_{\mathtt{diff}} \cup \mathcal{IDL}$.

Theorem 3. *$\mathcal{BAX}_{\mathtt{diff}} \cup \mathcal{IDL}$ admits quantifier-free interpolants (i.e., for every quantifier free formulae ϕ, ψ such that $\psi \wedge \phi$ is $(\mathcal{BAX}_{\mathtt{diff}} \cup \mathcal{IDL})$-unsatisfiable, there exists a quantifier free formula θ such that: (i) $\psi \vdash_{\mathcal{BAX}_{\mathtt{diff}} \cup \mathcal{IDL}} \theta$; (ii) $\theta \wedge \phi$ is not $(\mathcal{BAX}_{\mathtt{diff}} \cup \mathcal{IDL})$-satisfiable; and (iii) only the variables occurring both in ψ and ϕ occur also in θ).*

6 An Interpolating Solver for $\mathcal{AX}_{\mathtt{diff}} \cup \mathcal{IDL}$

We now sketch how to lift the interpolating solver for $\mathcal{BAX}_{\mathtt{diff}} \cup \mathcal{IDL}$ to one for $\mathcal{AX}_{\mathtt{diff}} \cup \mathcal{IDL}$ by combining the results from Section 5 with those of [5]. Since $\mathcal{BAX}_{\mathtt{diff}}$ is a sub-theory of $\mathcal{AX}_{\mathtt{diff}}$, it is straightforward to reuse the rewriting approach of [5] to extend the solver outlined in the previous section. Because of lack of space, we only describe the key ideas and we point the reader to [5] for details about the solver for $\mathcal{AX}_{\mathtt{diff}}$. The difference between $\mathcal{BAX}_{\mathtt{diff}}$ and $\mathcal{AX}_{\mathtt{diff}}$ is in the presence of the function symbol wr and the axioms (1) and (2). We explain how the rewriting techniques of [5], used to cope with terms consisting of nested wr's, can be seen as an extension of those used in this paper. We recall the following notation from [5]: $wr(a, I, E)$ abbreviates the term $wr(wr(\cdots wr(a, i_1, e_1)\cdots), i_n, e_n)$, i.e., a nested write on the array variable a where indexes and elements are represented by the free constants lists $I \equiv i_1, \ldots, i_n$ and $E \equiv e_1, \ldots, e_n$, respectively.

We extend our precedence in such a way that $a \succ wr \succ rd \succ \mathtt{diff} \succ i$ holds, for all constants a, i. This condition (satisfied by the precedence adopted in [5]) implies that an equality $a = wr(b, I, E)$ can be turned to a rewrite rule of the form $a \rightarrow wr(b, I, E)$ when $a \succ b$. As explained in [5], this is crucial to design an extended version of the Knuth-Bendix completion (see Section 4.2 of this paper), which allows for computing modularized constraints in $\mathcal{AX}_{\mathtt{diff}}$. Intuitively, the Knuth-Bendix completion is added transformations for eliminating "badly orientable" equalities (i.e., equalities of the form $b = wr(a, I, E)$ with $a \succ b$) that may arise. Such transformations solve the equality $b = wr(a, I, E)$ for a, thereby deriving a rewriting rule $a \rightarrow wr(b, I, E')$ for suitable E'.

Recall from [5] that an $\mathcal{AX}_{\mathtt{diff}}$ flat constraint contains only literals of the forms $\alpha = \beta, \alpha \neq \beta, rd(a, i) = e, \mathtt{diff}(a, b) = i$, and $b = wr(a, I, E)$. An $\mathcal{AX}_{\mathtt{diff}} \cup \mathcal{IDL}$ constraint is *flat* iff its restrictions to the signatures of $\mathcal{BAX}_{\mathtt{diff}} \cup \mathcal{IDL}$ and $\mathcal{AX}_{\mathtt{diff}}$ are flat.

Definition 3. *An $\mathcal{AX}_{\mathtt{diff}} \cup \mathcal{IDL}$ flat constraint is* modularized *iff (a) its restriction to the signature of $\mathcal{BAX}_{\mathtt{diff}} \cup \mathcal{IDL}$ is modularized (according to Definition 2) and (b) its restriction to the signature of $\mathcal{AX}_{\mathtt{diff}}$ is modularized according to Definition 3.1 of [5].*

The most important additional requirements of Definition 3.1 in [5] induce irredundant normal forms for terms built out of *rd*'s and *wr*'s by means of a set of non-ground rewrite rules corresponding to the axioms (1) and (2).

By "merging" the proof of Proposition 1 and that of the corresponding result in [5], we can show that the satisfiability of modularized constraint is invariant under addition of (implicit) inequalities, i.e., that *Proposition 1 holds also for $\mathcal{AX}_{\mathtt{diff}} \cup \mathcal{IDL}$.* By "merging" the proofs of Proposition 2 and Proposition 3.3 in [5], we can "combine" modularized $\mathcal{AX}_{\mathtt{diff}} \cup \mathcal{IDL}$ constraints.

Proposition 4. *Let A and B be $\mathcal{AX}_{\mathtt{diff}} \cup \mathcal{IDL}$ modularized constraints in expanded signatures Σ^A, Σ^B (here Σ is the signature of $\mathcal{AX}_{\mathtt{diff}} \cup \mathcal{IDL}$). We have that $A \cup B$ is modularized in case the conditions (O)-(III) of Proposition 2 and the conditions (O)-(III) of Proposition 3.3 from [5] are both satisfied.*

At this point, we have all the ingredients to design first a satisfiability solver for $\mathcal{AX}_{\mathtt{diff}} \cup \mathcal{IDL}$ and then to extend it to an interpolating solver. The first step consists of the merging of the transformations in the groups pre-processing and completion of Section 4 with those in Section 4 of [5]. It is then possible to prove that the resulting algorithm checks for $(\mathcal{AX}_{\mathtt{diff}} \cup \mathcal{IDL})$-satisfiability by "merging" the proofs of Theorem 1 and Theorem 4.1 of [5].

The second step amounts to integrate the instructions of the interpolating solver from Section 5.2 with those of the interpolating solver from Section 5.2 in [5] (notice that the interpolating metarules used in this paper are identical to those in [5]). The most important addition is the repairing of undesired literals of the form $c \rightarrow wr(c', I, E)$ whose left-hand side is AB-common but whose right-hand side is, say, only A-local: repairing requires a careful splitting of I and E into sub-lists, additional guessings, and manipulations of nested *wr*'s similar to those for eliminating badly orientable equations (see Section 5.2 in [5] for details). It is then possible to build interpolating trees (defined in a similar way as those in Section 5.2).

Theorem 4. *The theory $\mathcal{AX}_{\mathtt{diff}} \cup \mathcal{IDL}$ admits quantifier-free interpolants (in the sense of Theorem 3 where $\mathcal{BAX}_{\mathtt{diff}} \cup \mathcal{IDL}$ is replaced with $\mathcal{AX}_{\mathtt{diff}} \cup \mathcal{IDL}$).*

The proof of the above theorem is obtained by a straightforward merging of the proofs of the corresponding results for $\mathcal{BAX}_{\mathtt{diff}} \cup \mathcal{IDL}$ and $\mathcal{AX}_{\mathtt{diff}}$ (the complexity measures from the termination arguments are essentially the same).

7 Conclusions and Future Work

In this work we have shown how to derive an interpolating (satisfiability) solver for the theory of $\mathcal{AX}_{\mathtt{diff}} \cup \mathcal{IDL}$. Most importantly, the produced interpolants

are quantifier-free: we are not aware of any other approach that can derive interpolants for arrays and arithmetic without introducing quantifiers. Thus our work can find suitable applications in existing verification techniques based on abstraction-refinement loops with the help of interpolants.

In order to achieve our result, we have combined rewriting techniques for two variants of the theory of arrays, $\mathcal{BAX}_{\mathtt{diff}}$ and $\mathcal{AX}_{\mathtt{diff}}$, with a constraint solver for $\mathcal{IDL}$ based on a reduction to graph algorithms. Interpolants may be computed with the help of interpolating metarules to be applied in reverse order w.r.t. the algorithmic transformations steps.

We plan to implement our approach in the SMT solver OpenSMT [6] in order to carry out an extensive experimental evaluation.

References

1. Cherkassky, B., Goldberg, A.: Negative-cycle Detection Algorithms. In: Díaz, J. (ed.) ESA 1996. LNCS, vol. 1136, pp. 349–363. Springer, Heidelberg (1996)
2. Baader, F., Nipkow, T.: Term rewriting and all that. Cambridge University Press, Cambridge (1998)
3. 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)
4. Brillout, A., Kroening, D., Rümmer, P., Wahl, T.: Beyond Quantifier-Free Interpolation in Extensions of Presburger Arithmetic. In: VMCAI (to appear, 2012)
5. Bruttomesso, R., Ghilardi, S., Ranise, S.: Rewriting-based Quantifier-free Interpolation for a Theory of Arrays. In: RTA (2011)
6. Bruttomesso, R., Pek, E., Sharygina, N., Tsitovich, A.: The openSMT solver. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 150–153. Springer, Heidelberg (2010)
7. Cimatti, A., Griggio, A., Sebastiani, R.: Efficient Interpolation Generation in Satisfiability Modulo Theories. ACM Trans. Comput. Logic 12, 1–54 (2010)
8. Enderton, H.B.: A Mathematical Introduction to Logic. Academic Press, New York (1972)
9. Fuchs, A., Goel, A., Grundy, J., Krstić, S., Tinelli, C.: Ground Interpolation for the Theory of Equality. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 413–427. Springer, Heidelberg (2009)
10. Henzinger, T., McMillan, K.L., Jhala, R., Majumdar, R.: Abstractions from Proofs. In: POPL (2004)
11. Kapur, D., Majumdar, R., Zarba, C.: Interpolation for Data Structures. In: SIGSOFT 2006/FSE-14, pp. 105–116 (2006)
12. McMillan, K.L.: An Interpolating Theorem Prover. Theor. Comput. Sci. 345(1), 101–121 (2005)
13. Nelson, G., Oppen, D.C.: Simplification by Cooperating Decision Procedures. ACM Transactions on Programming Languages and Systems 1(2), 245–257 (1979)
14. Rybalchenko, A., Sofronie-Stokkermans, V.: Constraint Solving for Interpolation. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 346–362. Springer, Heidelberg (2007)
15. Yorsh, G., Musuvathi, M.: A Combination Method for Generating Interpolants. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 353–368. Springer, Heidelberg (2005)

Superposition Modulo Non-linear Arithmetic

Andreas Eggers[1], Evgeny Kruglov[2], Stefan Kupferschmid[3], Karsten Scheibler[3], Tino Teige[1], and Christoph Weidenbach[2]

[1] Dept. of Computing Science, Carl von Ossietzky Universität Oldenburg
D-26111 Oldenburg, Germany
{andreas.eggers,tino.teige}@informatik.uni-oldenburg.de
[2] Universität des Saarlandes, Max-Planck-Institut für Informatik
Campus E1 4, D-66123 Saarbrücken, Germany
{ekruglov,weidenbach}@mpi-inf.mpg.de
[3] Institute of Computer Science, Albert-Ludwigs-University
D-79110 Freiburg im Breisgau, Germany
{skupfers,scheibler}@informatik.uni-freiburg.de

Abstract. The first-order theory over non-linear arithmetic including transcendental functions (NLA) is undecidable. Nevertheless, in this paper we show that a particular combination with superposition leads to a sound and complete calculus that is useful in practice. We follow basically the ideas of the SUP(LA) combination, but have to take care of undecidability, resulting in "unknown" answers by the NLA reasoning procedure. A pipeline of NLA constraint simplification techniques related to the SUP(NLA) framework significantly decreases the number of "unknown" answers. The resulting approach is implemented as SUP(NLA) by a system combination of SPASS and iSAT. Applied to various scenarios of traffic collision avoidance protocols, we show by experiments that SPASS(iSAT) can fully automatically proof and disproof safety properties of such protocols using the very same formalization.

1 Introduction

In this paper we investigate the hierarchic combination of reasoning in non-linear arithmetic over the reals including transcendental functions (NLA) with superposition-based first-order logic (FOL) reasoning (SUP). The result is a very expressive language, where already validity in its parts (non-linear arithmetic, first-order logic) is undecidable, in general. Completeness for the resulting calculus SUP(NLA) or compactness of the resulting logic FOL(NLA) does not hold, in general.

Nevertheless, we show that for a particular class of FOL(NLA) formulae, the logic is compact and the SUP(NLA) calculus is sound and complete. This class, omitting function symbols on the first-order side, is very well-suited to formalize safety properties of non-linear systems, such as collision avoidance protocols. In order to provide automatic reasoning on such properties, including automatic proofs and disproofs, we develop a pipeline of simplification mechanisms for NLA constraints that eventually enables a completely automatic behavior of

C. Tinelli and V. Sofronie-Stokkermans (Eds.): FroCoS 2011, LNAI 6989, pp. 119–134, 2011.

SPASS(iSAT) on various scenarios of collision avoidance protocols. The clauses of the language have the form

$$\Lambda \parallel \Gamma \rightarrow \Delta$$

where Λ is a sequence of NLA literals, and Γ and Δ are sequences of first-order atoms without NLA signature symbols. Both parts share variables that are assumed to be universally quantified. The semantics of a clause is given by the implication $(\bigwedge \Lambda \wedge \bigwedge \Gamma) \rightarrow (\bigvee \Delta)$. A typical example is a clause like

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \leq 5, t' = 0 \parallel L(x_1, y_1, x_2, y_2, t) \rightarrow M(x_1, y_1, x_2, y_2, t')$$

where if (x_i, y_i) represent coordinates in the plane for two respective objects, the clause says that if the predicate L holds for the objects and they get closer than 5 then the predicate M shall hold for the objects. So the clause expresses a switch from a free movement to a maneuver movement if the two objects get too close, provided L and M are axiomatized accordingly.

The SUP(NLA) calculus reasons primarily on the first-order part via superposition (ordered resolution) and an additional constraint refutation rule. The NLA reasoning is invoked for redundancy elimination and a final check whether a contradiction has been found. Redundancy checks result in two different reasoning problems. A (newly generated) clause $\Lambda \parallel \Gamma \rightarrow \Delta$ is a tautology if Λ is not satisfiable, i.e., the closed NLA formula $\exists \boldsymbol{x} [\bigwedge \Lambda]$ is unsatisfiable. Checking redundancy among several clauses means testing implication between those clauses. In the context of this paper we mainly consider subsumption. If the FOL parts of two clauses $\Lambda_1 \parallel \Gamma_1 \rightarrow \Delta_1$, $\Lambda_2 \parallel \Gamma_2 \rightarrow \Delta_2$ subsume, i.e., $\Gamma_1\sigma \subseteq \Gamma_2$, $\Delta_1\sigma \subseteq \Delta_2$ for some matcher σ, the condition of the constraints to be checked is validity of the closed formula $\forall \boldsymbol{x} \exists \boldsymbol{y} [\Lambda_2 \rightarrow \Lambda_1\sigma]$, where $y_i \in (\mathrm{vars}(\Lambda_1\sigma) \setminus \mathrm{vars}(\Lambda_2))$ and $\boldsymbol{x}$ are all other variables. The third reasoning task is to check for the constraint refutation rule whether a single constraint is satisfiable. So it collapses with the tautology check.

In order to establish a first-order part empty clause $\Lambda \parallel \rightarrow$ to be a contradiction, NLA reasoning on Λ must *not* return the inconclusive answer UNKNOWN. This is indispensable for proving a conjecture. For disproving, i.e., having SUP(NLA) terminate on a set of clauses without finding an empty clause, the same holds for the above mentioned subsumption implication checks. For our case study the first-order part of the clauses is inherently recursive, e.g., the continuous linear movement clause is of the form $\Lambda \parallel L(x_1, y_1, x_2, y_2, t) \rightarrow L(x_1', y_1', x_2', y_2', t')$ where the positive and negative occurrence of the L literal have a first-order unifier. It turns out that SUP(NLA) termination can only be achieved by successful subsumption applications that require a pipelining of NLA simplification techniques related to the form of constraints generated by the superposition calculus.

For the experiments, we have actually implemented the calculus by a combination of SPASS [23] for the first-order reasoning with iSAT [13] for the NLA reasoning. Analogous to the behavior of SMT solvers, iSAT tries to solve the problem whether a formula of the form $\exists \boldsymbol{x}\, \phi$, where ϕ is a quantifier free boolean combination of non-linear atoms, is satisfiable. Although non-linear validity is undecidable, iSAT is a terminating procedure that provides the answers YES,

No, and Unknown. It is based on interval analysis, where an initial interval is assumed for each variable. Intervals are split in order to perform case analysis. If for each branch of the resulting tree the induced subproblem can be decided as unsatisfiable then the formula itself is unsatisfiable (iSAT answers No). If there is a branch which induced subproblem is proved to be satisfiable then the overall problem is as well (iSAT answers Yes). However, interval analysis is an incomplete calculus potentially causing that some of the subproblems cannot be decided. To avoid infinite interval splitting and to achieve termination, iSAT stops the proof search whenever the interval widths for all variables are small enough, e.g. less than 10^{-5} (iSAT answers Unknown).

Mapping iSAT to the above reasoning tasks in the context of SUP(NLA), there are two problems to be solved: the formula with quantifier alternation needed for the redundancy check which is not an SMT formula and the case iSAT answers Unknown. The former is solved by first reducing the number of existentially quantified variables through equational propagation. Then we test whether a linear relaxation of the result where the non-linear functions are considered as uninterpreted functions can already be proven (here we use Z3 [10]). If this does fail and there are no existentially quantified variables left, we pass the result to iSAT. All simplification is implemented on the Spass side. The latter problem is solved by an extension of iSAT called *strong satisfaction check* (Section 5). A disadvantage of the interval based reasoning approach of iSAT is the loss of precision when intervals are propagated through equations. Equations frequently induce point solutions. Here, without the strong satisfaction check, iSAT would terminate with an Unknown answer, providing narrow intervals. The strong satisfaction extension then takes those narrow intervals and tries to compute a certificate for a (point) solution. It turns out that this extension turned most of iSAT's Unknown answers in the SUP(NLA) context into definite Yes answers.

The application scenario (Section 6) is a collision avoidance protocol for moving objects (e.g., robots, aircrafts). The idea of the protocol is to prevent a collision, more precisely, a situation where the objects get too close. In order to achieve this goal, the movement of the objects is put into a maneuver mode, once their distance falls below a given limit. The maneuver mode then takes care by performing appropriate movement in form of sine curves to get the objects across each other and release them afterwards to their initial behavior. We studied the following three scenarios: (i) two objects in 2D space starting with linear movement, (ii) two objects in 3D space starting with linear movement, and (iii) two objects in 3D space starting with arbitrary movement. For all scenarios we can fully automatically prove and disprove (given different parameters) the collision freeness of the protocol. The protocol and the collision freeness property are modelled by a set of FOL(NLA) Horn clauses such that this set is satisfiable iff the protocol is collision free. If not, an unsatisfiability SUP(NLA) proof yields a counterexample. Our contributions are: (i) the first sound and complete combination of FOL(NLA) including an implementation, (ii) dedicated NLA simplification techniques providing (iii) nice experimental results by fully automatic verification of the above scenarios of a non-trivial

collision avoidance protocol. The paper ends with a summary including discussion of related work (Section 7). Missing proofs and the all formalizations can be found in an AVACS technical report [12].

2 Preliminaries

For non-linear arithmetic over the reals $\mathbb{R}$ we use the signature $\{+, -, \cdot, \text{abs}, \min, \max, \sin, \cos, \exp, \text{nrt}, \text{pow}, \leq, <, =, \neq, >, \geq\} \subset \Omega$, where Ω includes furthermore the reals. In our examples, we stick to rational coefficients for this paper. The semantics of the operators is the standard model $\mathcal{M}_{\mathbb{R}}$ of non-linear arithmetic over the reals. Terms, substitutions and first-order formulae over Ω and a set of variables $\mathcal{X}$ of sort $\mathbb{R}$ are defined as usual. A quantifier free formula ϕ over Ω is called an SMT-formula. Note that we only consider terms and formulae with total operators. An SMT-formula ϕ is *satisfiable*, if there exists a ground substitution τ such that $\mathcal{M}_{\mathbb{R}} \models \phi\tau$, also written $\tau \models \phi$. Note that this language is very expressive as total division $z = x/y$ can be coded as multiplication $x = y \cdot z \wedge y \neq 0$, the constant π can be coded as the solution to the variable x in the formula $\phi \equiv x > 3.1 \wedge x < 3.2 \wedge \sin(x) = 0$ and the integers as solutions to the variable y in $\phi \wedge \sin(x \cdot y) = 0$.

For the hierarchic FOL(NLA) setting [2, 1], where the first-order language includes equality $\approx$, the NLA operators Ω are extended by free operators to Ω', $\Omega \subseteq \Omega'$. In the resulting hierarchic specification $\mathbb{R}$ is the *base sort*, $\mathcal{M}_{\mathbb{R}}$ the *base theory* and the ground terms built over Ω the *base terms*.

Although we do not explicitly introduce a sort concept, in addition to the base sort we assume a *free sort* containing all other terms, in particular built over $\Omega' \setminus \Omega$. We say that a term is *pure*, if it does not contain both a base operator and a non-base operator. A substitution is called *simple*, if it maps every variable of the base sort to a base term. In general, there are non-base terms in the base sort provided a function symbol in $\Omega' \setminus \Omega$ ranging into the base sort. If σ is a simple substitution, $t\sigma$ is called a *simple instance* of t (analogously for equations and clauses). The set of simple ground instances of a clause C is denoted by $sgi(C)$, analogously $sgi(N)$ is the set of all simple ground instances of a clause set N.

For the purpose of this paper, all operators in $\Omega' \setminus \Omega$ are actually predicates implemented by functions mapping to the ordering minimal constant tt of the free sort. An equation of the form $p(t_1, \ldots, t_n) \approx \text{tt}$ is abbreviated by $P(t_1, \ldots, t_n)$, as usual. In addition, we only consider sets of clauses where the free part is Horn. A clause $\Lambda \parallel \Gamma \to \Delta$ is called a *Horn clause* if Δ contains at most one atom. $\square$ denotes the empty clause.

3 SMT for Non-linear Arithmetic

While most of the common *satisfiability modulo theories* (SMT) [3] approaches consider *decidable* theories, some authors have directed their attention to the

theory of non-linear arithmetic involving transcendental functions like exponential and trigonometric functions [13, 4, 14] which is *undecidable* in general. With regard to *superposition modulo non-linear arithmetic* considered in Section 4, this section is devoted to the logical framework of non-linear arithmetic SMT with a particular focus on an SMT solving algorithm.

While an *SMT formula wrt. the theory of non-linear arithmetic* can be an arbitrary quantifier-free Boolean combination of non-linear arithmetic constraints, like $\psi \equiv \big((\sin(y^2) \leq 0.1) \rightarrow (x \leq 0 \vee z > \sqrt{x^2+y^2})\big)$, it is common to deal with formulae of syntactically restricted shape for the sake of simplicity wrt. the development of SMT solving tools. Similar to propositional formulae in conjunctive normal form, we rewrite an arbitrary SMT formula as above into a *conjunction* of clauses where *clauses* are *disjunctions* of primitive constraints. A *primitive constraint* is an arithmetic predicate that contains one relational operator, at most one arithmetic operation which needs to be total[1], and up to three variables, e.g. $x \geq \sin(y)$, $x = y + z$, and $z < 3.7$. Resembling three-address code, each arithmetic constraint can be rewritten into a set of primitive constraints. SMT formulae of the above shape are called to be in *conjunctive form* or CF for short. In [16, Chapter 5], Herde presented a linear-time procedure to convert an arbitrary non-linear arithmetic SMT formula into an equi-satisfiable formula in CF. For the above formula ψ, an equi-satisfiable SMT formula in CF is $(h_{\sin(y^2)} > 0.1 \vee x \leq 0 \vee z > \sqrt{h_{x^2+y^2}}) \wedge (h_{\sin(y^2)} = \sin(h_{y^2})) \wedge (h_{y^2} = y^2) \wedge (h_{x^2+y^2} = h_{x^2} + h_{y^2}) \wedge (h_{x^2} = x^2)$ with fresh auxiliary variables $h_{\sin(y^2)}, h_{y^2}, h_{x^2+y^2}, h_{x^2}$ of sort $\mathbb{R}$. From the general semantics it follows that an SMT formula φ in CF is satisfied under an assignment τ iff at least one constraint in each clause of φ is satisfied under τ.

The iSAT algorithm [13] has been designed to address the satisfiability problem of non-linear arithmetic SMT formulae. The frontend of the iSAT tool automatically rewrites a given SMT formula into CF. In order to achieve termination of the interval-based approach, the domains of all variables of the given formula must be specified by *bounded* intervals.[2] The iSAT algorithm is a generalization of the Davis-Putnam-Logemann-Loveland (DPLL) procedure [8, 7] (with clause learning) using *interval constraint propagation* (ICP) [6]. Instead of real-valued assignments, iSAT manipulates *interval valuations* of the variables by alternating *deduction* and *splitting* phases, interspersed with *backtracking* whenever an empty interval valuation was detected.

During the *deduction* phase, the solver searches for clauses in which all but one atom are inconsistent under the current interval valuation. Such a remaining

[1] This is due to obviate the issue with undefined values of partial operations. Practically, this need not be a huge restriction as most common partial arithmetic operators can be expressed by their inverse operation. For example, the constraint $y = 1/x$ in which $1/x$ is undefined for $x = 0$ can be rephrased as $y \cdot x = 1 \wedge x \neq 0$.

[2] From a practical perspective, this prerequisite seems not to be too restrictive as variables encoding physical quantities like *temperature*, *velocity*, or *volume* are naturally bounded in their values. In cases where such an estimation should not be feasible for any reason, the lower and upper interval borders can be chosen arbitrarily small and arbitrarily large, respectively.

consistent atom is called *unit*. In order to retain a chance for satisfiability of the formula, unit atoms have to be satisfied. This is similar to unit propagation in SAT solving. The unit atoms are therefore used for ICP during the deduction phase. New interval bounds can thus be deduced until a fixed point is reached. Note that ICP can cause infinite deduction chains, e.g. for the (trivial) constraint $x = \frac{1}{2}x$ and $x \in [0, 1]$, ICP yields infinitely many interval contraction steps, namely $[0, 1] \rightsquigarrow [0, \frac{1}{2}] \rightsquigarrow [0, \frac{1}{4}] \rightsquigarrow [0, \frac{1}{8}] \rightsquigarrow \ldots$. There are, of course, more complicated situations that cannot be easily detected leading to this effect. In order to achieve termination, ICP is stopped if the progress of newly deduced bounds becomes negligible. If a *conflict* occurs, i.e. the interval of a variable becomes empty, then a conflict resolution procedure is called which analyzes the reason for the conflict. If the conflict cannot be resolved the given formula is unsatisfiable and iSAT stops with result No. Otherwise, a conflict clause is built (learnt) from the reason of the conflict and added to the formula in order to prevent the solver from revisiting the same conflict again. In order to retrieve a consistent solver state from which the proof search will be continued, conflict resolution involves *backtracking* that is undoing some of the decisions and their accompanying deductions that have been performed so far. It is worth mentioning that our current implementation of the iSAT algorithm is able to certify the unsatisfiability of a result. Such a certificate is produced during the conflict resolution and is very similar to a resolution proof. It consists of Boolean resolution and arithmetic deduction steps. A produced certificate can be easily verified using an external program. For a more detailed account please refer to [19]. If the solver finds a *solution*, i.e. at least one atom in each clause is satisfied by every point in the interval valuation, the algorithm stops with result Yes. In general, equations like $x = y \cdot z$ can only be satisfied by point intervals. However, reaching such point intervals by ICP cannot be guaranteed for continuous domains. One option to mitigate this problem is to stop the search when all intervals have a width smaller than a certain threshold, the so-called *minimum splitting width*. The resulting interval valuation can be considered as an *approximate* solution. Since the given problem could nevertheless not be decided, iSAT answers Unknown. Having completed the deduction phase and neither found a conflict nor an (approximate) solution, iSAT performs a decision by *splitting* an interval. A decision heuristics is used to select one of the intervals whose width is still greater than or equal to the minimum splitting width. The search is then resumed using this new interval bound which potentially triggers new deductions as described above.

As mentioned above, the core algorithm of iSAT is only able to detect satisfiability of a given formula φ if all points in the returned interval valuation satisfy φ. This strong condition cannot be expected as soon as formula φ includes some equations, as they are used for our experiments to encode the movement of objects. Consider the simple constraint $x = y + z$ and the interval valuation σ with $\sigma(x) = [0, 0.02]$, $\sigma(y) = [0, 0.01]$, and $\sigma(z) = [0, 0.01]$. Although σ actually contains a solution, e.g. $x = 0.01, y = 0.01, z = 0$, iSAT cannot conclude satisfiability since σ also contains some points that do not satisfy the constraint,

e.g. $x = 0.01, y = 0.01, z = 0.01$. To mitigate this dis-satisfactory issue, we implemented a technique, called *strong satisfaction check* [13], to certify satisfiability of non-linear arithmetic SMT formulae. Section 5 will report on the first successful implementation of this method.

4 Superposition Modulo Non-linear Arithmetic

We build on the framework of hierarchic superposition [2, 1] and shortly repeat the relevant notions for the SUP(NLA) combination considered here. Any given disjunction of literals can be transformed into a clause of the form $\Lambda \parallel \Gamma \to \Delta$, where Λ only contains terms of the base sort and all base terms in Γ, Δ are variables by introducing fresh variables for the respective subterms. We need to "purify" clauses only once – just before saturating the clauses, since if the premises of an inference are purified clauses, then the conclusion is also purified. For example, the clause $M(x_1, y_1, x_2, y_2, t) \to \delta < 0 \vee M(x_1 + t, y_1 + \cos(t) - \cos(t + \delta), x_2 - t, y_2 - (\cos(t) - \cos(t + \delta)))$ is purified to the clause $\delta \geq 0,\ t' \approx t + \delta,\ x_1' \approx x_1 + t,\ x_2' \approx x_2 - t,\ y_1' \approx y_1 + \cos(t) - \cos(t + \delta),\ y_2' \approx y_2 - (\cos(t) - \cos(t + \delta)) \parallel M(x_1, y_1, x_2, y_2, t) \to M(x_1', y_1', x_2', y_2', t')$. For the overall approach we consider the function-free Horn clause fragment. As usual we consider a reduction ordering $\prec$ for the free first-order symbols that is total on ground terms (atoms) and lifted to clauses. Ordering restrictions are solely calculated with respect to the free part.

Definition 1 (SUP(NLA)). *The superposition calculus consists of the inference rules superposition left and constraint refutation*

$$\mathcal{I}\frac{\Lambda_1 \parallel \Gamma_1 \to E_1 \qquad \Lambda_2 \parallel E_2, \Gamma_2 \to \Delta_2}{(\Lambda_1, \Lambda_2 \parallel \Gamma_1, \Gamma_2 \to \Delta_2)\sigma} \qquad \mathcal{I}\frac{\Lambda \parallel \to}{\square}$$

where for the superposition left rule σ is a simple and most general unifier of E_1 and E_2, $E_1\sigma$ is strictly maximal in $(\Gamma_1 \to E_1)\sigma$, and $E_2\sigma$ is maximal in $(E_2, \Gamma_2 \to \Delta_2)\sigma$; for the constraint refutation rule we require $\mathcal{M}_{\mathbb{R}} \models \exists \boldsymbol{x}\, [\bigwedge \Lambda]$.

Note that we restrict our attention to purely predicative Horn clauses. Therefore, the other inference rules superposition right and the factoring rules are not applicable. For simplicity, we don't consider selection nor sort constraints. Nevertheless, we still call this calculus superposition (and not resolution) because it comes with the important ingredients of superposition: an abstract redundancy criterion and an explicit model assumption. Both are substantial for our approach. The redundancy concept is indispensable for termination (see below). The explicit model assumption is indispensable for formalizing reachability (see Section 6). A clause $C \in N$ is called *redundant* if for all $C' \in sgi(C)$ there are clauses $C_1', \ldots, C_n' \in sgi(N)$ such that $C_1' \wedge \ldots \wedge C_n' \models C'$ and $C_i' \prec C'$ for all i. The concrete redundancy criteria considered here are *tautology* and *subsumption* deletion. A clause $\Lambda \parallel \Gamma \to \Delta$ is called a *tautology* iff $\models \forall \boldsymbol{x}\, [\bigwedge \Gamma \to \bigvee \Delta]$ or $\mathcal{M}_{\mathbb{R}} \not\models \exists \boldsymbol{x}\, [\bigwedge \Lambda]$. A clause $\Lambda_1 \parallel \Gamma_1 \to \Delta_1$ *subsumes* a clause $\Lambda_1 \parallel \Gamma_1 \to \Delta_1$ if for a simple matcher σ we have $\Gamma_1\sigma \subseteq \Gamma_2$, $\Delta_1\sigma \subseteq \Delta_2$, and $\mathcal{M}_{\mathbb{R}} \models \forall \boldsymbol{v} \exists \boldsymbol{u}\, [\bigwedge \Lambda_2 \Rightarrow \bigwedge \Lambda_1\sigma]$, where $\boldsymbol{v} = \mathrm{vars}(\Lambda_2)$ and

$\boldsymbol{u} = \mathrm{vars}(\Lambda_1\sigma) \setminus \mathrm{vars}(\Lambda_2)$. Note that the quantifier alternation is a direct consequence of the above defined abstract redundancy criterion for superposition. The hierarchic superposition calculus is complete [2] in the usual sense if the base theory is compact and all free function symbols ranging into the base sort are sufficiently defined. Both assumptions obviously hold, since compactness follows from the fact that the base theory is given by a standard model $\mathcal{M}_{\mathbb{R}}$, and we do not consider any free function symbols ranging into the NLA sort. However, in practice it happens that validity/satisfiability of NLA constraints cannot be decided and we obtain UNKNOWN from the NLA reasoning procedure. Therefore, we use the following more practical formulation of the completeness theorem, which adopts the results of [2].

Theorem 2 (SUP(NLA) Practical Completeness). *Let N be a set of Horn clauses from FOL(NLA) without free function symbols. Then N is unsatisfiable if SUP(NLA) derives $\square$; N is satisfiable, if the SUP(NLA) calculus terminates and the saturated set N^* does not contain $\square$ nor a clause $\Lambda \parallel \rightarrow$.*

The above version of the completeness theorem takes care of the fact that in practice an NLA procedure will not be able to decide the satisfiability of a constraint, in general. Then it may happen that clauses of the form $\Lambda \parallel \rightarrow$ can neither be refuted nor deleted and have to be kept. Thus being able to practically decide constraint satisfiability is crucial for precision. For termination the same applies to constraint implication, needed for subsumption. For the success of our experiments (Section 6), the following simplification pipeline turns out to be indispensable to this end.

Every time a new clause $\Lambda \parallel \Gamma \rightarrow \Delta$ is derived, the following simplifications are performed on the constraint Λ of the clause.

Constant propagation: if Λ contains a literal of the form $ax \approx b$, $(a, b \in \mathbb{R})$ then the substitution $\sigma = [x \mapsto b/a]$ is applied onto the constraint except the literal itself. Moreover, if the variable x does not occur in the free part of the clause, the literal is deleted after propagation.

Deletion of duplicates: if the constraint contains syntactically equivalent literals, say $\Lambda = L_1, \ldots, L_k, L_{k+1}, \ldots, L_n$, where $L_i = L_j$, for all $1 \leq i, j \leq k$, then only one of them is kept: $\Lambda' = L_1, L_{k+1}, \ldots, L_n$.

Product distribution over sum: every product $t \cdot \sum_{i=1}^{n} t_i$ occurring in Λ is transformed to the sum $\sum_{i=1}^{n} t \cdot t_i$.

Reduction of homogeneous summands: every sum $\sum_{i=1}^{n} a_i t + S$ occurring in Λ is reduced to $at + S$ where $a_1, \ldots, a_n, a$ are reals and $a = \sum_{i=1}^{n} a_i$, t – a term, S – the rest of the sum.

Reduction of homogeneous multipliers: every product $\prod_{i=1}^{k'} b_i \cdot \prod_{i=1}^{k} t^{a_i} \cdot P$ occurring in Λ is reduced to $bt^a \cdot P$, where $a_1, \ldots, a_k, a$ are naturals, $b_1, \ldots, b'_{k'}, b$ – reals, and $a = \sum_{i=1}^{k} a_i$, $b = \prod_{i=1}^{k'} b_i$, t are terms, P – the rest of the product.

Before an implication test between Λ_2 and $\Lambda_1\sigma$ takes place, the constraints are simplified in the above manner. Then variable-to-constant assignments occurring in the antecedent are propagated onto the succedent, whereupon the succedent is again simplified. Moreover every literal, occurring in the succedent

$\Lambda_1\sigma$ of the implication and having a syntactical equivalent in the antecedent Λ_2, is deleted from the succedent. Note that this simplification technology has potential for the SUP(NLA) calculus because constraints of newly generated clauses are always copies of the constraints from the parent clauses subject to a unifier mapping variables to variables. Then we replace every occurrence of the transcendental function symbols sin, cos, exp, nrt, pow with fresh uninterpreted function symbols $\sin'$, $\cos'$, $\exp'$, nrt', pow', respectively, and recursively rewrite every occurrence of terms with the top symbol being one of abs, min, or max in the following way: $\mathrm{abs}(t) \equiv \mathrm{ite}(t \geq 0, t, -t)$; $\min(s_1, \ldots, s_n) \equiv \mathrm{ite}(s_1 \leq \min(s_2, \ldots, s_n), s_1, \min(s_2, \ldots, s_n))$; $\max(s_1, \ldots, s_n) \equiv \mathrm{ite}(s_1 \geq \max(s_2, \ldots, s_n), s_1, \max(s_2, \ldots, s_n))$, where "ite" stands for the operator "*if then else*" available in most SMT systems. Then we check if the formula $\neg(\forall \boldsymbol{v}\, \exists \boldsymbol{u}\, [\Lambda_2 \rightarrow \Lambda_1\sigma])$ is unsatisfiable in the model of linear arithmetic plus uninterpreted functions by passing it to the SMT solver Z3 [10].

5 Strong Satisfaction

In the previous section (see Theorem 2), we have shown that the ability of SPASS(iSAT) to prove or disprove properties depends on iSAT's ability to conclusively decide whether a given formula is satisfiable or not. However, in Section 3, we have argued that iSAT alone may often terminate with result UNKNOWN. We will now try to bridge the gap by presenting an a-posteriori check that utilizes iSAT's inconclusive answer and an analysis of the formula in order to turn an UNKNOWN result into a definite YES. This check is referred to as *strong satisfaction check* that, if successful, actually gives a certificate of the existence of a solution.

For a motivating example, we assume that iSAT terminates with result UNKNOWN on SMT formula $\varphi = (x = \sin(y)) \wedge (x = y + z)$ over real variables $x, y, z \in [-100, 100]$, and returns the *approximate* solution σ with $\sigma(x) = [0.75, 0.85], \sigma(y) = [2.2, 2.3], \sigma(z) = [-1.5, -1.4]$. Though σ contains a solution, e.g. $y = 2.25$, $x = \sin(2.25)$, $z = \sin(2.25) - 2.25$, iSAT is not able to detect this since some points in σ violate φ, e.g. $y = 2.3$, $x = \sin(2.25)$, $z = \sin(2.25) - 2.25$. The core idea of the *strong satisfaction check* is to interpret above equations as assignments by giving them a direction with the condition that each variable is defined at most once. We then safely propagate small intervals through this equation system starting with intervals for non-defined variables. The latter intervals may be arbitrary but we use the approximate solution σ which heuristically is a good choice. This results in an interval valuation σ'. The first step is thus to (re)direct the equations s.t. each variable is defined at most once. In the example, we re-orient $x = y + z$ to $z = x - y$. Variable y is not defined by any assignment, so we set $\sigma'(y) = \sigma(y)$. Using equation $x = \sin(y)$, we determine the safe interval $\sigma'(x) = [0.74, 0.81]$, i.e. for each $v_y \in \sigma'(y)$ there is some $v_x \in \sigma'(x)$ s.t. $v_x = \sin(v_y)$. Finally, we compute $\sigma'(z) = [-1.56, -1.39]$ using $z = x - y$. Due to construction, σ' indeed contains a solution τ of φ, e.g. $\tau(y) = 2.25$, $\tau(x) = \sin(\tau(y))$, $\tau(z) = \tau(x) - \tau(y)$.

In general, let φ be a non-linear SMT formula in CF where the set of all variables occurring in φ is denoted by vars(φ). For technical reasons and w.l.o.g., we assume that the initial interval ranges of all variables in vars(φ) are encoded within φ, e.g. $x \in [-2.1, 5]$ is encoded by two clauses $(x \geq -2.1)$ and $(x \leq 5)$.[3] For a set PC of primitive constraints, we define the partition $PC = E(PC) \cup I(PC)$ into the set $E(PC)$ of equations of form $x = y \circ z$, $x = \circ y$, or $x = c$, and into a set $I(PC)$ of inequalities of form $x \sim y \circ z$, $x \sim \circ y$, or $x \sim c$, where x, y, z are variables, c is a constant, $\sim \in \{<, \leq, \geq, >\}$, and $\circ \in \{+, -, \cdot, \text{abs}, \text{min}, \text{max}, \text{sin}, \text{cos}, \text{exp}, \text{nrt}, \text{pow}\}$. Given an equation e of form $x = y \circ z$, or $x = \circ y$, or $x = c$ we call each of the equations $x = y \circ z$, $y = x \circ^z z$, $z = x \circ^y y$, or $x = \circ y$, $y = \circ^{-1} x$, or $x = c$, resp., *reshuffling of* e where $\circ^z$, $\circ^y$, and $\circ^{-1}$ are the corresponding inverse operations[4] of $\circ$. Recall that $\circ$ is a total operation. Thus, whenever a reshuffling (involving a potentially partial operation) is satisfied by an assignment τ then the original equation is also satisfied by τ. For a set $E = \{e_1, \ldots, e_k\}$ of equations, the set $R(E) = \{r_1, \ldots, r_k\}$ is called *reshuffling of* E iff each r_i is a reshuffling of e_i. We call an interval valuation σ *strongly satisfying for* φ, denoted $\sigma \models_{\text{s}} \varphi$, iff there is a set PC of primitive constraints and a reshuffling $R(E(PC)) = \{r_1, \ldots, r_k\}$ s.t. the following conditions hold:

1. Each clause C in φ contains at least one constraint in PC, i.e. $C \cap PC \neq \emptyset$.
2. Each inequality constraint in $I(PC)$ is satisfied by each point in σ.
3. For each reshuffling $r_i \in R(E(PC))$ of the form $x = y \circ z$, $x = \circ y$, or $x = c$ the following two conditions hold:
   (a) The left-hand side variable x is defined unambiguously in the sense that
      - $\sigma(x)$ is a point interval, i.e. $|\sigma(x)| = 1$, (this ensures that whenever x is defined by several reshufflings the value of x is unique) or
      - x does neither occur in any r_j with $j > i$ nor on the right-hand side of r_i (i.e., $x \neq y$ and $x \neq z$) (this ensures that x is defined by at most one reshuffling and no direct or indirect assignment cycles occur).

   (b) For all possible inputs of the right-hand side of r_i taken from σ there is a value in σ for the left-hand side s.t. reshuffling r_i is satisfied, i.e.
      - $\forall v_y \in \sigma(y)\ \forall v_z \in \sigma(z)\ \exists v_x \in \sigma(x) : v_x = v_y \circ v_z$,
      - $\forall v_y \in \sigma(y)\ \exists v_x \in \sigma(x) : v_x = \circ v_y$, or
      - $\exists v_x \in \sigma(x) : v_x = c$, resp.

Strong satisfaction is a *sufficient* condition for satisfiability as stated next.

Lemma 3. *If* $\sigma \models_{\text{s}} \varphi$ *then there exists an assignment* τ *such that* $\tau \models \varphi$.

In what follows, we briefly describe the essential algorithmic details of the strong satisfaction check. Condition 1 from above formal definition can in principle be satisfied by an arbitrary combination of constraints from all clauses. The first

[3] For integer variables like $y \in [-31, 89]$, we further assume here a clause $(y = -31 \vee \ldots \vee y = 89)$ that ensures an integer value for y. For efficiency reasons, such latter constraints are *implicitly* represented within the iSAT tool.

[4] It is important to remark that the inverse operator symbols must be only locally known in the strong satisfaction check and need not be part of the signature Ω.

heuristic choice is thus to find a combination for which the remaining conditions are likely to hold, especially condition 3b. Since we will make use of the approximate solution σ, taking constraints that are consistent under σ is a natural choice and can be easily realized. The core challenge that arises in a practical implementation of the strong satisfaction check emerges from the combination of condition 2 and condition 3: a sequence of assignments must be found such that no variable is assigned more than once and such that the deduced intervals do not violate any inequality constraint. For this purpose, we mark all variables for which the inequality constraints impose a very tight range of valid values as *a-priori feeders* and consider them to have already been assigned their specific small intervals or point-values. Such a-priori feeders potentially cause that other variables need to be defined by some equation, e.g. if x, y are feeders in equation $x = y + z$ then z becomes defined. The search for a strong satisfaction proof can fail in this step if a cycle or double definition of a variable is detected. Such a failure is not necessarily a sign that it is impossible to prove strong satisfaction just that the combination of chosen a-priori feeders and heuristically selected constraints cannot be used. While the strong satisfaction check is stopped in the current implementation, we will investigate different heuristics to continue with other a-priori feeders and constraints in future work.

If successful, however, there may still be several equations whose assignment direction needs to be determined. To solve this problem, we take essentially the algorithm from [18] which was an earlier attempt to tackle this problem. The core idea is to always select a variable that occurs in at most one constraint and to take it as the target for an assignment. By subsequently propagating backwards this direction through the remaining equation system, this will either lead to the empty system, i.e. a successful sequentialization which satisfies condition 3a, or the detection of a cycle in which case the algorithm fails to prove strong satisfaction. If successful, we thus have a sequence of assignments and a set of variables that have been reached by backwards traversal through the equation system and have thus become *feeder* variables.

For each feeder variable, we take the interval (or rather some value from it) specified by the approximate solution returned by iSAT. As shown in the example, we now perform interval propagation in the found assignment direction (using the MPFI library [21]) until an interval valuation σ for all variables is determined.[5] This step ensures that condition 3b is satisfied. If all remaining inequality constraints are satisfied by each point in σ then condition 2 is also satisfied, and it is therefore guaranteed by Lemma 3 that σ also contains a real-valued solution $\tau \in \sigma$, thus proving satisfaction of the given SMT formula.

We finally address the earlier implementation of the strong satisfaction check from [18]. This approach frequently fails on non-trivial formulae as it does not take into account the above mentioned issue of a-priori feeders. It turns out

[5] Note that we do not have to enclose all solutions, e.g. for $x := \arcsin(y)$ we only have to ensure that for every value of y a corresponding value for x is included in $\sigma(x)$ (cf. condition 3b). We use the approximate solution as a hint for which of the possibly infinitely many intervals is best and compute this one conservatively.

that this heuristics is of utmost importance in practice since it is pointless to use variables for which very tight ranges of valid values are imposed as non-feeder variables. For example, taking x as the defined variable in $x = y + z$ where inequalities $x \geq 3$ and $x \leq 3$ must be satisfied is very likely to fail as the intervals for y and z are non-point intervals in general.

6 Experiments

We have modelled a collision avoidance protocol, for which we want SPASS(iSAT) to fully automatically prove and disprove collision freeness. The idea behind the protocol for the basic 2D scenario is the following: initially there are two objects moving on straight lines in 2D-space (see Fig. 1); when the distance between the objects gets less or equal to some fixed value, they start maneuvering by sin-like trajectories such that at the beginning of the maneuver one of them goes up, the other goes down. The maneuver lasts for one period of sin, after that the objects continue straight line moving. Depending on the three involved parameters initial distance, distance to start the maneuver, and distance required for safety, the protocol yields or does not yield a collision.

More precisely, the behavior of the objects is modelled by a set of FOL(NLA) Horn clauses such that the minimal model of those clauses describes exactly the set of reachable states by the protocol. The minimal Horn clause model is identical to the SUP(NLA) model assumption for a set of Horn clauses. First-order predicates are used to model the reachable states. For example the clause $y_1 \geq y_2,\ \delta \geq 0,\ p \geq 3.1,\ p \leq 3.2,\ \cos(\frac{p}{2}) \approx 0,\ t'_m \approx t_m + \delta,\ t'_m \leq 2 \cdot p,\ \Delta_y = \cos(t_m) - \cos(t'_m),\ x'_1 \approx \delta,\ y'_1 \approx y_1 + \Delta_y,\ x'_2 \approx \delta,\ y'_2 \approx y_2 - \Delta_y \parallel M(x_1, y_1, x_2, y_2, t_m, p, x^0_1, y^0_1, x^0_2, y^0_2, t) \rightarrow M(x'_1, y'_1, x'_2, y'_2, t'_m, p, x^0_1, y^0_1, x^0_2, y^0_2, t)$ encodes part of the behavior of the two objects during the maneuver. Now the protocol is safe if there is no reachable state in the minimal model that causes a collision. This is obviously not a first-order property but can be attacked by superposition based reasoning as long as the safety condition has the closed form $\exists \boldsymbol{x}\ \phi$ where all first-order predicates in ϕ occur solely positively. In this case the negation of the safety condition results in a set of purely negative clauses. Then adding such a set to a set of Horn clauses the following holds [17]: (i) if $\square$ is derived, then the safety condition does not hold and a counter example can be extracted from the superposition proof; (ii) if SUP(NLA) terminates and neither $\square$ nor a clause $\Lambda \parallel \rightarrow$ is derived, then the safety condition holds (see also Theorem 2). This is exactly the way we proved (disproved) non-collision, by adding clauses of the form $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} < 4 \parallel M(x_1, y_1, x_2, y_2, t_m, p, x^0_1, y^0_1, x^0_2, y^0_2, t) \rightarrow$ to the axiomatization.

The 3D instance basic protocol extends the movement to 3D, where initially the objects are moving in parallel horizontal planes, and during the maneuver they are changing their heights following a sine-wave trajectory, see Fig. 2. The advanced 3D scenario adds to the basic scenario arbitrary initial 3D movement, see Fig. 3. Where in particular for this drawing we assume that the ends and starts of the arrows are time synchronization points between the objects.

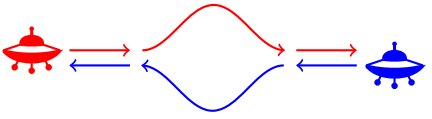


Fig. 1. Trajectories 2D

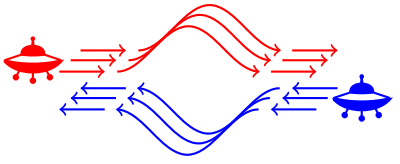


Fig. 2. Trajectories 3D

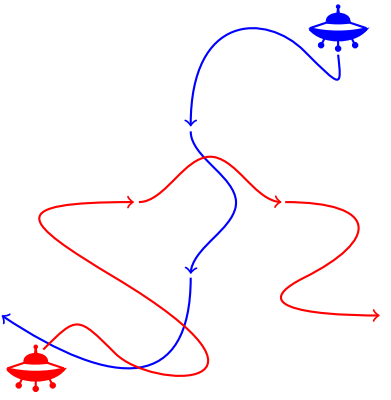


Fig. 3. Two objects with arbitrary movement

For all three scenarios we have proved and disproved safety by modifying the above mentioned parameters accordingly. Table 1 shows the timing for SPASS(iSAT) runs on all scenarios in seconds. The input as well as the output files are available from the SPASS homepage (`www.spass-prover.org/prototypes`). For each scenario we have ran one parameter setting where the protocol is collision free and SPASS(iSAT) finds a model and one setting where the objects collide, leading to a proof found by SPASS(iSAT). All runs were have been performed on computers equipped with Intel X5460 CPUs, 8 GB of main memory running Linux. Concerning all experiments we have done, more than 95% of the time has been spent by iSAT for solving NLA constraint proof obligations.

Table 1. Time Statistics

	2D	3D	
	linear	linear	arbitrary
Proof	10	9	41
Model	6	6	58

Table 2 shows the impact of our simplification pipeline for finding a proof in an unsafe parameter setting for the 3D arbitrary movement scenario. The first row shows which NLA simplification techniques developed in this paper have been omitted. So "None" means we have applied all, "SSC" means we have disabled the strongsat extension, "IA" means we have disabled the linear abstraction for implication testing, "IS" means we have disabled constraint simplification on the generated implication problems, and "CS" means we have disabled the basic constraint simplification techniques. Then the rest of the table shows the results of these settings on the testing of NLA problems during the run. If the basic constraint techniques have been disabled, SPASS(iSAT) does not terminate

on the problem. Note that disabling an NLA simplification technique typically results in extra clauses that cannot be subsumed nor detected as a tautology. Therefore, the set of generated and kept clauses for the different cases is different.

Table 2. NLA Simplification Impact for Proof Finding

		None	SSC	IA	IS	CS
Result		Proof	Unknown	Proof	Proof	-
Time (sec.)		45	520	38	290	out
Constr.	sat	356	20	364	355	-
	unsat	11	16	19	11	-
	unknown	0	440	0	1	-
Impl.	holds	281	296	256	280	-
	not holds	3040	3071	4218	6047	-
	unknown	0	3	0	29	-

Table 3 shows the respective impact of our simplification pipeline for finitely saturating the clause set in a safe parameter setting for the same scenario. Here both disabling CS or IA leads to non-termination. Disabling SSC leads to termination where the saturated clause set contains a clause $\Lambda \parallel \rightarrow$ for which iSAT without the strongsat extension cannot decide satisfiability of the constraint and returns UNKNOWN.

Table 3. NLA Simplification Impact for Model Finding

		None	SSC	IA	IS	CS
Result		Model	Unknown	-	Model	-
Time (sec.)		35	36	out	188	out
Constr.	sat	367	20	-	368	-
	unsat	31	28	-	29	-
	unknown	0	350	-	3	-
Impl.	holds	296	296	-	297	-
	not holds	3073	3071	-	6160	-
	unknown	1	3	-	44	-

7 Conclusion

To the best of our knowledge, the SUP(NLA) calculus presented here is the first (implemented) combination of first-order and non-linear arithmetic reasoning. Such combinations have been studied for linear arithmetic and first-order logic (e.g. [9, 5]). Also many of those approaches can in principle be extended to the NLA case and we believe that our simplification pipeline would be useful there as well.

Concerning the generated NLA constraint problems, we have tried to attack them by applying state-of-the-art computer algebra systems. To this end we have replaced iSAT by Maple[6]. But this approach has not been as successful as the iSAT combination as in many cases Maple has not been able to find a solution and therefore the overall solving process for the considered experiments here has failed. Redlog [11] shows a similar behavior. By appropriate approximations of the transcendental functions a reasonable portion of the constraints can be decided, however, in particular the crucial constraints (empty clauses, subsumption check) turn out to be specifically hard.

Collision avoidance protocols have been studied as benchmarks for various hybrid system analysis and verification tools (e.g. [15, 20, 22]). They are related to our collision avoidance protocol. However, these results are hard to compare as the models differ. For example, we have also considered an explicit 3D model where the above approaches all have developed 2D models.

We have developed the first sound and complete combination of FOL(NLA) including an implementation, where in particular, we can cope with UNKNOWN results by an NLA procedure (Theorem 2, Tables 2, 3). In order to decrease the number of UNKNOWN answers when executing the SUP(NLA) calculus we have suggested dedicated simplification techniques. All together with an implementation via SPASS(iSAT) the approach supports fully automatic verification of various scenarios of a non-trivial collision avoidance protocol. We are confident that by continuing the development of the suggested simplification techniques the performance of the overall procedure can be further significantly increased.

Acknowledgements. All authors are supported by the German Transregional Collaborative Research Center SFB/TR 14 AVACS.

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. AAECC 5(3/4), 193–212 (1994)

[3] Barrett, C., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Biere, A., Heule, M.J.H., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185, ch. 26, pp. 825–885. IOS Press, Amsterdam (2009)

[4] Bauer, A., Pister, M., Tautschnig, M.: Tool-support for the analysis of hybrid systems and models. In: DATE 2007, Nice, France, pp. 924–929 (2007)

[5] 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)

[6] Maple is a computer algebra system published by Maplesoft. For more details refer to http://www.maplesoft.com/products/maple/

[6] Benhamou, F., Granvilliers, L.: Continuous and interval constraints. In: Rossi, F., van Beek, P., Walsh, T. (eds.) Handbook of Constraint Programming. Foundations of Artificial Intelligence, ch. 16, pp. 571–603. Elsevier, Amsterdam (2006)
[7] Davis, M., Logemann, G., Loveland, D.: A Machine Program for Theorem Proving. CACM 5, 394–397 (1962)
[8] Davis, M., Putnam, H.: A Computing Procedure for Quantification Theory. Journal of the ACM 7(3), 201–215 (1960)
[9] de Moura, L.M., Bjørner, N.: Engineering DPLL(T) + saturation. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 475–490. Springer, Heidelberg (2008)
[10] de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
[11] Dolzmann, A., Sturm, T.: Redlog: Computer algebra meets computer logic. ACM SIGSAM Bulletin 31(2), 2–9 (1997)
[12] Eggers, A., Kruglov, E., Kupferschmid, S., Scheibler, K., Teige, T., Weidenbach, C.: Superposition modulo non-linear arithmetic. Report of SFB/TR 14 AVACS 80 (August 2011), http://www.avacs.org
[13] Fränzle, M., Herde, C., Teige, T., Ratschan, S., Schubert, T.: Efficient solving of large non-linear arithmetic constraint systems with complex Boolean structure. JSAT 1(3-4), 209–236 (2007)
[14] Gao, S., Ganai, M.K., Ivancic, F., Gupta, A., Sankaranarayanan, S., Clarke, E.M.: Integrating ICP and LRA solvers for deciding nonlinear real arithmetic problems. In: FMCAD 2010 (2010)
[15] Henzinger, T., Horowitz, B., Majumdar, R., Wong-Toi, H.: Beyond hytech: Hybrid systems analysis using interval numerical methods. In: Lynch, N.A., Krogh, B.H. (eds.) HSCC 2000. LNCS, vol. 1790, pp. 130–144. Springer, Heidelberg (2000)
[16] Herde, C.: Efficient Solving of Large Arithmetic Constraint Systems with Complex Boolean Structure: Proof Engines for the Analysis of Hybrid Discrete–Continuous Systems. Doctoral dissertation, Carlvon Ossietzky Universität Oldenburg (2010)
[17] Horbach, M., Weidenbach, C.: Superposition for fixed domains. ACM Transactions on Computational Logic 11(4), 1–35 (2010)
[18] Keddis, N.: Strong satisfaction. Bachelorthesis, Albert-Ludwigs-Universität Freiburg (September 2008)
[19] Kupferschmid, S., Becker, B., Teige, T., Fränzle, M.: Proof certificates and non-linear arithmetic constraints. In: IEEE Design and Diagnostics of Electronic Circuits and Systems. IEEE, Los Alamitos (2011)
[20] Platzer, A., Clarke, E.: The image computation problem in hybrid systems model checking. In: Bemporad, A., Bicchi, A., Buttazzo, G. (eds.) HSCC 2007. LNCS, vol. 4416, pp. 473–486. Springer, Heidelberg (2007)
[21] Revol, N., Rouillier, F., Chevillard, S., Lauter, C., Nguyen, H.D., Theveny, P.: Mpfi: Multiple precision floating-point interval arithmeticm, https://gforge.inria.fr/projects/mpfi/
[22] Tomlin, C.J., Pappas, G.J., Sastry, S.: Conflict resolution for air traffic management: A study in multi-agent hybrid systems. IEEE Transactions on Automatic Control 43(4), 509–521 (1998)
[23] Weidenbach, C., Dimova, D., Fietzke, A., Suda, M., Wischnewski, P.: SPASS version 3.5. In: Schmidt, R.A. (ed.) CADE-22. LNCS, vol. 5663, pp. 140–145. Springer, Heidelberg (2009)

The Modal Logic of Equilibrium Models

Luis Fariñas del Cerro and Andreas Herzig

University of Toulouse
IRIT, CNRS
http://www.irit.fr

Abstract. Here-and-there models and equilibrium models were investigated as a semantical framework for answer set programming by Pearce, Cabalar, Lifschitz, Ferraris and others. The semantics of equilibrium logic is indirect in that the notion of satisfiability is defined in terms of satisfiability in the logic of here-and-there. We here give a direct semantics of equilibrium logic, stated in terms of a modal language into which the language of equilibrium logic can be embedded.

Keywords: equilibrium logic, here-and-there models, bimodal logic, answer-set programming.

1 Introduction

A here-and-there model (HT model) is made up of two sets of propositional variables H ('here') and T ('there') such that $H \subseteq T$. The logical language to talk about such models has connectives $\bot$, $\wedge$, $\vee$, and $\Rightarrow$. The latter is interpreted in a non-classical way and is therefore different from the material implication $\rightarrow$:

$$H, T \models \varphi \Rightarrow \psi \quad \text{iff} \quad H, T \models \varphi \rightarrow \psi \text{ and } T, T \models \varphi \rightarrow \psi$$

where $\rightarrow$ is interpreted just as in classical propositional logic. Such models were studied since Gödel in order to give semantics to an implication with strength between intuitionistic and material implication [7]. They were later investigated by Pearce, Cabalar, Lifschitz, Ferraris and others as the basis of equilibrium logic, which is a semantical framework for answer set programming [10,9,11,2,3,6,8]; we refer to the equilibrium logic website[1] for an overview.

Equilibrium models of a formula φ are defined in an indirect way that is based on HT models: an equilibrium model of φ is a set of propositional variables T such that

1. $T, T \models \varphi$, and
2. there is no HT model (H, T) such that H is *weaker* than T and $H, T \models \varphi$,

where 'weaker' means that H is strictly included in T. For example, $T = \emptyset$ is an equilibrium model of $p \Rightarrow \bot$ because (1) for the HT model $(\emptyset, \emptyset)$ we have $\emptyset, \emptyset \models p \Rightarrow \bot$ and because (2) there is no set H that is strictly included in the empty set.

We here give a direct semantics of equilibrium logic in terms of a modal language having two unary modal operators [T] and [S]. Roughly speaking, [T] allows to talk

[1] http://www.equilibriumlogic.net

C. Tinelli and V. Sofronie-Stokkermans (Eds.): FroCoS 2011, LNAI 6989, pp. 135–146, 2011.

about the there-world: a valuation that is at least as strong as the actual valuation; and [S] allows to talk about all here-worlds that are possible if we take the actual world as a there-world: it quantifies over all valuations that are weaker than the actual world. This language is again interpreted on HT models. We also give a semantics in terms of Kripke models. We call our logic **MEM**: the Modal logic of Equilibrium Models.

We relate the language of equilibrium logic to our bimodal language by means of a translation *tr*. The main clause of the translation is:

$$tr(\varphi \Rightarrow \psi) = (tr(\varphi) \rightarrow tr(\psi)) \wedge [\mathrm{T}](tr(\varphi) \rightarrow tr(\psi))$$

We prove that φ has a HT model if and only if its translation $tr(\varphi)$ is satisfiable in **MEM**. This paves the way to the proof that φ is a consequence of χ in equilibrium logic if and only if the modal formula

$$[\mathrm{T}]\,(tr(\chi) \wedge [\mathrm{S}]\neg tr(\chi)) \rightarrow [\mathrm{T}]tr(\varphi)$$

is valid in **MEM**.

A first attempt to relate modal logic to equilibrium logic in the style of the present approach was presented in [5].

The paper is organised as follows. In Section 2 we introduce our modal logic of equilibrium models **MEM** both semantically and axiomatically. In Section 3 we recall the logic of here-and-there and equilibrium logic. In Section 4 we define the translation *tr* from the language of the logic of here-and-there to modal logic; we prove that a formula φ of the former language has a HT model if and only if $tr(\varphi)$ has a HT model, and that φ has an equilibrium model if and only if $[\mathrm{T}](tr(\varphi) \wedge [\mathrm{S}]\neg tr(\varphi))$ has a HT model. Section 5 concludes.

2 The Modal Logic of Equilibrium Models MEM

We now introduce the modal logic of equilibrium models **MEM** in the classical way: we start by defining its bimodal language and its semantics and then axiomatise its validities.

2.1 Language

Throughout the paper we suppose given a countably infinite set of propositional variables $\mathbb{P}$. The elements of $\mathbb{P}$ are noted p, q, etc. Our language $\mathcal{L}_{[\mathrm{T}],[\mathrm{S}]}$ is bimodal: it has two modal operators [T] and [S]. Precisely $\mathcal{L}_{[\mathrm{T}],[\mathrm{S}]}$ is defined by the following grammar:

$$\varphi ::= p \mid \bot \mid \varphi \rightarrow \varphi \mid [\mathrm{T}]\varphi \mid [\mathrm{S}]\varphi$$

where p ranges over $\mathbb{P}$. $[\mathrm{T}]\varphi$ may be read "φ holds at the there world" and $[\mathrm{S}]\varphi$ may be read "φ holds at every (strictly) weaker world".

The set of propositional variables occurring in formula φ is noted $\mathbb{P}_\varphi$.

$\mathcal{L}_{[\mathrm{T}]}$ is the sublanguage of $\mathcal{L}_{[\mathrm{T}],[\mathrm{S}]}$ formulas without [S], i.e. $\mathcal{L}_{[\mathrm{T}]}$ formulas are built from [T] and the Boolean connectives only.

We employ the standard abbreviations of the Boolean connectives: $\top \stackrel{\text{def}}{=} \bot \rightarrow \bot$, $\neg\varphi \stackrel{\text{def}}{=} \varphi \rightarrow \bot$, $\varphi \vee \psi \stackrel{\text{def}}{=} \neg\varphi \rightarrow \psi$, and $\varphi \wedge \psi \stackrel{\text{def}}{=} \neg(\varphi \rightarrow \neg\psi)$. Moreover, $\langle \mathrm{T}\rangle\varphi$ and $\langle \mathrm{S}\rangle\varphi$ respectively abbreviate $\neg[\mathrm{T}]\neg\varphi$ and $\neg[\mathrm{S}]\neg\varphi$.

2.2 Kripke Models

We interprete the formulas of our language $\mathcal{L}_{[T],[S]}$ in a class of Kripke models that has to satisfy particular constraints. We then give an axiomatisation of the validities of that class of models and prove its completeness.

Consider the class of Kripke models $M = \langle W, \mathcal{T}, \mathcal{S}, V\rangle$ such that

- W is a set of possible worlds;
- V is a valuation on W mapping possible worlds $w \in W$ to sets of propositional variables $V_w \subseteq \mathbb{P}$;
- $\mathcal{T}, \mathcal{S} \subseteq W \times W$ are relations on W such that:

(d) for every w there is a $v \in W$ such that $w\mathcal{T}v$;
(alt) for every w, if $w\mathcal{T}v$ and $w\mathcal{T}v'$ then $v = v'$;
(heredity) for every w, u, if $w\mathcal{S}u$ then $V_u \subseteq V_w$;
(fullpast) for every w, for every finite $P, Q \subseteq V_w$ such that P is nonempty, there is u such that: $w\mathcal{S}u$, $V_u \cap P = \emptyset$ and $Q \subseteq V_u$;
(mtrans) for every w, u, v, if $w\mathcal{S}u$ and $u\mathcal{T}v$ then $w\mathcal{T}v$;
(wconv) for every w, v, if $w\mathcal{T}v$ then $w = v$ or $v\mathcal{S}w$;

The first two constraints are about the relation $\mathcal{T}$, the next two are about the relation $\mathcal{S}$, and the last two are about both. Constraints (d) and (alt) say that at any world w there is exactly one possible world that is accessible via $\mathcal{T}$. The (heredity) constraint is just as the heredity constraint of intuitionistic logic, except that the intuitionistic relation is the inverse of $\mathcal{S}$. In the finite case the (fullpast) constraint basically says that for every w, the set of worlds accessible from w contains all those worlds u whose valuations V_u are included in V_w. The mixed transitivity constraint (mtrans) together with (d) and (alt) entails that in $\mathcal{S}$ connected parts of the graph M there is a unique there-world. The weak conversion constraint (wconv) says that $\mathcal{T}$ is contained in $\mathcal{S}^{-1} \cup id_W$, where id_W is the diagonal of W.

Let us denote by $\mathcal{T}(w)$ the unique world that is accessible from w via $\mathcal{T}$. The function $\mathcal{T}$ is well-defined because of constraints (d) and (alt).

Proposition 1. *Every Kripke model satisfies the following properties.*

1. *For every w, $\mathcal{T}(\mathcal{T}(w)) = \mathcal{T}(w)$.*
2. *For every w, u, if $w\mathcal{S}u$ then $\mathcal{T}(w) = \mathcal{T}(u)$.*
3. *For every w such that V_w is finite, the set $\{V_u : w\mathcal{S}u\}$ equals either $\{V : V \subseteq V_w\}$, or $\{V : V \subset V_w\}$.*

The last property is due to the (fullpast) constraint and says that for finite V_w, the set of valuations associated to the worlds that are accessible from w via $\mathcal{S}$ is either the set of subsets of V_w or the set of strict subsets of V_w: it equals either 2^{V_w} or $2^{V_w} \setminus \{V_w\}$. This will be used in the proof of Proposition 8.

2.3 Truth Conditions

The truth conditions for our bimodal logic are standard. The relation $\mathcal{T}$ interprets [T] and $\mathcal{S}$ interprets [S]:

$$\begin{array}{ll} M, w \models p & \text{iff } p \in V_w; \\ M, w \not\models \bot; & \\ M, w \models \varphi \rightarrow \psi & \text{iff } M, w \not\models \varphi \text{ or } M, w \models \psi \\ M, w \models [\mathrm{T}]\varphi & \text{iff } M, \mathcal{T}(w) \models \varphi; \\ M, w \models [\mathrm{S}]\varphi & \text{iff } M, u \models \varphi \text{ for every } u \text{ such that } w\mathcal{S}u. \end{array}$$

When $M, w \models \varphi$ then we say that φ has a Kripke model. Moreover, φ is *valid in Kripke models* if and only if $M, w \models \varphi$ for every model M and possible world w of M. Finally, φ is *satisfiable in Kripke models* if and only if $\neg\varphi$ is invalid in Kripke models, i.e. if and only if $M, w \models \varphi$ for some model M and possible world w of M.

The next proposition says that when checking satisfaction it is enough to only consider models with finite valuations.

Proposition 2. *Let φ be a $\mathcal{L}_{[\mathrm{T}],[\mathrm{S}]}$ formula. Let $M = \langle W, \mathcal{T}, \mathcal{S}, V\rangle$ be a Kripke model satisfying (d), (alt), (heredity), (fullpast), (mtrans), and (wconv). Let the valuation V^φ be defined as follows:*

$$V^\varphi_w = V_w \cap \mathbb{P}_\varphi, \text{ for every } w \in W$$

Then $M^\varphi = \langle W, \mathcal{T}, \mathcal{S}, V^\varphi\rangle$ is a Kripke model satisfying (d), (alt), (heredity), (fullpast), (mtrans), and (wconv), and $M, w \models \varphi$ if and only if $M^\varphi, w \models \varphi$.

Proof. That $M, w \models \varphi$ if and only if $M^\varphi, w \models \varphi$ can be shown by straightforward induction on the form of φ.

As to the constraints, those that are only about the accessibility relations are clearly preserved because we just modify the valuation. The model M^φ satisfies constraint (heredity): suppose $w\mathcal{S}u$; as M satisfies (heredity) we have $V_u \subseteq V_w$; hence $V^\varphi_u \subseteq V^\varphi_w$. Finally, the model M^φ satisfies (fullpast): suppose $P, Q \subseteq V^\varphi_w = V_w \cap \mathbb{P}_\varphi$ are finite sets such that $P \neq \emptyset$; as M satisfies (fullpast) there is u such that $w\mathcal{S}u$ and $V_u \cap P = \emptyset$ and $Q \subseteq V_u$. Clearly, for that u we also have $V^\varphi_u \cap P = \emptyset$; and for that very u we also have $Q \subseteq V^\varphi_u = V_u \cap \mathbb{P}_\varphi$. q.e.d.

We note that this property is different from the standard finite model property of modal logics which requires a finite set of possible worlds.

2.4 Axiomatics, Decidability, and Complexity

We now give an axiomatisation of the **MEM** validities.

First we define the fragment of *positive Boolean formulas* of $\mathcal{L}_{[\mathrm{T}],[\mathrm{S}]}$ by the following grammar:

$$\varphi^+ ::= p \mid \varphi^+ \wedge \varphi^+ \mid \varphi^+ \vee \varphi^+$$

Observe that every positive formula is falsifiable. (Note that $\top$ is not a positive Boolean formula.)

Table 1. Axiomatisation of **MEM**

K([T])	the axioms and inference rules of modal logic **K** for [T]	
K([S])	the axioms and inference rules of modal logic **K** for [S]	
D([T])	$[T]\varphi \to \langle T\rangle\varphi$	
Alt([T])	$\langle T\rangle\varphi \to [T]\varphi$	
Heredity([S])	$\langle S\rangle\varphi^+ \to \varphi^+$	for φ^+ a positive Boolean formula
Negatable([S])	$(\varphi^+ \wedge \psi^+) \to \langle S\rangle(\neg\varphi^+ \wedge \psi^+)$	for φ^+, ψ^+ positive Boolean formulas s.th. $\mathbb{P}_{\varphi^+} \cap \mathbb{P}_{\psi^+} = \emptyset$
MTrans([S], [T])	$[T]\varphi \to [S][T]\varphi$	
WConv([T], [S])	$\varphi \to [T](\varphi \vee \langle S\rangle\varphi)$	

Our axiom schemas and inference rules are listed in Table 1. The axiom schemas D([T]) and Alt([T]) are familiar from standard textbooks on modal logic. The schema Heredity([S]) captures the heredity constraint of intuitionistic logic. Note that it could be replaced by the axiom schema $\langle S\rangle p \to p$, where p is a propositional variable. It could also be replaced by $\neg\varphi^+ \to [S]\neg\varphi^+$, for φ^+ a positive Boolean formula. The schema Negatable([S]) ensures that the modal operator [S] quantifies over all strict subsets of the actual valuation. The schema MTrans([S], [T]) is an axiom of mutual transitivity. The schema WConv([T], [S]) is a weak conversion axiom familiar from tense logics.

The notions of a proof and of *provability of a formula* are defined as usual in modal logic. For example $[S]\bot \to \neg p$ can be proved from Negatable([S]) by **K**([S]), i.e. by standard modal principles. The proof of the transitivity axiom $[T]\varphi \to [T][T]\varphi$ and its converse are provable is a bit longer.

Proposition 3. *The schema* $[T]\varphi \leftrightarrow [T][T]\varphi$ *is provable.*

Proof.

1.	$[T]\varphi \to [T]([T]\varphi \vee \langle S\rangle[T]\varphi)$	(axiom WConv([T], [S]))
2.	$\langle S\rangle[T]\varphi \to \langle S\rangle\langle T\rangle\varphi$	(axiom D([T]) and **K**([S]))
3.	$\langle S\rangle\langle T\rangle\varphi \to \langle T\rangle\varphi$	(axiom MTrans([S], [T]))
4.	$\langle T\rangle\varphi \to [T]\varphi$	(axiom Alt([T]))
5.	$\langle S\rangle[T]\varphi \to [T]\varphi$	(from 2, 3, 4)
6.	$[T]\varphi \to [T]([T]\varphi \vee [T]\varphi)$	(from 1 and 5)
7.	$[T]\varphi \to [T][T]\varphi$	(from 6)
8.	$[T][T]\varphi \to \langle T\rangle\langle T\rangle\varphi$	(axiom D([T]) twice, and **K**([S]))
9.	$\langle T\rangle\varphi \to \langle T\rangle\langle T\rangle\varphi$	(from 4, 7, 8)
10.	$[T]\varphi \leftrightarrow [T][T]\varphi$	(from 7, 9)

q.e.d.

The next schema is also going to be useful.

Proposition 4. *The following formula schema is provable:*

$$\textit{Negatable}'([\mathrm{S}]) \quad \Big((\bigwedge_{p\in P} p) \wedge (\bigwedge_{q\in Q} q)\Big) \rightarrow \langle \mathrm{S}\rangle \Big((\bigwedge_{p\in P} \neg p) \wedge (\bigwedge_{q\in Q} q)\Big)$$
$$\textit{for } P, Q \subseteq \mathbb{P} \textit{ finite, } P \textit{ nonempty, and } P \cap Q = \emptyset$$

Proof. Negatable′([S]) can be proved from Negatable([S]) as follows. Suppose $P, Q \subseteq \mathbb{P}$ finite, P nonempty, and $P \cap Q = \emptyset$. The implication

$$((\bigwedge_{p\in P} p) \wedge (\bigwedge_{q\in Q} q)) \rightarrow ((\bigvee_{p\in P} p) \wedge (\bigwedge_{q\in Q} q))$$

is valid in classical propositional logic. Then Negatable′([S]) follows with the axiom schema Negatable([S]). q.e.d.

Our axiomatisation is sound and complete w.r.t. the set of formulas that are **MEM** valid.

Theorem 1. *Let φ be a $\mathcal{L}_{[\mathrm{T}],[\mathrm{S}]}$ formula. φ is valid in Kripke models of* **MEM** *if and only if φ is provable from the axioms and inference rules of* **MEM**.

Proof.
Soundness is proved as usual. We just consider the case of axiom Negatable([S]). Let φ^+ and ψ^+ be positive Boolean formulas such that $\mathbb{P}_{\varphi^+} \cap \mathbb{P}_{\psi^+} = \emptyset$. Suppose $M, w \models \varphi^+ \wedge \psi^+$. Put φ^+ in conjunctive normal form, and let $\kappa = (\bigvee P)$ be some clause of that CNF, for some $P \subseteq \mathbb{P}_{\varphi^+} \neq \emptyset$. (Observe that $P \neq \emptyset$ by the definition of positive formulas.) Let $P_w = P \cap V_w$. We have $P_w \neq \emptyset$ because $M, w \models \kappa$. Let $Q = \mathbb{P}_\varphi \setminus P_w$. As M satisfies (fullpast) there is a $u \in W$ such that $u\mathcal{T}w$, $V_u \cap P_w = \emptyset$ and $Q \subseteq V_w$. Hence $M, u \not\models \kappa$, and therefore $M, u \not\models \varphi^+$. As $\mathbb{P}_{\varphi^+} \cap \mathbb{P}_{\psi^+} = \emptyset$ and as V_u differs from V_w only by variables from $\mathbb{P}_{\varphi^+}$ we also have $M, u \models \psi^+$. Hence $M, u \models \neg\varphi^+ \wedge \psi^+$, and therefore $M, w \models \langle \mathrm{S}\rangle(\neg\varphi^+ \wedge \psi^+)$.

To prove completeness w.r.t. Kripke models of **MEM** we use canonical models [1,4]. Consider the set W of maximal consistent sets of **MEM**. Define the accessibility relations $\mathcal{T}$ and $\mathcal{S}$ on W by:

$$u\mathcal{T}w \text{ iff } \{\varphi : [\mathrm{T}]\varphi \in u\} \subseteq w$$
$$u\mathcal{S}w \text{ iff } \{\varphi : [\mathrm{S}]\varphi \in u\} \subseteq w$$

and define a valuation V such that $V_w = w \cap \mathbb{P}$ for every $w \in W$. Let us prove that the canonical model is a legal Kripke model of **MEM**.

- Axioms D([T]) and Alt([T]) ensure that $\mathcal{T}$ is a total function, i.e. the canonical model satisfies constraints (d) and (alt).
- Axiom Heredity([S]) ensures that the canonical model satisfies the heredity constraint, viz. that $w\mathcal{S}u$ implies $V_u \subseteq V_w$. Indeed, suppose $w\mathcal{S}u$ and $p \in u$. As u contains $\langle \mathrm{S}\rangle p \rightarrow p$ and as w is maximal consistent we have $p \in w$.

- Axiom Negatable([S]) guarantees the (fullpast) constraint. To see this take some $w \in W$ and two finite sets of propositional variables $P, Q \subseteq w \cap V_w$ such that P is nonempty. As w is a maximal consistent set it contains $(\bigwedge_{p\in P} p) \wedge (\bigwedge_{q\in Q} q)$. As by Proposition 4 w contains every instance of Negatable′([S]), it must also contain $\langle S\rangle((\bigwedge_{p\in P} \neg p) \wedge (\bigwedge_{q\in Q} q))$. Hence by definition of $\mathcal{S}$ there is some $u \in W$ such that $u\mathcal{S}w$ and u contains $(\bigwedge_{p\in P} \neg p) \wedge (\bigwedge_{q\in Q} q)$. Therefore $P \cap u = \emptyset$ and $Q \subseteq u$.
- The weak conversion axiom WConv([T], [S]) ensures constraint (wconv).
- The mixed transitivity axiom MTrans([S], [T]) ensures constraint (mtrans).

Hence the canonical model satisfies all constraints, and is therefore a legal Kripke model of **MEM**.

The proof of the truth lemma is as usual. q.e.d.

3 HT Logic and Equilibrium Logic

In this section we are going to formally define HT logic and equilibrium logic.

3.1 The Language $\mathcal{L}_{\Rightarrow}$

The language $\mathcal{L}_{\Rightarrow}$ is common to HT logic and equilibrium logic. It is defined by the following grammar:

$$\varphi ::= p \mid \bot \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \Rightarrow \varphi$$

where p ranges over $\mathbb{P}$. The other Boolean connectives are defined as abbreviations in the same way as for our bimodal language: negation $\neg\varphi$ is defined as $\varphi \Rightarrow \bot$, and $\top$ is defined as $\bot \Rightarrow \bot$.

3.2 Here-and-There Logic

A *HT model* is a couple (H, T) such that $H \subseteq T \subseteq \mathbb{P}$. The set T is called 'there' and H is called 'here'.

Let (H, T) be a HT model. The truth conditions for $\mathcal{L}_{\Rightarrow}$ formulas are as follows:[2]

$$\begin{array}{ll}
H, T \models p & \text{iff } p \in H \\
H, T \not\models \bot & \\
H, T \models \varphi \wedge \psi & \text{iff } H, T \models \varphi \text{ and } H, T \models \psi \\
H, T \models \varphi \vee \psi & \text{iff } H, T \models \varphi \text{ or } H, T \models \psi \\
H, T \models \varphi \Rightarrow \psi & \text{iff } H, T \models \varphi \rightarrow \psi \text{ and } T, T \models \varphi \rightarrow \psi
\end{array}$$

When $H, T \models \varphi$ then we say that (H, T) is a HT model of φ. A formula φ is *HT valid* if and only if every HT model is also a HT model of φ.

[2] In the last clause we use material implication as a shorthand in order to give a concise formulation. To spell this out, the truth condition for $\rightarrow$ is the standard condition for material implication: $H, T \models \varphi \rightarrow \psi$ iff $H, T \not\models \varphi$ or $H, T \models \psi$.

3.3 Equilibrium Logic

An *equilibrium model* of a $\mathcal{L}_{\Rightarrow}$ formula φ is a set of propositional variables $T \subseteq \mathbb{P}$ such that

1. (T, T) is a HT model of φ;
2. no (H, T) with $H \subset T$ is a HT model of φ.

Here are two examples. First, the empty set is the only equilibrium model of both $\top$ and $\neg p$: for example $\{q\}$ has the strict subset $\emptyset$ such that $\emptyset, \{q\} \models \top$ and $\emptyset, \{q\} \models p$. Second, the set $\{p\}$ is *not* an equilibrium model of $\neg p \Rightarrow q$ because $\emptyset, \{p\} \not\models \neg p \Rightarrow q$.

Let φ and χ be $\mathcal{L}_{\Rightarrow}$ formulas. φ is a *consequence of* χ *in equilibrium models*, written $\chi \models_{HT^*} \varphi$, if and only if for every equilibrium model T of χ, (T, T) is an HT model of φ. For example we have $\top \models_{HT^*} \neg p$ and $\neg p \Rightarrow q \models_{HT^*} q$.

4 From HT Logic and Equilibrium Logic to Modal Logic

In this section we are going to translate HT logic and equilibrium logic into our logic **MEM**.

4.1 Translating $\mathcal{L}_{\Rightarrow}$ to $\mathcal{L}_{[\mathrm{T}]}$

To start we translate the language $\mathcal{L}_{\Rightarrow}$ of both HT logic and equilibrium logic into the language $\mathcal{L}_{[\mathrm{T}]}$ of **MEM**. We recursively define the mapping tr as follows:

$$\begin{array}{ll} tr(p) & = p \quad \text{for } p \in \mathbb{P} \\ tr(\bot) & = \bot \\ tr(\varphi \wedge \psi) & = tr(\varphi) \wedge tr(\psi) \\ tr(\varphi \vee \psi) & = tr(\varphi) \vee tr(\psi) \\ tr(\varphi \Rightarrow \psi) & = (tr(\varphi) \rightarrow tr(\psi)) \wedge [\mathrm{T}](tr(\varphi) \rightarrow tr(\psi)) \end{array}$$

This translation combines the Gödel translation from intuitionistic logic to modal logic **S4** with Boolos's splitting translation from modal logic **S4** to modal logic **K4**. The main clause of the former is $tr(\varphi \Rightarrow \psi) = \Box(tr(\varphi) \rightarrow tr(\psi))$, for some **S4** operator $\Box$. The main clause of the latter is $tr(\Box\varphi) = tr(\varphi) \wedge [\mathrm{T}]tr(\varphi)$, where [T] is a **K4** operator (the operator of our bimodal logic).

Here are some examples.

$$tr(\top) = tr(\bot \Rightarrow \bot) = (\bot \rightarrow \bot) \wedge [\mathrm{T}](\bot \rightarrow \bot).$$

The latter is equivalent to $\top$ in any normal modal logic.

$$tr(\neg p) = tr(p \Rightarrow \bot) = (p \rightarrow \bot) \wedge [\mathrm{T}](p \rightarrow \bot).$$

This is equivalent to $\neg p \wedge [\mathrm{T}]\neg p$ in any normal modal logic.

$$tr(p \vee \neg p) = tr(p) \vee tr(p \Rightarrow \bot) = p \vee ((p \rightarrow \bot) \wedge [\mathrm{T}](p \rightarrow \bot)).$$

This is equivalent to $p \vee [\mathrm{T}]\neg p$ in any normal modal logic.

Observe that translated formulas may be exponentially longer than the original formulas.

Our translation will be used to relate both HT logic and equilibrium logic to **MEM**.

4.2 From HT Logic to MEM

The fragment $\mathcal{L}_{[T]}$ of the language $\mathcal{L}_{[T],[S]}$ is at least as expressive on HT models as $\mathcal{L}_{\Rightarrow}$, modulo the translation tr.

Proposition 5. *Let T be a set of propositional variables and let $M_T = \langle W, \mathcal{T}, \mathcal{S}, V\rangle$ be a quadruple such that:*

$$\begin{aligned} W &= 2^T; \\ V_h &= h, \quad \text{for every } h \in W; \\ \mathcal{T} &= W \times \{T\}; \\ \mathcal{S} &= \supset. \end{aligned}$$

Then M is a **MEM** *model, and $(H, T) \models \varphi$ if and only if $M_T, H \models tr(\varphi)$, for every $H \subseteq T$ and for every $\mathcal{L}_{\Rightarrow}$ formula φ.*

So in the last line $\mathcal{S}$ is defined to be the strict superset relation on 2^T. For example for the HT model $(\emptyset, \emptyset)$ we obtain $M_\emptyset = \langle W, \mathcal{T}, \mathcal{S}, V\rangle$ with $W = \{\emptyset\}$, $\mathcal{T} = \{\langle\emptyset, \emptyset\rangle\}$, and $\mathcal{S} = \emptyset$; and for the HT model $(\emptyset, \{p\})$ we obtain $M_{\{p\}} = \langle W, \mathcal{T}, \mathcal{S}, V\rangle$ with $W = \{\emptyset, \{p\}\}$, $\mathcal{T} = \{\langle\emptyset, \{p\}\rangle, \langle\{p\}, \{p\}\rangle\}$, and $\mathcal{S} = \{\{p\}, \emptyset\}\}$.

Proof. First, M is a legal **MEM** model: M satisfies constraints (d), (alt), (heredity), (fullpast), (mtrans), and (wconv). Second, one can prove by a straightforward induction on the form of φ that $H, T \models \varphi$ iff $M, T \models \varphi$, for every $H \subseteq T$. q.e.d.

Proposition 6. *Let $M = \langle W, \mathcal{T}, \mathcal{S}, V\rangle$ be a* **MEM** *model. Then $M, w \models tr(\varphi)$ if and only if $V_w, V_{\mathcal{T}(w)} \models \varphi$, for every $w \in W$ and for every $\mathcal{L}_{\Rightarrow}$ formula φ.*

Proof. As expected the proof is by induction on the form of φ. The only non trivial case is that of the intuitionistic implication $\psi_1 \Rightarrow \psi_2$. We have:

$$\begin{aligned} M, w \models tr(\psi_1 \Rightarrow \psi_2) \text{ iff } & M, w \models tr(\psi_1) \rightarrow tr(\psi_2) \text{ and } M, \mathcal{T}(w) \models tr(\psi_1) \rightarrow tr(\psi_2) \\ \text{iff } & V_w, V_{\mathcal{T}(w)} \models \psi_1 \rightarrow \psi_2 \text{ and } V_{\mathcal{T}(w)}, V_{\mathcal{T}(w)} \models \psi_1 \rightarrow \psi_2 \quad \text{(by I.H.)} \\ \text{iff } & V_w, V_{\mathcal{T}(w)} \models \psi_1 \Rightarrow \psi_2 \end{aligned}$$

q.e.d.

Theorem 2. *Let φ be a $\mathcal{L}_{\Rightarrow}$ formula. Then φ is HT valid if and only if $tr(\varphi)$ is* **MEM** *valid.*

Proof. This follows from Proposition 5 and Proposition 6. q.e.d.

4.3 From Equilibrium Logic to MEM

The same construction as for HT logic allows us to turn equilibrium models into **MEM** models.

Proposition 7. *Let $T \subseteq \mathbb{P}$ and let $M_T = \langle W, \mathcal{T}, \mathcal{S}, V\rangle$ be a quadruple such that:*

$$\begin{aligned} W &= 2^T; \\ V_h &= h, \quad \text{for every } h \in W; \\ \mathcal{T} &= W \times \{T\}; \\ \mathcal{S} &= \supset. \end{aligned}$$

Then M_T is a **MEM** *model, and T is an equilibrium model of φ if and only if $M_T, \mathcal{T}(H) \models tr(\varphi) \wedge [\mathrm{S}]\neg tr(\varphi)$, for every $H \subseteq T$ and for every $\mathcal{L}_{\Rightarrow}$ formula φ.*

Proof. First of all, M_T is a legal **MEM** model as in Proposition 5. By definition T is an equilibrium model of φ if and only if $T, T \models tr(\varphi)$ and $H, T \not\models tr(\varphi)$ for every $H \subset T$. So Proposition 5 tells us that T is an equilibrium model of φ if and only if $M_T, T \models tr(\varphi)$ and $M_T, H \not\models tr(\varphi)$ for every $H \subset T$. As $T\mathcal{S}H$ iff $H \subset T$, it follows that the latter is the case if and only if $M_T, T \models tr(\varphi)$ and $M_T, H \not\models tr(\varphi)$ for every H such that $T\mathcal{S}H$, i.e. if and only if $M_T, T \models tr(\varphi) \wedge [\mathrm{S}]\neg tr(\varphi)$. Therefore $M_T, \mathcal{T}(H) \models [\mathrm{T}](tr(\varphi) \wedge [\mathrm{S}]\neg tr(\varphi))$ for every $H \in W$ (because T is the only element of W such that $H\mathcal{T}T$). q.e.d.

Proposition 8. *Let $M = \langle W, \mathcal{T}, \mathcal{S}, V\rangle$ be a* **MEM** *model, let $q \in \mathbb{P} \setminus \mathbb{P}_\varphi$ be a propositional variable not occurring in φ, and let T be defined as:*

$$T = \begin{cases} V_{\mathcal{T}(w)} & \text{if } V_u \subset V_w \text{ for every } u \text{ such that } w\mathcal{S}u \\ V_{\mathcal{T}(w)} \cup \{q\} & \text{if } V_u = V_w \text{ for some } u \text{ such that } w\mathcal{S}u \end{cases}$$

Then $M, \mathcal{T}(w) \models tr(\varphi) \wedge [\mathrm{S}]\neg tr(\varphi)$ if and only if T is an equilibrium model of φ, for every $w \in W$.

Proof. By Proposition 2 we may suppose w.l.o.g. that V_w is finite for every $w \in W$. We consider two cases.

The first case is when $V_u \subset V_w$ for every u such that $w\mathcal{S}u$. By item 3 of Proposition 1 the set of $\mathcal{S}$ accessible worlds equals the set of strict subsets of V_w. Therefore:

$M, \mathcal{T}(w) \models tr(\varphi) \wedge [\mathrm{S}]\neg tr(\varphi)$
iff $M, \mathcal{T}(w) \models tr(\varphi)$ and $M, u \not\models tr(\varphi)$ for every u such that $\mathcal{T}(w)\mathcal{S}u$
iff $V_{\mathcal{T}(w)}, V_{\mathcal{T}(w)} \models \varphi$ and $V_u, V_{\mathcal{T}(w)} \not\models \varphi$ for every u such that $\mathcal{T}(w)\mathcal{S}u$ (by Prop. 6)
iff $V_{\mathcal{T}(w)}, V_{\mathcal{T}(w)} \models \varphi$ and $H, V_{\mathcal{T}(w)} \not\models \varphi$ for every $H \subset V_{\mathcal{T}(w)}$ (v.s.)
iff $T, T \models \varphi$ and $H, T \not\models \varphi$ for every $H \subset T$

Second, if $V_u = V_w$ for some u such that $w\mathcal{S}u$ then we have $T = V_{\mathcal{T}(w)} \cup \{q\}$. Therefore:

$M, \mathcal{T}(w) \models tr(\varphi) \wedge [\mathrm{S}]\neg tr(\varphi)$
iff $M, \mathcal{T}(w) \models tr(\varphi)$ and $M, u \not\models tr(\varphi)$ for every u such that $\mathcal{T}(w)\mathcal{S}u$
iff $V_{\mathcal{T}(w)}, V_{\mathcal{T}(w)} \models \varphi$ and $V_u, V_{\mathcal{T}(w)} \not\models \varphi$ for every u such that $\mathcal{T}(w)\mathcal{S}u$ (by Prop. 6)
iff $V_{\mathcal{T}(w)}, V_{\mathcal{T}(w)} \models \varphi$ and $H, V_{\mathcal{T}(w)} \not\models \varphi$ for every $H \subseteq V_{\mathcal{T}(w)}$ (v.s.)
iff $V_{\mathcal{T}(w)} \cup \{q\}, V_{\mathcal{T}(w)} \cup \{q\} \models \varphi$ and $H, V_{\mathcal{T}(w)} \cup \{q\} \not\models \varphi$ for every $H \subseteq V_{\mathcal{T}(w)} \cup \{q\}$
iff $T, T \models \varphi$ and $H, T \not\models \varphi$ for every $H \subset T$

q.e.d.

For example consider the set $T = \emptyset$ and the formula $\varphi = \top$. We have seen above that $\emptyset$ is the only equilibrium model of $\top$. Likewise, $(\emptyset, \emptyset)$ is the only HT model of $[\mathrm{T}]\,(tr(\top) \wedge [\mathrm{S}]\neg tr(\top))$. This can be seen by simplifying the latter:

$$\begin{aligned}[\mathrm{T}]\,(tr(\top) \wedge [\mathrm{S}]\neg tr(\top)) &\leftrightarrow [\mathrm{T}]\,(\top \wedge [\mathrm{S}]\neg\top)\\ &\leftrightarrow [\mathrm{T}][\mathrm{S}]\bot\end{aligned}$$

As we have seen, the only HT model of $[\mathrm{T}][\mathrm{S}]\bot$ is $(\emptyset, \emptyset)$.

We are now ready for the grand finale where we capture equilibrium logic in our bimodal logic.

Theorem 3. *Let φ and χ be $\mathcal{L}_{\Rightarrow}$ formulas. Then $\chi \models_{HT^*} \varphi$ if and only if*

$$[\mathrm{T}]\,(tr(\chi) \wedge [\mathrm{S}]\neg tr(\chi)) \rightarrow [\mathrm{T}]tr(\varphi)$$

is **MEM** *valid.*

Proof. This follows from Proposition 7 and Proposition 8. q.e.d.

Let us consider an example. We have seen that $\top \models_{HT^*} \neg p$, i.e. that $\neg p$ is a consequence of $\top$ in equilibrium models. We have seen in Section 4.1 that $tr(\top)$ is equivalent to $\top$ and that $tr(\neg p)$ is equivalent to $\neg p \wedge [\mathrm{T}]\neg p$. Theorem 3 tells us that the formula $\varphi = [\mathrm{T}](tr(\top) \wedge [\mathrm{S}]\neg tr(\top)) \rightarrow [\mathrm{T}](tr(\neg p))$ must be provable from the axioms and inference rules of **MEM**. This can be established by the following sequence of equivalent formulas:

1. $[\mathrm{T}](tr(\top) \wedge [\mathrm{S}]\neg tr(\top)) \rightarrow [\mathrm{T}](tr(\neg p))$
2. $[\mathrm{T}](\top \wedge [\mathrm{S}]\neg\top) \rightarrow [\mathrm{T}](\neg p \wedge [\mathrm{T}]\neg p)$ (v.s.)
3. $[\mathrm{T}][\mathrm{S}]\bot \rightarrow ([\mathrm{T}]\neg p \wedge [\mathrm{T}][\mathrm{T}]\neg p)$ (by **K**([T]))
4. $[\mathrm{T}][\mathrm{S}]\bot \rightarrow ([\mathrm{T}]\neg p \wedge [\mathrm{T}]\neg p)$ (by Prop. 3)
5. $[\mathrm{T}][\mathrm{S}]\bot \rightarrow [\mathrm{T}]\neg p$

The last line is provable in our logic: indeed, we have seen that $[\mathrm{S}]\bot \rightarrow \neg p$ can be proved from Negatable([S]) by standard modal principles. From this we can prove the last formula in our list by standard modal principles. Therefore the original formula φ is provable in our logic.

5 Conclusion

In this paper we have investigated the modal logic **MEM** that is behind equilibrium logic. We have shown that a logic with two modal operators [T] and [S] allows to capture the minimisation that is only expressed in the metalanguage in the standard definition of equilibrium models. We have shown that **MEM** satisfiability is decidable and that can be checked in polynomial space. We have also given a sound and complete axiomatisation.

It remains to give a lower bound for the complexity of **MEM**. It also remains to design a translation from the language of equilibrium logic to that of our bimodal logic

that avoids exponential growth of the formula length. This can however be done in a quite straightforward way by integrating a modal operator $[\mathrm{T}]_*$ whose truth condition in HT models is:

$$H, T \models [\mathrm{T}]_*\varphi \text{ iff } H, T \models \varphi \text{ and } T, T \models \varphi$$

In terms of Kripke models $[\mathrm{T}]_*$ is interpreted by the reflexive closure of the accessibility relation $\mathcal{T}$ interpreting [T]. However, a drawback of the addition of a third modal operator is that the formalism gets more cumbersome.

Acknowledgements. This work was partially supported by the French-Spanish *Laboratoire Européen Associé (LEA) "French-Spanish Lab of Advanced Studies in Information Representation and Processing"*.

Thanks are due to David Pearce and Levan Uridia for their explanations about equilibrium logic. Thanks are also due to the FroCoS reviewers whose comments helped to improve the paper.

References

1. Blackburn, P., de Rijke, M., de Venema, Y.: Modal Logic. Cambridge Tracts in Theoretical Computer Science. University Press (2001)
2. Cabalar, P., Ferraris, P.: Propositional theories are strongly equivalent to logic programs. Theory and Practice of Logic Programming (TPLP) 7(6), 745–759 (2007)
3. Cabalar, P., Pearce, D., Valverde, A.: Minimal Logic Programs. In: Dahl, V., Niemelä, I. (eds.) ICLP 2007. LNCS, vol. 4670, pp. 104–118. Springer, Heidelberg (2007)
4. Carnielli, W.A., Pizzi, C., Bueno-Soler, J.: Modalities and Multimodalities. In: Logic, Epistemology, and the Unity of Science. Springer, Heidelberg (2009)
5. Fariñas del Cerro, L., Herzig, A.: Contingency-Based Equilibrium Logic. In: Delgrande, J.P., Faber, W. (eds.) LPNMR 2011. LNCS, vol. 6645, pp. 223–228. Springer, Heidelberg (2011), http://www.springerlink.com/
6. Ferraris, P., Lee, J., Lifschitz, V.: A new perspective on stable models. In: Veloso, M.M. (ed.) IJCAI, pp. 372–379 (2007)
7. Heyting, A.: Die formalen Regeln der intuitionistischen Logik. Sitzungsber. preuss. Akad. Wiss. 71, 158–169 (1930)
8. Lifschitz, V.: Thirteen Definitions of a Stable Model. In: Blass, A., Dershowitz, N., Reisig, W. (eds.) Fields of Logic and Computation. LNCS, vol. 6300, pp. 488–503. Springer, Heidelberg (2010)
9. Lifschitz, V., Pearce, D., Valverde, A.: Strongly equivalent logic programs. ACM Transactions on Computational Logic 2(4), 526–541 (2001)
10. Pearce, D.: A new logical characterisation of stable models and answer sets. In: Dix, J., Przymusinski, T.C., Moniz Pereira, L. (eds.) NMELP 1996. LNCS, vol. 1216, pp. 57–70. Springer, Heidelberg (1997)
11. Pearce, D., de Guzmán, I.P., Valverde, A.: A tableau calculus for equilibrium entailment. In: Dyckhoff, R. (ed.) TABLEAUX 2000. LNCS, vol. 1847, pp. 352–367. Springer, Heidelberg (2000)

Harnessing First Order Termination Provers Using Higher Order Dependency Pairs*

Carsten Fuhs[1] and Cynthia Kop[2]

[1] RWTH Aachen University, LuFG Informatik 2, 52056 Aachen, Germany
fuhs@informatik.rwth-aachen.de
[2] Vrije Universiteit, De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands
kop@few.vu.nl

Abstract. Many functional programs and higher order term rewrite systems contain, besides higher order rules, also a significant first order part. We discuss how an automatic termination prover can split a rewrite system into a first order and a higher order part. The results are applicable to all common styles of higher order rewriting with simple types, although some dependency pair approach is needed to use them.

Keywords: Higher order rewriting, termination, dependency pairs, modularity.

1 Introduction

Termination of term rewrite systems has been an area of active research for several decades. In recent years the field of *automatically* proving termination has flourished, and several strong provers have been developed to participate in the annual *International Termination Competition*; there is a wide range of automated methods available for (and used in) these tools: the dependency pair framework [3,17,14], polynomial and matrix orderings [9,8], recursive path orderings [7], semantic labelling [36], and many more techniques.

In higher order termination, however, fewer results have been obtained so far. Recursive and monotonic semantic path orderings have been generalised to a higher order setting [21,5,6], but other automatable term orderings have not (yet?) been extended to this setting.

In the last three years there has been a lot of work on higher order dependency pair approaches and several strong results have been obtained, such as the ability to use argument filterings and to restrict to non-collapsing dependency pairs [28,32,26]. But after simplifying the ordering requirements on terms with this approach, we still have little but a higher order RPO to compare them.

However, in many (realistic) term rewrite systems, only a small number of the rules use functional variables or λ-abstraction. The majority of the rules usually consists entirely of first order symbols. It would therefore be convenient to analyse termination of at least those rules directly with first order techniques.

* This research is supported by the G.I.F. grant 966-116.6 and by the Netherlands Organisation for Scientific Research (NWO-EW) under grant 612.000.629 (HOT).

C. Tinelli and V. Sofronie-Stokkermans (Eds.): FroCoS 2011, LNAI 6989, pp. 147–162, 2011.

The progress in dependency pair approaches opens possibilities: we can split the dependency pairs into those which could be considered first order and the higher order remainder, and analyse termination of these parts separately. While the first order dependency pairs classically still need to be regarded together with *all* rules from the underlying system (some of which are higher order), it may be possible to remove these higher order rules, or replace them by first order ones.

In this paper we discuss how to reduce the termination of an orthogonal or finitely branching higher order term rewrite system to the termination of a first order (sub-)system and a (smaller) dependency pair problem. The technique is comparable to a *usable rules* [14,18] approach, but focusses on first order rules. We aim to be as general as possible by not choosing a definition of dependency pairs and assuming as little as possible about the formalism. Consequently, the results presented in this paper can be used for all the common styles of higher order rewriting, and with both dynamic and static dependency pairs [31,32].

We have implemented the method in the higher order termination prover WANDA [24], using the tool AProVE [12] to analyse termination of the first order part of a higher order rewrite system. As far as we know, this is the first time a tool for termination of higher order rewriting is combined with a first order termination tool. Experimental results (see Section 5) demonstrate that this combination significantly improves the strength of the prover.

Higher Order Rewriting. "Higher order rewriting", rewriting with some form of functional variables, comes in several forms: typed and untyped, with and without λ-abstraction. To understand the relevance of this work, it should be noted that these styles are *fundamentally different*.

Without giving complete definitions, consider the system with two function symbols: $\mathtt{app} : \mathtt{o} \Rightarrow \mathtt{o} \Rightarrow \mathtt{o}$ (which takes two arguments of type $\mathtt{o}$ and returns an object of type $\mathtt{o}$) and $\mathtt{lam} : (\mathtt{o} \Rightarrow \mathtt{o}) \Rightarrow \mathtt{o}$ (which takes a functional argument of type $\mathtt{o} \Rightarrow \mathtt{o}$ and returns an object of type $\mathtt{o}$), and a single rule $\mathtt{app}\ (\mathtt{lam}\ F)\ x \to F\ x$. In simply-typed applicative systems, terms are built from typed constants and a binary application operator. The given system terminates, because the size of a term decreases with every reduction step. In higher order systems with λ-abstraction, β-reduction may increase the size of a term. Here, defining $\omega = \mathtt{lam}\ (\lambda x.\mathtt{app}\ x\ x)$, there is a loop $\mathtt{app}\ \omega\ \omega \to (\lambda x.\mathtt{app}\ x\ x)\ \omega \to_\beta \mathtt{app}\ \omega\ \omega$.

Since terms in a formalism with λ-abstraction may include *anonymous functions* (such as $\lambda x.\mathtt{app}\ x\ x$) whose presence may give rise to non-termination, these formalisms cannot easily be simulated with applicative systems. In addition, in an applicative system it is impossible to express rules like this derivation rule:

$$\mathtt{D}\ (\lambda x.\mathtt{sin}(Z(x))) \to \lambda x.(\mathtt{D}\ (\lambda y.Z(y))\ x) \times \mathtt{cos}(Z(x))$$

As we will see below, applicative systems can be transformed into standard first order TRSs via some kind of uncurrying; thus, this work is primarily relevant for formalisms which do have λ-abstraction.

Related Work. Other work on using first order techniques in higher order rewriting is often focussed on applicative systems, where only terms without

binders like λ are considered. By [22], currying untyped TRSs does not affect termination and certain other properties. In [19] an uncurrying transformation from untyped applicative systems to standard first order TRSs is used, which preserves and reflects termination. The result of [13] is similar, but works on a more restricted set of problems, and presents termination techniques that operate directly on applicative systems. In [1,2] simply typed applicative systems are discussed; here, leading variables are eliminated by instantiating them with "template" terms of the right type. Having this, they can be transformed into many-sorted TRSs.

However, these results do not apply to systems with binders and β-reduction, nor does it seem likely that they can easily be extended to such a formalism.

In [11] termination is studied for Haskell programs, a (higher order) polymorphic functional language, via a translation to first order term rewriting. The approach relies on symbolic partial evaluation of a start term, which is made feasible essentially by Haskell's deterministic evaluation strategy. In a general term rewrite setting, however, there is no fixed strategy, which renders the construction from [11] infeasible. Moreover, we are interested in termination of *all* terms, while the construction in [11] considers only terms of a given form.

In this paper, we consider typed higher order rewriting which may have binders; we show how *part* of a higher order termination problem can be dealt with as a first order problem (leaving the truly higher order part to higher order techniques). An early work in this context, [33], considers termination of the combination of typed λ-calculus with first order TRSs. A first modularity result with higher order rules is given in [20], where the authors show that a terminating first order system combined with a number of higher order rules is terminating if the higher order rules satisfy certain restrictions, and the first order part is non-duplicating. The restriction on the first order rules is not present in the current work, nor do we pose limitations on the higher order part.

Another relevant work is [32], which studies static dependency pairs for a subset of the HRS formalism and defines a usable rules approach. The usable rules for a set of first order dependency pairs are all first order. However, this approach does not give an equivalence result like our Theorem 9. In addition, we do not choose a definition of dependency pairs or a formalism.

2 Preliminaries

As stated in the introduction, we aim for generality. Rather than focussing on a formalism, we will discuss the basic definitions used in common styles of higher order rewriting with simple types. Consequently, these definitions are incomplete, but our results can be used for instance with AFSs [21], HRSs [29] and CRSs [23].

Types. Given a set of *base types* $\mathcal{B}$, *types* are built according to the grammar:

$$\mathcal{T} = \mathcal{B} \mid \mathcal{T} \Rightarrow \mathcal{T}$$

The $\Rightarrow$ associates to the right; a type of the form $\sigma \Rightarrow \tau$ is called *functional*. Every type can be written in the form $\sigma_1 \Rightarrow \ldots \Rightarrow \sigma_n \Rightarrow \iota$ with $n \geq 0$ and $\iota \in \mathcal{B}$.

Terms. A *term* is an expression s over a set $\mathcal{F}$ of typed function symbols and a set $\mathcal{V}$ of typed variables, for which we can derive $s : \sigma$ for some type σ using the following recursive rules (which also define the set $FV(s)$ of *free variables* of s):

(`var`) $x : \tau$	if $x : \tau \in \mathcal{V}$	$FV(x) = \{x\}$
(`fun`) $f : \tau$	if $f : \tau \in \mathcal{F}$	$FV(f) = \emptyset$
(`abs`) $\lambda x.s : \sigma \Rightarrow \tau$	if $x : \sigma \in \mathcal{V}$ and $s : \tau$	$FV(\lambda x.s) = FV(s) \setminus \{x\}$
(`app`) $s \cdot t : \tau$	if $s : \sigma \Rightarrow \tau$ and $t : \sigma$	$FV(s \cdot t) = FV(s) \cup FV(t)$

The $\cdot$ operator associates to the left and is usually omitted; a term $s\ t\ r$ is short for $(s \cdot t) \cdot r$. We consider term equality modulo renaming of bound variables (α-conversion), so $\lambda x.s = \lambda y.s[x := y]$ if y does not occur in s.
Note that this is a general definition of terms; there are several higher order formalisms which do not allow, for instance, a term $(\lambda x.s) \cdot t$, or $f\ s : \sigma \Rightarrow \tau$.

Define *head*(s), the *head symbol* of s, as the first part of an application: *head*$(s) = s$ if s is a variable, constant or abstraction, and *head*$(u\ v) =$ *head*(u).

Meta-Terms. Some formalisms, like Klop's CRSs [23] or Blanqui's definition of IDTSs [4], use special *meta-terms* to construct rules. A meta-term is a typed expression generated with clauses (`var`), (`fun`), (`abs`), (`app`) and additionally:

$$(\texttt{meta})\ Z(s_1, \ldots, s_n) : \tau \text{ if } s_1 : \sigma_1, \ldots, s_n : \sigma_n \text{ and } Z : [\sigma_1, \ldots, \sigma_n] \Rightarrow \tau \in \mathcal{MV}$$

where $\mathcal{MV}$ is a fresh set of *meta-variables*, each equipped with a vector of input types ($\sigma_1, \ldots, \sigma_n$, where n may be 0) and an output type (τ); the s_i are meta-terms. Evidently, all terms are also meta-terms. Meta-terms can be used to match a term which may contain some bound variables, for instance in a rule like:

$$\texttt{map}\ (\lambda x.F(x))\ (\texttt{cons}\ h\ t) \to \texttt{cons}\ F(h)\ (\texttt{map}\ (\lambda x.F(x))\ t)$$

Note that not all higher order formalisms use meta-variables; for instance Jouannaud's and Okada's AFSs [21] use variables for matching instead, at the price of some (easy) expressivity. Nipkow's HRSs [29] also use variables, but here terms are equivalence classes modulo β/η, which is not always a practical modelling. In the examples in this paper, we will use meta-variables to define rules.

Contexts and Subterms. A *context* is a term containing one occurrence of a special symbol $\Box_\sigma : \sigma$. Contexts are usually denoted as $C[]$, and $C[]$ with $\Box_\sigma$ replaced by some t of type σ is denoted $C[t]$. If $s = C[t]$, then t is a *subterm* of s, denoted $s \trianglerighteq t$. If C is non-empty, then t is a strict subterm of s, denoted $s \triangleright t$.

Substitutions. A *substitution* is a type-preserving function mapping variables and meta-variables to terms; substitutions on a finite domain are usually denoted $[x_1 := s_1, \ldots, x_n := s_n]$. A substitution γ may be applied on (meta-)terms by placewise replacing variables and meta-variables by their image in γ; depending on the rewriting formalism the result might be β-normalised. Formally:

$$\begin{array}{rll}
x\gamma = x & & \text{if } x \in \mathcal{V},\ x \notin \text{dom}(\gamma) \\
x\gamma = \gamma(x) & & \text{if } x \in \mathcal{V},\ x \in \text{dom}(\gamma) \\
(f\ s_1 \cdots s_n)\gamma = f\ (s_1\gamma) \cdots (s_n\gamma) & & (f \in \mathcal{F},\ n \geq 0) \\
((\lambda x.q)\ s_1 \cdots s_n)\gamma = (\lambda x.q\gamma)\ (s_1\gamma) \cdots (s_n\gamma) & & (n \geq 0, **) \\
Z(s_1, \ldots, s_n)\gamma = q[x_1 := s_1\gamma, \ldots, x_n := s_n\gamma] & & \text{if } \gamma(Z) = \lambda x_1 \ldots x_n.q
\end{array}$$

(**): When substituting an abstraction $\lambda x.q$, the variable x may not occur in either domain or range of γ. Using α-conversion, this is always defined. We assume that $\gamma(Z)$ has the form $\lambda x_1 \dots x_n.q$ whenever $Z : [\sigma_1, \dots, \sigma_n] \Rightarrow \tau \in \mathcal{MV}$.

This definition is incomplete. The cases where formalisms differ, in particular $(x\ s_1 \cdots s_n)\gamma$ with $n \geq 1$, are omitted. However, the given cases are the only ones we will need.

Rules. A *rewrite rule* is a pair $l \to r$ of (meta-)terms such that l and r have the same type and $head(l)$ is a function symbol or abstraction. Let $\mathcal{R}$ be a (possibly infinite) set of rewrite rules. The *rewrite relation* $\to_{\mathcal{R}}$ generated by $\mathcal{R}$ is given by: $s \to_{\mathcal{R}} t$ if $s = C[l\gamma]$, $t = C[r\gamma]$ for some $l \to r \in \mathcal{R}$, substitution γ and context C; write $s \to_{\mathcal{R},top} t$ if C is empty and $s \to_{\mathcal{R},in} t$ otherwise.
Depending on the formalism, this rewrite relation may only be defined on terms of a given form, for instance β/η-normal form; however, base-type variables and terms $f\ s_1 \cdots s_n$ of base type always have such a form if the s_i do.

A set of rules $\mathcal{R}$ is *finitely branching* if, for any term s, there are only finitely many different t with $s \to_{\mathcal{R}} t$. This is commonly the case when $\mathcal{R}$ is finite. A set of rules is *terminating* if there is no infinite reduction $s_0 \to_{\mathcal{R}} s_1 \to_{\mathcal{R}} \dots$

Example 1. An example system we will use throughout this paper is the system $\mathcal{R}_{\texttt{list}}$, a module for list manipulation, with the following function symbols:

$$\begin{array}{lll} \texttt{nil} : \texttt{list} & \texttt{append} : \texttt{list} \Rightarrow \texttt{list} \Rightarrow \texttt{list} & \texttt{reverse} : \texttt{list} \Rightarrow \texttt{list} \\ \texttt{cons} : \texttt{nat} \Rightarrow \texttt{list} & \texttt{map} : (\texttt{nat} \Rightarrow \texttt{nat}) \Rightarrow \texttt{list} \Rightarrow \texttt{list} & \texttt{shuffle} : \texttt{list} \Rightarrow \texttt{list} \\ & & \texttt{mirror} : \texttt{list} \Rightarrow \texttt{list} \end{array}$$

And moreover ten rules:

$$\begin{aligned} \texttt{append}\ \texttt{nil}\ l &\to l \\ \texttt{append}\ (\texttt{cons}\ h\ t)\ l &\to \texttt{cons}\ h\ (\texttt{append}\ t\ l) \\ \texttt{reverse}\ \texttt{nil} &\to \texttt{nil} \\ \texttt{reverse}\ (\texttt{cons}\ h\ t) &\to \texttt{append}\ (\texttt{reverse}\ t)\ (\texttt{cons}\ h\ \texttt{nil}) \\ \texttt{shuffle}\ \texttt{nil} &\to \texttt{nil} \\ \texttt{shuffle}\ (\texttt{cons}\ h\ t) &\to \texttt{cons}\ h\ (\texttt{shuffle}\ (\texttt{reverse}\ t)) \\ \texttt{mirror}\ \texttt{nil} &\to \texttt{nil} \\ \texttt{mirror}\ (\texttt{cons}\ h\ t) &\to \texttt{append}\ (\texttt{cons}\ h\ (\texttt{mirror}\ t))\ (\texttt{cons}\ h\ \texttt{nil}) \\ \texttt{map}\ (\lambda x.F(x))\ \texttt{nil} &\to \texttt{nil} \\ \texttt{map}\ (\lambda x.F(x))\ (\texttt{cons}\ h\ t) &\to \texttt{cons}\ F(h)\ (\texttt{map}\ (\lambda x.F(x))\ t) \end{aligned}$$

There is only one really higher order function symbol (`map`), as its rules use an abstraction. Intuitively, the first eight rules are first order. Note that `mirror` has a duplicating rule, so the result from [20] cannot be used to prove termination.

Remarks. Despite our aim for generality, we do make a number of assumptions:

- the requirement that $head(l) \notin \mathcal{V}$ for left-hand sides l of a rule is not present in Yamada's STTRSs [35] or (certain variations of) Jouannaud's AFSs [21];
- we use applicative rather than functional notation ($f\ s_1 \cdots s_n$ rather than $f(s_1, \dots, s_n)$), where the latter is used in AFSs and Blanqui's IDTSs [4];
- unlike in AFSs, we do not assume the presence of a β-rule;
- unlike in CRSs or ERSs, typing is enforced;

– we assume monomorphic, simple types, while more advanced type classes are regularly used in several of the formalisms.

Only the first of these is essential: all the proofs in this paper pass almost unmodified even if we use functional notation and admit polymorphic types or include a β-rule. We chose this definition because, through simple transformations, we can usually obtain a system as described above without affecting termination: for the removal of head-variables and currying see for instance [25], to ignore typing embed abstractions into some new symbol $T : (\mathtt{term} \Rightarrow \mathtt{term}) \Rightarrow \mathtt{term}$, to add β-reduction create, for every two types σ, τ, a rule $(\lambda x.Z(x))\ y \to Z(y)$ with $Z : [\sigma] \Rightarrow \tau \in \mathcal{MV}$, and for dealing with (ML-style) polymorphism, instantiate all type variables in all closed ways and consider types of the form $\mathtt{list}(\mathtt{nat})$ as base types. These last two transformations lead to an infinite system, but only in so far as infinity was already implicit in the formalism. If the original system was finitely branching or orthogonal, the same holds for the result.

Variables or Meta-Variables. Due to our aim of giving formalism-independent definitions, matching may be done either with variables or meta-variables. To ease definitions, we will identify meta-variables without arguments with variables. Thus, a meta-variable $Z : [] \Rightarrow \sigma$ is considered as a variable of type σ. In the $\mathcal{R}_{\mathtt{list}}$ example, l, h and t can be seen as variables, while F is a meta-variable.

3 Splitting the System

To give some formal backing to the intuitive notion of a first order rule, we partition the signature $\mathcal{F}$ into two groups: symbols which have some higher order potential (i.e., they have a non-base type, there is a rule where they are not given all arguments allowed by their type, or they match on or rewrite to such symbols) and symbols which do not. The first group, *potentially higher order* symbols, is denoted $\mathtt{PHO}$ and the second one, consisting of *truly first order* symbols, is denoted $\mathtt{TFO}$. Using this partitioning, we obtain the first order rules by uncurrying the rules which only contain symbols in $\mathtt{TFO}$.

Splitting the Symbols. Let A be the set consisting of those function symbols $f : \sigma_1 \Rightarrow \ldots \Rightarrow \sigma_n \Rightarrow \iota$ (with $\iota \in \mathcal{B}$) such that one of the σ_i is functional, or there is a rule $f\ s_1 \cdots s_m \to r$ where $m < n$ or the rule contains any abstraction, meta-variable with arguments or functional (meta-)variable. We define $\mathtt{PHO}$ recursively: $\mathtt{PHO}$ contains all symbols in A and, if there is a rule $f\ \boldsymbol{l} \to r$ where some l_i or r contains a symbol in $\mathtt{PHO}$, then also $f \in \mathtt{PHO}$. Let $\mathtt{TFO} = \mathcal{F} \setminus \mathtt{PHO}$. A term is *truly first order* if it consists only of function symbols in $\mathtt{TFO}$ and base-type variables.

Example 2. In $\mathcal{R}_{\mathtt{list}}$ we have $A = \{\mathtt{map}\}$. Since the symbol $\mathtt{map}$ only occurs in the $\mathtt{map}$-rules, we have $\mathtt{PHO} = \{\mathtt{map}\}$ and hence $\mathtt{TFO} = \{\mathtt{nil}, \mathtt{cons}, \mathtt{append}, \mathtt{reverse}, \mathtt{shuffle}, \mathtt{mirror}\}$. Should we add a symbol $\mathtt{up} : \mathtt{list} \Rightarrow \mathtt{list}$ and a rule $\mathtt{up}\ l \to \mathtt{map}\ (\lambda x.\mathtt{s}\ x)\ l$, then A would still be $\{\mathtt{map}\}$, but $\mathtt{PHO}$ would also include $\{\mathtt{up}\}$.

Splitting the Rules. We say that a *rule* $f\ l_1 \cdots l_n \to r$ is *truly first order* if $f \in \mathtt{TFO}$ and *potentially higher order* otherwise; write $\mathcal{R}_{\mathtt{TFO}}$ for the set of rules

of the first kind and $\mathcal{R}_{\texttt{PHO}}$ for the second. Note that if a rule is truly first order then both sides are truly first order terms.

Example 3. The truly first order rules $\mathcal{R}_{\texttt{TFO}}$ of $\mathcal{R}_{\texttt{list}}$ are those whose left-hand side has a head symbol from $\{\texttt{append}, \texttt{reverse}, \texttt{shuffle}, \texttt{mirror}\}$. The potentially higher order rules $\mathcal{R}_{\texttt{PHO}}$ are those with `map` as the head symbol.

We can safely assume that in the truly first order rules $l \to r$, all variables in r also occur in l: if this is not the case, the system is obviously non-terminating.

Splitting Infinite Chains. A term rewrite system (first or higher order) is non-terminating iff there exists a *(minimal) infinite chain* $s_1, t_1, s_2, t_2, \ldots$ where:

- each $s_i \to_{\mathcal{R},top} \cdot \trianglerighteq t_i$
- each $t_i \to^*_{\mathcal{R},in} s_{i+1}$
- the strict subterms of each of the t_i are terminating

This observation is at the heart of any dependency pair approach. Now note that if ever $head(t_i) \in \texttt{TFO}$ then for all $j > i$ also $head(s_j), head(t_j) \in \texttt{TFO}$:

Lemma 4. *If* $head(t_i) \in \texttt{TFO}$ *then also* $head(s_{i+1}), head(t_{i+1}) \in \texttt{TFO}$.

Proof. Write $t_i = f\ u_1 \cdots u_n$ with $f \in \texttt{TFO}$. As all rules of the form $f\ l_1 \cdots l_m \to r$ have $n = m$, a $\to_{\mathcal{R},in}$-step on t_i reduces one of the u_j. Thus s_{i+1} has the same head symbol and its immediate subterms are terminating (as they are reducts from the immediate subterms of t_i). Let $s_{i+1} = l\gamma$ with $l \to r \in \mathcal{R}_{\texttt{TFO}}$ and $r\gamma \trianglerighteq t_{i+1}$. Let p be the smallest subterm of r such that $p\gamma \trianglerighteq t_{i+1}$; since r is a truly first order term p is either a variable (which, as assumed, also occurs in l), or has the form $g\ p_1 \cdots p_k$ with $g \in \texttt{TFO}$. In the former case, $s_{i+1} \triangleright \gamma(p) \trianglerighteq t_{i+1}$ is terminating because the strict subterms of s_{i+1} are, contradiction. Thus $g\ p_1 \cdots p_k\gamma \trianglerighteq t_{i+1}$ but (by the choice of p) no $p_j\gamma \trianglerighteq t_{i+1}$; we conclude: $head(t_{i+1}) = g \in \texttt{TFO}$. □

Corollary 5. *If there is an infinite chain, there is one using either only* TFO*-rules, or only* PHO*-rules, for the topmost steps. In the first case, all* t_i *have base type as well (since* $f\ s_1 \cdots s_m : \sigma \Rightarrow \tau$ *does not top-reduce if* $f \in \texttt{TFO}$*).*

4 Simplifying the First Order Part

Using some dependency pair approach, we could now investigate the two possible forms of chains separately. But does this help us significantly? A priori we cannot use first order results to prove non-existence of (minimal infinite) TFO-chains, since even in TFO-chains a step involving higher order symbols might be done in the $\to^*_{\mathcal{R},in}$-reduction. However, note that the rules in $\mathcal{R}_{\texttt{TFO}}$ do not match on the PHO-symbols and that, by minimality, any higher order subterm can be assumed to be terminating. Therefore, as we will see, such subterms are mostly harmless.

Splitting with Orthogonal Rules. Orthogonality is a common property in term rewriting with many nice consequences, including confluence. In first order

orthogonal systems, termination using an innermost rewriting strategy (which is often easier to prove) implies general termination [16].

Orthogonality is not defined for all higher order formalisms; however, where defined it implies confluence, and coincides with the first order definition on first order rules (for an overview of such results, see [34, Section 11.6.2]).

We actually use slightly less than orthogonality: we will show that, if $\mathcal{R}$ has unique normal forms and $\mathcal{R}_{\texttt{TFO}}$ is overlay, then the potentially higher order rules can be omitted when studying TFO-chains. It is not in general decidable whether a system has unique normal forms, but orthogonality of $\mathcal{R}$, which implies both unique normal forms and $\mathcal{R}_{\texttt{TFO}}$ being overlay, is easy to check automatically.

Roughly, the idea is as follows: by unicity of normal forms and the overlay property, the subterms of all s_i in a minimal infinite TFO-chain can be assumed to be normalised. As topmost TFO-steps cannot create PHO-redexes, higher order subterms anywhere in the chain are normalised, and can be replaced by variables.

We say $\mathcal{R}_{\texttt{TFO}}$ is *overlay* if for all $l \to r$, $u \to v \in \mathcal{R}_{\texttt{TFO}}$, substitutions γ, δ and non-empty contexts C: if $l = C[l']$ with $l'\gamma = u\delta$, then l' is a variable.

In Lemmas 6–8 we will assume that all terminating terms s have a unique normal form, and that $\mathcal{R}_{\texttt{TFO}}$ is overlay. Let $\nu(s)$ denote the normal form $s{\downarrow_{\mathcal{R}}}$ of s and, if $s = f\ s_1 \cdots s_n$, then $\nu'(s) = f\ \nu(s_1) \cdots \nu(s_n)$.

Lemma 6 (TFO-steps cannot create PHO-redexes). *If all higher order subterms of s are $\mathcal{R}$-normalised – that is, if, when $s \trianglerighteq q$ either $q = f\ q_1 \cdots q_n$ with $f \in \texttt{TFO}$, or q is in $\mathcal{R}$-normal form – then the same holds for the reducts of s.*

Proof. Suppose s has this property and $s \to_{\mathcal{R}} t$; we use induction on the size of s. Since s is not in normal form, $s = f\ s_1 \cdots s_n$ with $f \in \texttt{TFO}$. If $s \to_{\mathcal{R},top} t$, therefore, $s = l\gamma$, $t = r\gamma$ with $l \to r \in \mathcal{R}_{\texttt{TFO}}$; since r contains no higher order symbols, and higher order subterms of any $\gamma(x)$ are normalised, the property holds for $r\gamma$. Otherwise $t = f\ s_1 \cdots s_i' \cdots s_n$ with $s_i \to_{\mathcal{R}} s_i'$; by the induction hypothesis all higher order subterms of s_i' are $\mathcal{R}$-normalised, and by assumption the same holds for the other s_j. □

Lemma 7 (Normalising Chains). *If there exists a minimal infinite chain $s_1 \to_{\mathcal{R}_{\texttt{TFO}},top} \cdot \trianglerighteq t_1 \to^*_{\mathcal{R},in} s_2 \to_{\mathcal{R}_{\texttt{TFO}},top} \cdot \trianglerighteq t_2 \to^*_{\mathcal{R},in} \dots$ there exists also a minimal infinite chain $\nu'(s_1) \to_{\mathcal{R}_{\texttt{TFO}},top} \cdot \trianglerighteq q_1 \to^*_{\mathcal{R}_{\texttt{TFO}},in} \nu'(s_2) \to_{\mathcal{R}_{\texttt{TFO}},top} \cdot \trianglerighteq q_2 \to^*_{\mathcal{R}_{\texttt{TFO}},in} \cdots$*

Proof. For given i, let $l \to r \in \mathcal{R}_{\texttt{TFO}}$, a subterm p of r and a substitution γ be such that $s_i = l\gamma$ and $t_i = p\gamma$; let $\gamma^{\downarrow}$ be the substitution mapping x to $\gamma(x){\downarrow_{\mathcal{R}}}$ for x in the domain of γ and write $l = f\ l_1 \cdots l_n$. Since $\mathcal{R}_{\texttt{TFO}}$ is overlay, $l'\gamma^{\downarrow}$ cannot be an instance of the left-hand side of a rule for any strict subterm l' of l, so each $l_j\gamma{\downarrow_{\mathcal{R}}}$ is exactly $l_j\gamma^{\downarrow}$. Let $q_i = p\gamma^{\downarrow}$; then $\nu'(s_i) = l\gamma^{\downarrow} \to_{\mathcal{R}_{\texttt{TFO}},top} \cdot \trianglerighteq q_i$.

We can write $p = g\ p_1 \cdots p_m$, $s_{i+1} = g\ v_1 \cdots v_m$ and $q_i = g\ u_1 \cdots u_m$, where each $u_j = p_j\gamma^{\downarrow}$; since $p_j\gamma \to^*_{\mathcal{R}} v_j$, we have $u_j{\downarrow_{\mathcal{R}}} = v_j{\downarrow_{\mathcal{R}}}$. Noting that all higher order subterms of q_i are $\mathcal{R}$-normalised, Lemma 6 gives us that $u_j{\downarrow_{\mathcal{R}_{\texttt{TFO}}}} = u_j{\downarrow_{\mathcal{R}}}$. Thus, $q_i \to^*_{\mathcal{R}_{\texttt{TFO}},in} g\ \nu(u_1) \cdots \nu(u_m) = \nu'(t_i) = \nu'(s_{i+1})$ as required. □

Finally, to get rid of (normalised!) higher order subterms, introduce a variable $\bot_\iota$ for all base types ι. For base-type term s, define $\texttt{rep}(s) = f\ \texttt{rep}(s_1) \cdots \texttt{rep}(s_n)$ if $s = f\ s_1 \cdots s_n$ with $f \in \texttt{TFO}$; otherwise $\texttt{rep}(s) = \bot_\iota$. It follows easily that:

Lemma 8 (Replacing higher order terms by variables). *If all higher order subterms of s are $\mathcal{R}$-normalised, and $s \to_{\mathcal{R}_{\mathtt{TFO}}} t$, then $\mathtt{rep}(s) \to_{\mathcal{R}_{\mathtt{TFO}}} \mathtt{rep}(t)$.*

Proof. With induction on p it is evident that, for base-type terms p, always $\mathtt{rep}(p\gamma) = p\gamma^{\mathtt{rep}}$, where $\gamma^{\mathtt{rep}}(x) = \mathtt{rep}(\gamma(x))$. Using induction on the position of the redex in s, this provides the base case ($s \to_{\mathcal{R}_{\mathtt{TFO}},top} t$); the induction case, $s = f\ s_1 \cdots s_n \to_{\mathcal{R}_{\mathtt{TFO}},in} f\ s_1 \cdots s_i' \cdots s_n = t$, holds by induction hypothesis. □

We now have all the preparations to see that if there is an infinite chain with all head symbols in TFO, there is one on first order terms and with first order rules.

Theorem 9. *Let $(\mathcal{F}, \mathcal{R})$ be a higher order rewrite system with unique normal forms and let $\mathcal{R}_{\mathtt{TFO}}$ be overlay. Then $\to_{\mathcal{R}}$ is terminating if and only if:*

- *there is no minimal infinite chain using only* PHO*-rules in the $\to_{\mathcal{R},top}$-steps, and*
- *$\mathcal{R}_{\mathtt{TFO}}$ is terminating on truly first order terms*

Proof. Suppose $(\mathcal{F}, \mathcal{R})$ is terminating. Then $\mathcal{R}_{\mathtt{TFO}}$ is also terminating (since $\mathcal{R}_{\mathtt{TFO}} \subseteq \mathcal{R}$), and there is no minimal infinite chain at all (since termination of $\to_{\mathcal{R}}$ implies termination of $\to_{\mathcal{R}} \cup \rhd$), let alone using only PHO-rules.

Suppose both properties hold; by Corollary 5, $\to_{\mathcal{R}}$ is terminating if in addition there is no minimal infinite chain using only TFO-rules in the $\to_{top}$-steps. Towards a contradiction, suppose that such a chain exists. By Lemma 7 there is a chain which uses only TFO-rules, $\nu'(s_1) \to_{\mathcal{R}_{\mathtt{TFO}},top} \cdot \trianglerighteq q_1 \to^*_{\mathcal{R}_{\mathtt{TFO}},in} \nu'(s_2) \to_{\mathcal{R}_{\mathtt{TFO}},top} \cdot \trianglerighteq \ldots$; by Lemma 6 (strict subterms of $\nu'(s_1)$ are normalised) higher order subterms are normalised in all terms in the chain. Therefore, by Lemma 8, $\mathtt{rep}(\nu'(s_1)) \to_{\mathcal{R}_{\mathtt{TFO}},top} \cdot \trianglerighteq \mathtt{rep}(q_1) \to^*_{\mathcal{R}_{\mathtt{TFO}},in} \mathtt{rep}(\nu'(s_2)) \to_{\mathcal{R}_{\mathtt{TFO}},top} \cdot \trianglerighteq \ldots$ is an infinite $\mathcal{R}_{\mathtt{TFO}}$-chain on truly first order terms, contradicting termination of $\mathcal{R}_{\mathtt{TFO}}$. □

Thus, given an orthogonal system (or at least, a system where $\mathcal{R}_{\mathtt{TFO}}$ is overlay, and some property guarantees unicity of normal forms), we can split the termination proof into two parts: first, some dependency pair approach, where the dependency pairs for the first order rules can be omitted, and second, proving $\mathcal{R}_{\mathtt{TFO}}$ terminating on truly first order terms.

For the latter part, note that only base-type terms top-reduce, and a base-type, truly first order term corresponds exactly to a purely functional term: we define $\mathtt{uncurry}(f\ s_1 \cdots s_n) = f(\mathtt{uncurry}(s_1), \ldots, \mathtt{uncurry}(s_n))$. The system is terminating if and only if its uncurried version (a many-sorted TRS) is terminating (using [25, Theorem 5], or with a straightforward induction to show that $\mathtt{uncurry}(s) \to_{\mathcal{R}_{\mathtt{TFO}}^{\mathtt{uncurry}}} \mathtt{uncurry}(t)$ if and only if $s \to_{\mathcal{R}_{\mathtt{TFO}}} t$).

Since $\mathcal{R}_{\mathtt{TFO}}$ is a first order overlay TRS with unique normal forms, it is terminating if it is innermost terminating: by [16] this holds for a locally confluent overlay TRS, and by e.g. [34] an innermost terminating (so weakly normalising) TRS with unique normal forms is confluent. Since [10] shows that innermost termination is persistent (a many-sorted TRS is innermost terminating if and only if it is innermost terminating without regarding types), we can send the resulting TRS to any first order termination prover without losing generality, whether or not this prover is type-conscious.

Example 10. $\mathcal{R}_{\texttt{list}}$ is terminating iff the following TRS is terminating:

$$\begin{aligned}
\texttt{append}(\texttt{nil}, l) &\to l\\
\texttt{append}(\texttt{cons}(h,t), l) &\to \texttt{cons}(h, \texttt{append}(t,l))\\
\texttt{reverse}(\texttt{nil}) &\to \texttt{nil}\\
\texttt{reverse}(\texttt{cons}(h,t)) &\to \texttt{append}(\texttt{reverse}(t), \texttt{cons}(h, \texttt{nil}))\\
\texttt{shuffle}(\texttt{nil}) &\to \texttt{nil}\\
\texttt{shuffle}(\texttt{cons}(h,t)) &\to \texttt{cons}(h, \texttt{shuffle}(\texttt{reverse}(t)))\\
\texttt{mirror}(\texttt{nil}) &\to \texttt{nil}\\
\texttt{mirror}(\texttt{cons}(h,t)) &\to \texttt{append}(\texttt{cons}(h, \texttt{mirror}(t)), \texttt{cons}(h, \texttt{nil}))
\end{aligned}$$

and there are no infinite chains using for top-steps only the two `map`-rules.

Termination of $\mathcal{R}_{\texttt{TFO}}$ cannot be demonstrated with HORPO, even combined with dependency pairs and argument filterings (since the first order recursive path orderings with these techniques cannot handle it). However, a first order approach using e.g. dependency pairs and a polynomial interpretation to the natural numbers has no trouble with the resulting first order rules.

As for the higher order part, using the static dependency pair approach from [32] there is one dependency pair $\texttt{map}^\sharp\ (\lambda x.F(x))\ (\texttt{cons}\ x\ y) \to \texttt{map}^\sharp\ (\lambda x.F(x))\ y$ with an empty set of usable rules; HORPO easily solves this.

Splitting the Rules in a Finitely Branching System. The requirements for Theorem 9 are essential; consider for example the following system, where the higher order part lacks the "unique normal forms" property:

$$\begin{aligned}
\texttt{f}\ x\ \texttt{b} &\to \texttt{g}\ x\ x & \qquad \texttt{h}\ (\lambda x.F(x)) &\to F(\texttt{a})\\
\texttt{g}\ x\ \texttt{a} &\to \texttt{f}\ x\ x & \qquad \texttt{h}\ (\lambda x.F(x)) &\to F(\texttt{b})
\end{aligned}$$

Although $\mathcal{R}_{\texttt{TFO}}$ (which consists of the two rules on the left) is terminating and orthogonal, there is an infinite chain with all top-steps in $\mathcal{R}_{\texttt{TFO}}$:

$$\begin{aligned}
\texttt{f}\ (\texttt{h}\ (\lambda x.x))\ \texttt{b} &\to \texttt{g}\ (\texttt{h}\ (\lambda x.x))\ (\texttt{h}\ (\lambda x.x)) \to \texttt{g}\ (\texttt{h}\ (\lambda x.x))\ \texttt{a}\\
&\to \texttt{f}\ (\texttt{h}\ (\lambda x.x))\ (\texttt{h}\ (\lambda x.x)) \to \texttt{f}\ (\texttt{h}\ (\lambda x.x))\ \texttt{b}
\end{aligned}$$

This happens because the first order part is duplicating, and $\texttt{h}\ (\lambda x.x)\ \texttt{a}$ reduces both to $\texttt{a}$ and to $\texttt{b}$ (the F in the corresponding rules is a meta-variable, so a β-step is implied). Note that the role of the higher order part could be taken over by a pair of first order rules, $\texttt{c}(x,y) \to x, \texttt{c}(x,y) \to y$: $\mathcal{R}_{\texttt{TFO}}$ is not $\mathcal{C}_\varepsilon$*-terminating.* Following a technique originally due to Gramlich [15], and occurring in definitions for *usable rules* for full termination [14,18,32], we will see that absence of minimal infinite chains for $\mathcal{R}_{\texttt{TFO}}$ holds if $\mathcal{R}_{\texttt{TFO}}$ (seen as a first order TRS) is $\mathcal{C}_\varepsilon$-terminating.

Roughly, the idea is thus: in a finitely branching system, any term s which is not headed by a symbol in `TFO` can be replaced by the list $s' := \texttt{c}\ t_1\ (\texttt{c}\ t_2 \dots\ (\texttt{c}\ t_n \bot))$ of its immediate reducts; by the two `c`-rules, s' still reduces to all reducts of s. Doing this replacement everywhere in a term does not affect the applicability of first order rules. Thus, in a term $f\ s_1 \cdots s_n$ where all s_i are terminating, the transformation can be repeated until only first order symbols, `c` and $\bot$ remain.

In the following definitions and Lemma 11, let $\mathcal{R}$ be finitely branching.

For all base types ι, let $\bot_\iota$ be a variable of type ι and let $\mathsf{c}_\iota : \iota \Rightarrow \iota \Rightarrow \iota$ be a new function symbol. Let $\mathcal{R}^C_{\mathtt{TFO}} := \mathcal{R}_{\mathtt{TFO}} \cup \{\mathsf{c}_\iota\ x\ y \to x,\ \mathsf{c}_\iota\ x\ y \to y \mid \iota \in \mathcal{B}\}$. Now, for terminating base-type terms s we define $\psi(s)$ and $A(s)$ with a shared induction on $\to_{\mathcal{R}} \cup \rhd$, as follows:

- $\psi(f\ s_1 \cdots s_n) = f\ \psi(s_1) \cdots \psi(s_n)$ if $f \in \mathtt{TFO}$; $\psi(s) = A(s)$ for other s
- $A(s) = D_\iota(\{t \mid s \to_{\mathcal{R}} t\})$ (if $s : \iota$), where D_ι is a function on finite sets of terminating terms, defined by: $D_\iota(X) = \bot_\iota$ if $X = \emptyset$, and $\mathsf{c}_\iota\ \psi(t)\ D_\iota(X \setminus \{t\})$ if X is nonempty and t is its smallest element (ordered lexicographically).

Note that $\{t \mid s \to_{\mathcal{R}} t\}$ is finite by the assumption that $\mathcal{R}$ is finitely branching.

Lemma 11. *If $s \to_{\mathcal{R}} t$ with s a terminating base-type term, then $\psi(s) \to^*_{\mathcal{R}^C_{\mathtt{TFO}}} \psi(t)$.*

Proof. First note that:

1. for truly first order terms q and substitutions γ whose domain includes $FV(q)$, if $q\gamma$ is terminating then $\psi(q\gamma) = q\gamma^\psi$, where $\gamma^\psi(x) = \psi(\gamma(x))$ for x in the domain of γ. This follows immediately with induction on q.
2. $D_\iota(X) \to^*_{\mathcal{R}^C_{\mathtt{TFO}}} \psi(q)$ for any $q \in X$, by a straightforward induction on the size of X. Therefore $A(s) \to^*_{\mathcal{R}^C_{\mathtt{TFO}}} \psi(t)$ if $s \to_{\mathcal{R}} t$.

We prove Lemma 11 by induction on the size of s. If $head(s) \notin \mathtt{TFO}$, then by (2), $\psi(s) = A(s) \to^*_{\mathcal{R}^C_{\mathtt{TFO}}} \psi(t)$. Otherwise, let $s = f\ s_1 \cdots s_n$ with $f \in \mathtt{TFO}$; all s_i have base type, so if a step is done in one of the s_i, we can apply the induction hypothesis. If $s \to_{\mathcal{R},top} t$ then $s = l\gamma$, $t = r\gamma$ for some $l \to r \in \mathcal{R}_{\mathtt{TFO}}$ and substitution γ. Using (1): $\psi(s) = \psi(l\gamma) = l\gamma^\psi \to_{\mathcal{R}^C_{\mathtt{TFO}}} r\gamma^\psi = \psi(r\gamma) = \psi(t)$. □

Theorem 12. *A finitely branching higher order term rewrite system $(\mathcal{F}, \mathcal{R})$ is terminating if:*

- *there is no infinite chain using only* PHO*-rules in the $\to_{\mathcal{R},top}$-steps, and*
- *$\mathcal{R}^C_{\mathtt{TFO}}$ is terminating on truly first order terms*

Proof. By Corollary 5, it suffices if termination of $\mathcal{R}^C_{\mathtt{TFO}}$ implies that there is no minimal infinite chain using only TFO-rules in the $\to_{\mathcal{R},top}$-steps. So suppose there is such a chain $s_1 \to_{\mathcal{R}_{\mathtt{TFO}},top} \cdot \trianglerighteq t_1 \to^*_{\mathcal{R},in} s_2 \ldots$ We must see that $\mathcal{R}^C_{\mathtt{TFO}}$ is non-terminating or, equivalently, that there is an infinite $\to_{\mathcal{R}^C_{\mathtt{TFO}}} \cdot \trianglerighteq$-reduction, on truly first order terms. For a term $u = f\ u_1 \cdots u_n$ with all u_j terminating, let $\psi'(u) = f\ \psi(u_1) \cdots \psi(u_n)$. Then each $\psi'(t_i) \to^*_{\mathcal{R}^C_{\mathtt{TFO}}} \psi'(s_{i+1})$ by Lemma 11, and $\psi'(s_i) = \psi'(l_i\gamma_i) = l_i\gamma_i^\psi \to_{\mathcal{R}_{\mathtt{TFO}},top} \cdot \trianglerighteq p_i\gamma_i^\psi = \psi'(t_i)$ by Observation (1) from its proof. Thus, $\psi'(s_1) \to_{\mathcal{R}_{\mathtt{TFO}},top} \cdot \trianglerighteq \psi'(t_1) \to^*_{\mathcal{R}^C_{\mathtt{TFO}},in} \ldots$ gives the required infinite reduction. □

Note that, unlike Theorem 9, Theorem 12 is not an equivalence. Even if $\mathcal{R}$ is terminating, $\mathcal{R}^C_{\mathtt{TFO}}$ may not be. Consequently, if proving termination of $\mathcal{R}_{\mathtt{TFO}}$ fails, a (sufficiently advanced) higher order approach might still succeed.

As before, we can uncurry the resulting system to obtain a many-sorted TRS. This time, however, dropping types may result in losing termination.

Example 13. Consider the system with six function symbols,

$$\begin{array}{llll} \mathtt{0} : \mathtt{nat} & \mathtt{avg} : \mathtt{nat} \Rightarrow \mathtt{nat} \Rightarrow \mathtt{nat} & \mathtt{fun} : (\mathtt{nat} \Rightarrow \mathtt{nat}) \Rightarrow \mathtt{nat} \\ \mathtt{s} : \mathtt{nat} \Rightarrow \mathtt{nat} & \mathtt{check} : \mathtt{nat} \Rightarrow \mathtt{nat} & \mathtt{apply} : \mathtt{nat} \Rightarrow \mathtt{nat} \Rightarrow \mathtt{nat} \end{array}$$

and the following rules:

$$\begin{array}{rlrl} \mathtt{avg}\ \mathtt{0}\ \mathtt{0} & \to \mathtt{0} & \mathtt{avg}\ x\ (\mathtt{s}\ (\mathtt{s}\ (\mathtt{s}\ y))) & \to \mathtt{s}\ (\mathtt{avg}\ (\mathtt{s}\ x)\ y) \\ \mathtt{avg}\ \mathtt{0}\ (\mathtt{s}\ \mathtt{0}) & \to \mathtt{0} & & \\ \mathtt{avg}\ \mathtt{0}\ (\mathtt{s}\ \mathtt{0}) & \to \mathtt{s}\ \mathtt{0} & \mathtt{apply}\ (\mathtt{fun}\ (\lambda x.F(x)))\ y & \to F(\mathtt{check}\ y) \\ \mathtt{avg}\ \mathtt{0}\ (\mathtt{s}\ (\mathtt{s}\ \mathtt{0})) & \to \mathtt{s}\ \mathtt{0} & \mathtt{check}\ \mathtt{0} & \to \mathtt{0} \\ \mathtt{avg}\ (\mathtt{s}\ x)\ y & \to \mathtt{avg}\ x\ (\mathtt{s}\ y) & \mathtt{check}\ (\mathtt{s}\ x) & \to \mathtt{s}\ (\mathtt{check}\ x) \end{array}$$

The symbol `fun` signifies an encoding of a function in the natural numbers, and `apply` decodes it. To avoid losing termination, the `apply` function employs a check that the function is applied only on a constructor ground term.

This system does not satisfy the requirements from [20], nor can the static framework from [32] be applied. However, we *can* use the dynamic approach from [26]. Thus, by Theorem 12 (not Theorem 9, because the first order part does not have unique normal forms), it suffices to show termination of the TRS:

$$\begin{array}{rrl} \mathtt{avg}(\mathtt{0},\mathtt{0}) \to \mathtt{0} & \mathtt{avg}(\mathtt{s}(x),y) \to \mathtt{avg}(x,\mathtt{s}(y)) & \mathtt{c}(x,y) \to x \\ \mathtt{avg}(\mathtt{0},\mathtt{s}(\mathtt{0})) \to \mathtt{0} & \mathtt{avg}(x,\mathtt{s}(\mathtt{s}(\mathtt{s}(y)))) \to \mathtt{s}(\mathtt{avg}(\mathtt{s}(x),y)) & \mathtt{c}(x,y) \to y \\ \mathtt{avg}(\mathtt{0},\mathtt{s}(\mathtt{0})) \to \mathtt{s}(\mathtt{0}) & \mathtt{check}(\mathtt{0}) \to \mathtt{0} & \\ \mathtt{avg}(\mathtt{0},\mathtt{s}(\mathtt{s}(\mathtt{0}))) \to \mathtt{s}(\mathtt{0}) & \mathtt{check}(\mathtt{s}(x)) \to \mathtt{s}(\mathtt{check}(x)) & \end{array}$$

And additionally find a higher order reduction pair which satisfies $l \geq r$ for all rules, and moreover $\mathtt{apply}^\sharp\ (\mathtt{fun}\ (\lambda x.F(x)))\ y > F(\mathtt{check}(y))$.

For the first part, all rules are strictly oriented with a polynomial interpretation of $f_{\mathtt{0}} = 1,\ f_{\mathtt{s}}(x) = x + 1,\ f_{\mathtt{avg}}(x,y) = 3x + 2y,\ f_{\mathtt{check}}(x) = 2x,\ f_{\mathtt{c}}(x,y) = x+y+1$. For the latter part, consider an argument filtering $\pi(\mathtt{check}\ x) = \mathtt{check}_\pi$, $\pi(\mathtt{s}\ x) = x,\ \pi(\mathtt{avg}\ x\ y) = \mathtt{avg}_\pi$. It suffices to find a reduction pair such that:

$$\begin{array}{lll} \mathtt{apply}^\sharp\ (\mathtt{fun}\ (\lambda x.F(x)))\ y > F(\mathtt{check}_\pi) & \mathtt{check}_\pi \geq 0 & \mathtt{avg}_\pi \geq \mathtt{avg}_\pi \\ \mathtt{apply}\ (\mathtt{fun}\ (\lambda x.F(x)))\ y \geq F(\mathtt{check}_\pi) & \mathtt{check}_\pi \geq \mathtt{check}_\pi & \mathtt{avg}_\pi \geq 0 \end{array}$$

Which is satisfied with HORPO, using a precedence $\mathtt{fun} >_{\mathcal{F}} \mathtt{check}_\pi, \mathtt{avg}_\pi >_{\mathcal{F}} 0$.

Discussion. The restriction to finitely branching systems cannot be dropped, as might be demonstrated with a higher order adaptation of [30, Example 4.6]. However, in practice it is no great problem: a system given by a finite set of rules, even polymorphic rules, is finitely branching in common higher order formalisms.

5 Experiments

We have implemented the contributions of this paper in the higher order termination tool WANDA [24], using a combination of dynamic and static dependency

pairs. WANDA is a participant in the higher order category of the annual International Termination Competition.[1] Here, termination tools compete for power on benchmarks from several categories, with examples from the *Termination Problem Database (TPDB)*. This database is a collection of termination problems from research papers and applications that has been accumulated over the years.[2]

In the competition of 2010, WANDA could prove termination of 7 out of the 12 randomly chosen examples from the TPDB in the category *Higher-Order Rewriting - Union Beta*, coming a close second to THOR, which could handle the same examples plus `Mixed_HO_10/prefixsum.xml` (in the mean time WANDA can also deal with this example). This shows that WANDA is among the state-of-the-art higher order termination provers.

We have coupled WANDA with the first order termination tool AProVE [12] as a black-box to analyse termination of the first order TRSs generated by WANDA. To assess our contributions empirically, we have conducted experiments on an Intel Xeon CPU 5140 with four cores clocked at 2.33 GHz, investigating full termination of in total 152 higher order rewrite systems. As in the termination competition, the proof attempt is aborted after a timeout of 60 seconds.

The Higher Order category in the current TPDB (v8.0) is not very rich in examples (there are only 40 benchmarks). Therefore, we additionally consider higher order termination (union beta) for the 110 (originally untyped) applicative TRSs of the TPDB which could automatically be assigned a simple type.[3] We assume λ-abstraction is allowed in term formation, even though the rules do not use it. Of course, this solves a different problem than the one originally intended; thus these results should not be compared to first order tools analysing the same examples as untyped applicative systems. Additionally, we tested the systems from Examples 1 and 13. We did not include examples from the Haskell category, because WANDA's type system cannot yet deal with the polymorphism present.

	WANDAProVE	WANDA without first order back-end
YES	110	100
NO	10	10
MAYBE	25	38
TIMEOUTS	7	4
Avg. runtime	5.17 *s*	2.90 *s*

Fig. 1. Experimental results of WANDA with and without AProVE as first order prover

Our experiments, which are summarised in Figure 1, show that WANDA combined with AProVE can deal with all examples where plain WANDA succeeds, and 10 more. Out of these 10 additional examples, 8 stem from the applicative benchmarks from the TPDB; the other 2 are the examples used in this paper.

On the benchmarks available in the higher order category of TPDB v8.0, the number of termination proofs is unchanged. This is not surprising since each of

[1] See also `http://termination-portal.org/wiki/Termination_Competition`

[2] For further information we refer to `http://termination-portal.org/wiki/TPDB`

[3] A variation of these examples has by now been accepted for the next TPDB version.

these benchmarks focusses on the higher order aspect, so improvements on the side of the first order aspect can be expected to have only little impact. Runtime increases moderately from an average of 2.90 seconds to 5.17 seconds, which is still far from the timeout of 60 seconds per example, whereas termination proving power increases by 10%.

For details on our experiments and for access to our example suites, we refer to http://aprove.informatik.rwth-aachen.de/eval/WANDAProVE/.

6 Discussion

Overview of the Technique. Using Theorems 9 and 12 we can use a first order termination prover as a "black box" for a higher order tool, as follows:

1. determine TFO and PHO as described in Section 3, as well as $\mathcal{R}_{\mathtt{TFO}}$;
2. if $\mathcal{R}_{\mathtt{TFO}}$ is overlay and $\mathcal{R}$ has unique normal forms, let $\mathcal{R}'$ be the uncurried form of $\mathcal{R}_{\mathtt{TFO}}$; if the system does not satisfy these properties (or we cannot determine whether it does) let $\mathcal{R}'$ be the uncurried form of $\mathcal{R}^{C}_{\mathtt{TFO}}$;
3. feed $\mathcal{R}'$ into a first order termination prover (ignoring the types, unless a prover for many-sorted TRSs is used);
4. if this returns NO and no $\mathtt{c}_\iota$-rules were added to $\mathcal{R}'$, return NO;
   if it returns YES, continue with a dependency pair approach which omits the dependency pairs headed by symbols in TFO;
   otherwise continue with a direct approach for the complete system.

Note that, if the first order prover fails, this algorithm does not abort, but attempts to prove termination of the first order rules along with the rest. It is arguably not very likely that this will be more successful, but a higher order tool may be able to take steps which a type-oblivious first order tool cannot.

Dependency Pairs. While we have not explicitly used dependency pairs except in the examples, the notion of a minimal infinite chain naturally suggests the use of dependency pairs. Several approaches have been suggested for various forms of higher order rewriting [2,31,32,27]. Theorems 9 and 12 provide a way to remove some (perhaps most!) of the dependency graph components of realistic higher order systems, by delegating these to a first order termination prover.

Contribution. The approach outlined in this paper allows (automatic) termination provers to use first order techniques to deal with first order dependency pairs. If we work on finitely branching HRSs with static dependency pairs, Theorem 12 is a direct result of the usable rules approach in [32], but our result holds on *all* common formalisms for higher order rewriting and *any* kind of dependency pair framework. Moreover, for orthogonal systems the result from Theorem 9 is strictly stronger than the theory obtained from this usable rules approach. Our experiments reveal a notable increase of termination proving power by this successful combination of a higher order termination prover with a first order termination prover as a back-end. Therefore, we expect that it will become essential for successful higher order termination provers to either use first order techniques immediately or enlist an external first order termination prover.

Future Work. It might be possible to extend the use of first order provers further by identifying groups of dependency pairs where the higher order aspect is not actively used (such as a dependency pair $\mathtt{map}^\sharp(\lambda x.F(x), \mathtt{cons}(h,t)) \to \mathtt{map}^\sharp(\lambda x.F(x), t)$); dropping types, and transforming an abstraction into a single variable, such pairs might also be handled with first order techniques.

Acknowledgement. We are very grateful for the constructive remarks of the anonymous referees and Femke van Raamsdonk, which helped improve the paper.

References

1. Aoto, T., Yamada, T.: Dependency pairs for simply typed term rewriting. In: Giesl, J. (ed.) RTA 2005. LNCS, vol. 3467, pp. 120–134. Springer, Heidelberg (2005)
2. Aoto, T., Yamada, T.: Argument filterings and usable rules for simply typed dependency pairs. In: Ghilardi, S., Sebastiani, R. (eds.) FroCoS 2009. LNCS, vol. 5749, pp. 117–132. Springer, Heidelberg (2009)
3. Arts, T., Giesl, J.: Termination of term rewriting using dependency pairs. Theoretical Computer Science 236, 133–178 (2000)
4. Blanqui, F.: Termination and confluence of higher-order rewrite systems. In: Bachmair, L. (ed.) RTA 2000. LNCS, vol. 1833, pp. 47–61. Springer, Heidelberg (2000)
5. Blanqui, F., Jouannaud, J.-P., Rubio, A.: The computability path ordering: The end of a quest. In: Kaminski, M., Martini, S. (eds.) CSL 2008. LNCS, vol. 5213, pp. 1–14. Springer, Heidelberg (2008)
6. Borralleras, C., Rubio, A.: A monotonic higher-order semantic path ordering. In: Nieuwenhuis, R., Voronkov, A. (eds.) LPAR 2001. LNCS (LNAI), vol. 2250, pp. 531–547. Springer, Heidelberg (2001)
7. Codish, M., Giesl, J., Schneider-Kamp, P., Thiemann, R.: SAT solving for termination proofs with recursive path orders and dependency pairs. Journal of Automated Reasoning (to appear, 2011)
8. Endrullis, J., Waldmann, J., Zantema, H.: Matrix interpretations for proving termination of term rewriting. Journal of Automated Reasoning 40(2-3), 195–220 (2008)
9. Fuhs, C., Giesl, J., Middeldorp, A., Schneider-Kamp, P., Thiemann, R., Zankl, H.: SAT solving for termination analysis with polynomial interpretations. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 340–354. Springer, Heidelberg (2007)
10. Fuhs, C., Giesl, J., Parting, M., Schneider-Kamp, P., Swiderski, S.: Proving termination by dependency pairs and inductive theorem proving. Journal of Automated Reasoning 47(2), 133–160 (2011)
11. Giesl, J., Raffelsieper, M., Schneider-Kamp, P., Swiderski, S., Thiemann, R.: Automated termination proofs for Haskell by term rewriting. ACM Transactions on Programming Languages and Systems 33 (2011)
12. Giesl, J., Schneider-Kamp, P., Thiemann, R.: AProVE 1.2: automatic termination proofs in the dependency pair framework. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 281–286. Springer, Heidelberg (2006)
13. 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)

14. Giesl, J., Thiemann, R., Schneider-Kamp, P., Falke, S.: Mechanizing and improving dependency pairs. Journal of Automated Reasoning 37(3), 155–203 (2006)
15. Gramlich, B.: Generalized sufficient conditions for modular termination of rewriting. Applicable Algebra in Engineering, Communication and Computing 5, 131–158 (1994)
16. Gramlich, B.: Abstract relations between restricted termination and confluence properties of rewrite systems. Fundamenta Informaticae 24, 3–23 (1995)
17. Hirokawa, N., Middeldorp, A.: Automating the dependency pair method. Information and Computation 199(1,2), 172–199 (2005)
18. Hirokawa, N., Middeldorp, A.: Tyrolean termination tool: Techniques and features. Information and Computation 205(4), 474–511 (2007)
19. Hirokawa, N., Middeldorp, A., Zankl, H.: Uncurrying for termination. In: Cervesato, I., Veith, H., Voronkov, A. (eds.) LPAR 2008. LNCS (LNAI), vol. 5330, pp. 667–681. Springer, Heidelberg (2008)
20. Jouannaud, J.-P., Okada, M.: Abstract data type systems. Theoretical Computer Science 173(2), 349–391 (1997)
21. Jouannaud, J.-P., Rubio, A.: The higher-order recursive path ordering. In: Proc. LICS 1999, pp. 402–411 (1999)
22. Kennaway, R., Klop, J.W., Sleep, M.R., de Vries, F.-J.: Comparing curried and uncurried rewriting. Journal of Symbolic Computation 21(1), 15–39 (1996)
23. Klop, J.W., van Oostrom, V., van Raamsdonk, F.: Combinatory reduction systems: introduction and survey. Theoretical Computer Science 121(1-2), 279–308 (1993)
24. Kop, C.: WANDA – a higher order termination tool, http://www.few.vu.nl/~kop/code.html
25. Kop, C.: Simplifying algebraic functional systems. In: Winkler, F. (ed.) CAI 2011. LNCS, vol. 6742, pp. 201–215. Springer, Heidelberg (2011)
26. Kop, C., van Raamsdonk, F.: Higher-order dependency pairs with argument filterings. In: Proc. WST 2010 (2010), http://www.few.vu.nl/~kop/wst10.pdf
27. Kop, C., van Raamsdonk, F.: Higher order dependency pairs for algebraic functional systems. In: Proc. RTA 2011. LIPIcs, vol. 10, pp. 203–218. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2011)
28. Kusakari, K., Sakai, M.: Enhancing dependency pair method using strong computability in simply-typed term rewriting. Applicable Algebra in Engineering, Communication and Computing 18(5), 407–431 (2007)
29. Nipkow, T.: Higher-order critical pairs. In: Proc. LICS 1991, pp. 342–349 (1991)
30. Ohlebusch, E.: On the modularity of termination of term rewriting systems. Theoretical Computer Science 136(2), 333–360 (1994)
31. Sakai, M., Watanabe, Y., Sakabe, T.: An extension of the dependency pair method for proving termination of higher-order rewrite systems. IEICE Transactions on Information and Systems E84-D(8), 1025–1032 (2001)
32. Suzuki, S., Kusakari, K., Blanqui, F.: Argument filterings and usable rules in higher-order rewrite systems. IPSJ Transactions on Programming 4(2), 1–12 (2011)
33. Tannen, V., Gallier, G.H.: Polymorphic rewriting conserves algebraic strong normalization. Theoretical Computer Science 83(1), 3–28 (1991)
34. Terese: Term Rewriting Systems. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge (2003)
35. Yamada, T.: Confluence and termination of simply typed term rewriting systems. In: Middeldorp, A. (ed.) RTA 2001. LNCS, vol. 2051, pp. 338–352. Springer, Heidelberg (2001)
36. Zantema, H.: Termination of term rewriting by semantic labelling. Fundamenta Informaticae 24, 89–105 (1995)

Stochastic Local Search for SMT: Combining Theory Solvers with WalkSAT*

Alberto Griggio[1], Quoc-Sang Phan[2], Roberto Sebastiani[2], and Silvia Tomasi[2]

[1] FBK-Irst, Trento, Italy
[2] DISI, University of Trento, Italy

Abstract. A dominant approach to Satisfiability Modulo Theories (SMT) relies on the integration of a Conflict-Driven-Clause-Learning (CDCL) SAT solver and of a decision procedure able to handle sets of atomic constraints in the underlying theory $\mathcal{T}$ ($\mathcal{T}$*-solver*). In pure SAT, however, Stochastic Local-Search (SLS) procedures sometimes are competitive with CDCL SAT solvers on satisfiable instances. Thus, it is a natural research question to wonder whether SLS can be exploited successfully also inside SMT tools.

In this paper we investigate this issue. We first introduce a general procedure for integrating a SLS solver of the WalkSAT family with a $\mathcal{T}$*-solver*. Then we present a group of techniques aimed at improving the synergy between these two components. Finally we implement all these techniques into a novel SLS-based SMT solver for the theory of linear arithmetic over the rationals, combining UBCSAT/UBCSAT++ and MathSAT, and perform an empirical evaluation on satisfiable instances. The results confirm the potential of the approach.

1 Introduction

Satisfiability Modulo Theories (SMT) is the problem of deciding the satisfiability of a (typically quantifier-free) first-order formula with respect to some decidable theory $\mathcal{T}$. A dominant approach to SMT, called *lazy approach*, relies on the integration of a Conflict-Driven Clause-Learning (CDCL) SAT solver and of a decision procedure able to handle sets of atomic constraints in the underlying theory $\mathcal{T}$ ($\mathcal{T}$*-solver*) (see, e.g., [13, 5]). In pure SAT, however, Stochastic Local-Search (SLS) procedures (see [11]) sometimes are competitive with or even outperform CDCL SAT solvers on satisfiable instances, in particular when dealing with unstructured problems. Therefore, it is a natural research question to wonder whether SLS can be exploited successfully also inside SMT tools. In this paper we start investigating this issue.

Remarkably, CDCL and SLS SAT solvers are very different in the way they perform search. CDCL SAT solvers reason on *partial* truth assignments, which are updated in a stack-based manner. Moreover, they intensively use techniques like *boolean constraint-propagation (BCP)*, *conflict-directed backtracking (backjumping)* and *learning*, which

* A. Griggio is supported by Provincia Autonoma di Trento and the European Community's FP7/2007-2013 under grant agreement Marie Curie FP7 - PCOFUND-GA-2008-226070 "progetto Trentino", project ADAPTATION. R. Sebastiani is supported in part by SRC/GRC under Custom Research Project 2009-TJ-1880 WOLFLING. We wish to thank H. Hoos, D. Tompkins, A. Belov and Z. Stachniak for their help with their tools and for useful insights.

C. Tinelli and V. Sofronie-Stokkermans (Eds.): FroCoS 2011, LNAI 6989, pp. 163–178, 2011.

are heavily exploited in the lazy-SMT paradigm and allow for very-efficient SMT optimization techniques like *early pruning*, *theory-propagation*, *theory-driven backjumping and learning* (see [13,5]). SLS SAT solvers, instead, reason on *total* truth assignments, which are updated by swapping the phase of single literals according to some mixed greedy/stochastic strategy. Moreover, they typically do not use BCP, backjumping and learning. Therefore, the problem of an effective integration of a $\mathcal{T}$-solver with a SLS SAT solver is not a straightforward variant of the standard integration with a CDCL solver in lazy SMT. Moreover, the standard SMT optimization techniques mentioned above cannot be applied in a straightforward way.

In order to cope with these problems, we perform the following steps. First, inspired by the idea of "partially-invisible" SAT formulas, we present a novel and general architecture for integrating a $\mathcal{T}$*-solver* with a Boolean SLS solver based on the widely-used WalkSAT algorithm, resulting in a basic SLS-based SMT solver, which we call WALKSMT. Second, we analyze the differences between the interaction of a $\mathcal{T}$-solver with a CDCL-based and a SLS-based SAT solver, and we introduce and discuss a group of optimization techniques aimed at improving the synergy between an SLS solver and the $\mathcal{T}$-solver. Third, we present an implementation of WALKSMT with the optimization techniques above, which is based on the integration of the UBCSAT [17] and UBCSAT++ [6] SLS solvers with the $\mathcal{LA}(\mathbb{Q})$-solver of MATHSAT [7]. Finally, we perform an extensive experimental evaluation of our implementation. We consider satisfiable industrial problems coming from the SMT-LIB, and we evaluate the effects of the various optimization techniques, also comparing them against MATHSAT. We observe that (i) the basic "naive" version of WALKSMT was not able to solve any problem within a 600s timeout; (ii) the optimization techniques drastically improve the performances of the basic version, allowing the optimized WALKSMT to solve 149/225 problems; (iii) as a comparison, MATHSAT solved 208/225 problems. We also compare the optimized WALKSMT and MATHSAT on randomly-generated unstructured problems, obtaining small differences in performances.

The rest of the paper is organized as follows. In §2 we introduce the necessary background on SLS and SMT. In §3 and §4 we describe respectively our basic algorithm and the optimization techniques we have conceived for improving its performance. In §5 we experimentally evaluate our approach. In §6 we conclude and highlight directions for future work.

2 Background

2.1 Stochastic Local Search for SAT

Local search (LS) algorithms [11,10] are widely used for solving hard combinatorial search problems. The idea behind LS is to inspect the search space of a given problem instance starting at some position and then iteratively moving from the current position to a neighboring one where each move is determined by a decision based on information about the local neighborhood. LS algorithms making use of randomized choices during the search process are called *Stochastic Local search (SLS) algorithms*. SLS algorithms have been successfully applied to the solution of many NP-complete decision problems, including SAT. Notice, however, that SLS algorithms typically do not guarantee that

Algorithm 1. WalkSAT (φ)

Require: CNF formula φ, MAX_TRIES, MAX_FLIPS
1: **for** $i = 1$ to MAX_TRIES **do**
2: $\mu \leftarrow$ INITIALTRUTHASSIGNMENT(φ)
3: **for** $j = 1$ to MAX_FLIPS **do**
4: **if** ($\mu \models \varphi$) **then**
5: **return** SAT
6: **else**
7: $c \leftarrow$ CHOOSEUNSATISFIEDCLAUSE(φ)
8: $\mu \leftarrow$ NEXTTRUTHASSIGNMENT(φ, c)
9: **end if**
10: **end for**
11: **end for**
12: **return** UNKNOWN

eventually an existing solution is found, so that they cannot verify the unsatisfiability of a problem.

SLS algorithms for SAT typically work with a CNF input formula (namely φ) and share a common high-level schema: (i) they initialize the search by generating an initial truth assignment (typically at random); (ii) they iteratively select one variable and flip it within the current truth assignment. The search terminates when the current truth assignment satisfies the formula φ or after MAX_TRIES sequences of MAX_FLIPS variable flips without finding a model for φ. The main difference in SLS SAT algorithms is typically given by the different strategies applied to select the variable to be flipped.

WalkSAT Algorithms. WalkSAT is a popular family of SLS-based SAT algorithms [11,10]. The schema of such algorithms is shown in Algorithm 1. Initially, a complete truth assignment μ for the variables of the input problem φ is selected by INITIALTRUTHASSIGNMENT according to some heuristic criterion (e.g., uniformly at random). If this assignment satisfies the formula, then the algorithm terminates. Otherwise, a variable is selected and flipped in μ using a two-stage process. In the first stage, a currently-unsatisfied clause c is selected by CHOOSEUNSATISFIEDCLAUSE according to some heuristic criterion (e.g., uniformly at random). In the second stage, one of the variables occurring in the selected clause c is flipped by NEXTTRUTHASSIGNMENT according to some mixed greedy/random heuristic criterion, so that to generate another truth assignment. The procedure is repeated until either a solution is found, or the limit for the number of tries is reached.

Over the last ten years, several variants of the basic WalkSAT algorithm have been proposed [14,12,16], which differ mainly for the different heuristics used for the functions described above —in particular on the degree of greediness and randomness and in the criteria used for selecting the variable to flip in c within NEXTTRUTHASSIGNMENT. From our own empirical experience [15], the best performing WalkSAT-based algorithm for SAT seems to be Adaptive Novelty$^+$ [16]. It adopts the Novelty$^+$'s variable selection heuristic, and it adjusts its degree of greediness according to the search progress. Novelty$^+$ chooses the variable to be flipped from c depending on the score (i.e. the difference in the total number of satisfied clauses a flip would cause) and the

variable's age (i.e. the number of search steps performed since a variable was last flipped). If the variable with the highest score does not have minimal age among the variables in c, then it is selected. Otherwise, it is selected with a probability $1-p$, where p is a parameter (called *noise setting*). While in the remaining cases p, the variable is picked uniformly at random (*random walk*). Adaptive Novelty$^+$ changes the probability of making greedy choices by increasing the noise setting p only when it needs to escape from situations in which there is no further progress in finding a solution (once the stagnation situation is overcome, the noise setting is gradually decreased). We refer the reader to [11] for a more detailed explanation.

Trimming Variable Selection and Literal Commitment Strategy. A few attempts have been made in order to enhance SLS algorithms with techniques borrowed from CDCL solvers (e.g. [6,4]). In particular, Belov and Stachniak [6] propose two techniques that exploit the search history to improve the variable selection process of the classic SLS procedures for SAT. They modify the WalkSAT schema by adding a database (DB) that represents a set of constraints that help to guide the search process. It consists in (1) a set of clauses ψ obtained by storing selected unsatisfied-clauses (see line 7 of Algorithm 1) and (2) a partial truth assignment η that records assignments made by the local search heuristic. The goal of the *trimming variable selection* technique is to prune the search by preventing the selection of variables whose flip will cause a conflict in the database. In particular, for every variable v belonging to the selected clause c, the procedure checks the satisfiability of $\psi \wedge \eta'$ by unit propagation, where η' is obtained from η by adding the (flipped) truth assignment of v. If it is unsatisfiable, the variable v cannot be flipped. When all variables cause a conflict, the database is reset (i.e. η is set to $\emptyset$) so that any variable can be chosen by the local search heuristic. Notice that, once the truth value of a variable has been flipped, η is updated accordingly and the clause c is added to the database.

The *literal commitment strategy* aims at exploiting the power of unit propagation inside SLS procedures that naturally work with total truth assignments rather than partial ones. It iteratively deduces literals l in ψ deriving from η (i.e. $\psi \wedge \eta \models l$) and updates the current total truth assignment μ accordingly during a single search step. We refer the reader to [6] for a more detailed explanation.

2.2 Satisfiability Modulo Theory

Let $\mathcal{T}$ be a first-order theory. We call $\mathcal{T}$*-literal* a ground atomic formula in $\mathcal{T}$ or its negation. We call a *theory solver for* $\mathcal{T}$, $\mathcal{T}$*-solver*, a tool able to decide the $\mathcal{T}$-satisfiability of a conjunction/set μ of $\mathcal{T}$-literals. If μ is $\mathcal{T}$-unsatisfiable, then $\mathcal{T}$*-solver* returns UNSAT and the subset η of $\mathcal{T}$-literals in μ which was found $\mathcal{T}$-unsatisfiable; (η is hereafter called a $\mathcal{T}$*-conflict set*, and $\neg\eta$ a $\mathcal{T}$*-conflict clause*.) if μ is $\mathcal{T}$-satisfiable, then $\mathcal{T}$*-solver* returns SAT; it may also be able to return some unassigned $\mathcal{T}$-literal $l \notin \mu$[1] s.t. $\{l_1, ..., l_n\} \models_{\mathcal{T}} l$, where $\{l_1, ..., l_n\} \subseteq \mu$. We call this process $\mathcal{T}$*-deduction* and $(\bigvee_{i=1}^{n} \neg l_i \vee l)$ a $\mathcal{T}$*-deduction clause*. Notice that $\mathcal{T}$-conflict and $\mathcal{T}$-deduction clauses

[1] Taken from a set of all the available $\mathcal{T}$-literals; when combined with a SAT solver, such set would be the set of all the $\mathcal{T}$-literals occurring in the input formula to solve.

are valid in $\mathcal{T}$. We call them $\mathcal{T}$*-lemmas*. Given a $\mathcal{T}$-formula φ, the formula φ^p obtained by rewriting each $\mathcal{T}$-atom in φ into a fresh atomic proposition is the Boolean abstraction of φ, and φ is the *refinement* of φ^p. Notationally, we indicate by φ^p and μ^p the Boolean abstraction of φ and μ, and by φ and μ the refinements of φ^p and μ^p respectively. With a little abuse of notation, we say that μ^p is $\mathcal{T}$-(un)satisfiable iff μ is $\mathcal{T}$-(un)satisfiable.

In a lazy SMT($\mathcal{T}$) solver, the *Boolean abstraction* φ^p of the input formula φ is given as input to a CDCL SAT solver, and whenever a satisfying assignment μ^p is found s.t. $\mu^p \models \varphi^p$, the corresponding set of $\mathcal{T}$-literals μ is fed to the $\mathcal{T}$*-solver*; if μ is found $\mathcal{T}$-consistent, then φ is $\mathcal{T}$-consistent; otherwise, $\mathcal{T}$*-solver* returns the $\mathcal{T}$-conflict set η causing the inconsistency, so that the clause $\neg\eta^p$ (the Boolean abstraction of $\neg\eta$) is used to drive the backjumping and learning mechanism of the SAT solver. Important optimizations are *early pruning* and $\mathcal{T}$*-propagation*: the $\mathcal{T}$*-solver* is invoked also on an intermediate assignment μ: if it is $\mathcal{T}$-unsatisfiable, then the procedure can backtrack; if not, and if the $\mathcal{T}$*-solver* is able to perform a $\mathcal{T}$-deduction $\{l_1, ..., l_n\} \models_{\mathcal{T}} l$, then l can be unit-propagated, and the $\mathcal{T}$-deduction clause $(\bigvee_{i=1}^{n} \neg l_i \vee l)$ can be used in backjumping and learning. The above schema is a coarse abstraction of the procedures underlying all the state-of-the-art lazy SMT tools. The interested reader is pointed to, e.g., [13,5] for details and further references.

3 Stochastic Local Search for SMT

We start from a simple observation: in principle, from the perspective of a SAT solver, an SMT problem instance φ can be seen as the problem of solving a *partially-invisible* CNF SAT formula $\varphi^p \wedge \tau^p$, s.t. the "visible" part φ^p is the Boolean abstraction of φ and the "invisible" part τ^p is (the Boolean abstraction of) the set τ of all the $\mathcal{T}$-lemmas providing the obligations induced by the theory $\mathcal{T}$ on the $\mathcal{T}$-atoms of φ. (See the example in Fig 1.) Thus, every assignment μ^p s.t. $\mu^p \models \varphi^p$ is $\mathcal{T}$-unsatisfiable iff μ^p falsifies some non-empty set of clauses $\{c_1^p, ..., c_n^p\} \subseteq \tau^p$. To this extent, a traditional "lazy" SMT solver can be seen as a CDCL SAT solver which knows φ^p but not τ^p: whenever a model μ^p for φ^p is found, it is passed to a $\mathcal{T}$*-solver* which (behaves as if it) knows τ^p, and hence checks if μ^p falsifies some clause $c_i^p \in \tau^p$: if this is the case, it returns one (or more) such clause(s) c_i^p, which is then used to drive the future search and which is optionally added to φ^p.

3.1 A Basic WalkSMT Procedure

The above observation inspired to us a procedure integrating a $\mathcal{T}$*-solver* into a SLS algorithm of the WalkSAT family (WalkSMT hereafter). A high-level description of the pseudo-code of WalkSMT is shown in Algorithm 2. (We present first a basic version of WalkSMT, in which we temporarily ignore steps 1-3 and 12-13, which we will describe in §4, together with other enhancements.) WalkSMT receives in input a SMT($\mathcal{T}$) CNF formula and applies a WalkSAT scheme to its Boolean abstraction φ^p. InitialTruthAssignment, ChooseUnsatisfiedClause and NextTruthAssignment are the functions described in §2.1. (Notice that their underlying heuristics vary with the different variants of WalkSAT adopted.)

ϕ :
c_1 : $\{A_1\}$
c_2 : $\{\neg A_1 \vee (x - z > 4)\}$
c_3 : $\{\neg A_3 \vee A_1 \vee (y \geq 1)\}$
c_4 : $\{\neg A_2 \vee \neg(x - z > 4) \vee \neg A_1\}$
c_5 : $\{(x - y \leq 3) \vee \neg A_4 \vee A_5\}$
c_6 : $\{\neg(y - z \leq 1) \vee (x + y = 1) \vee \neg A_5\}$
c_7 : $\{A_3 \vee \neg(x + y = 0) \vee A_2\}$
c_8 : $\{\neg A_3 \vee (z + y = 2)\}$

ϕ^p :
c_1 : $\{A_1\}$
c_2 : $\{\neg A_1 \vee B_1\}$
c_3 : $\{\neg A_3 \vee A_1 \vee B_2\}$
c_4 : $\{\neg A_2 \vee \neg B_1 \vee \neg A_1\}$
c_5 : $\{B_3 \vee \neg A_4 \vee A_5\}$
c_6 : $\{\neg B_4 \vee B_5 \vee \neg A_5\}$
c_7 : $\{A_3 \vee \neg B_6 \vee A_2\}$
c_8 : $\{\neg A_3 \vee B_7\}$

τ : (all possible $\mathcal{T}$-lemmas on the $\mathcal{T}$-atoms of ϕ)
c_9 : $\{\neg(x + y = 0) \vee \neg(x + y = 1)\}$
c_{10} : $\{\neg(x - z > 4) \vee \neg(x - y \leq 3) \vee \neg(y - z \leq 1)\}$
c_{11} : $\{(x - z > 4) \vee (x - y \leq 3) \vee (y - z \leq 1)\}$
c_{12} : $\{\neg(x - z > 4) \vee \neg(x + y = 1) \vee \neg(z + y = 2)\}$
c_{13} : $\{\neg(x - z > 4) \vee \neg(x + y = 0) \vee \neg(z + y = 2)\}$
... ...

τ^p :
c_9 : $\{\neg B_6 \vee \neg B_5\}$
c_{10} : $\{\neg B_1 \vee \neg B_3 \vee \neg B_4\}$
c_{11} : $\{B_1 \vee B_3 \vee B_4\}$
c_{12} : $\{\neg B_1 \vee \neg B_5 \vee \neg B_7\}$
c_{13} : $\{\neg B_1 \vee \neg B_6 \vee \neg B_7\}$
... ...

$B_1 \stackrel{\text{def}}{=} (x - z > 4)$, $B_2 \stackrel{\text{def}}{=} (y \geq 1)$, $B_3 \stackrel{\text{def}}{=} (x - y \leq 3)$, $B_4 \stackrel{\text{def}}{=} (y - z \leq 1)$,
$B_5 \stackrel{\text{def}}{=} (x + y = 1)$, $B_6 \stackrel{\text{def}}{=} (x + y = 0)$, $B_7 \stackrel{\text{def}}{=} (z + y = 2)$.

φ :
c_2 : $\{(x - z > 4)\}$
c_5 : $\{(x - y \leq 3) \vee \neg A_4 \vee A_5\}$
c_6 : $\{\neg(y - z \leq 1) \vee (x + y = 1) \vee \neg A_5\}$
c_7 : $\{A_3 \vee \neg(x + y = 0)\}$
c_8 : $\{\neg A_3 \vee (z + y = 2)\}$
c_9 : $\{\neg(x + y = 0) \vee \neg(x + y = 1)\}$

φ^p :
c_2 : $\{B_1\}$
c_5 : $\{B_3 \vee \neg A_4 \vee A_5\}$
c_6 : $\{\neg B_4 \vee B_5 \vee \neg A_5\}$
c_7 : $\{A_3 \vee \neg B_6\}$
c_8 : $\{\neg A_3 \vee B_7\}$
c_9 : $\{\neg B_6 \vee \neg B_5\}$

$\mu_1^p = \{B_1, A_3, \neg A_4, \neg A_5, \neg B_6, B_5, B_3, B_4, B_7\}$
$\mu_1 = \{(x - z > 4), \neg(x + y = 0), (x + y = 1), (x - y \leq 3), (y - z \leq 1), (z + y = 2)\}$

Fig. 1. Top: example of an SMT($\mathcal{LA}(\mathbb{Q})$) formula ϕ as a "partially-invisible" formula $\phi^p \wedge \tau^p$. Middle: the formula φ [resp φ^p] obtained from ϕ [resp ϕ^p] after preprocessing (see §4). Bottom: a truth assignment μ^p satisfying φ^p and violating c_{10}, c_{12} in τ^p, and its refinement μ_1.

Since we are temporarily ignoring steps 1-3 and 12-13, the only significant difference wrt. Algorithm 1 is in steps 7-14. Whenever a total model μ^p is found s.t. $\mu^p \models \varphi^p$, it is passed to $\mathcal{T}$-*solver*. If (the set of $\mathcal{T}$-literals corresponding to) μ^p is $\mathcal{T}$-satisfiable (i.e., $\mu^p \models \varphi^p \wedge \tau^p$) the procedures ends returning SAT. Otherwise, $\mathcal{T}$-*solver* returns CONFLICT and a $\mathcal{T}$-lemma c^p. Notice that this corresponds to say that $\mu^p \not\models \varphi^p \wedge \tau^p$, and that c^p is one of the (possibly-many) clauses in $\varphi^p \wedge \tau^p$ which are falsified by μ^p. Thus, c^p is used by NEXTTRUTHASSIGNMENT as "selected" unsatisfied clause to drive the flipping of the variable. To this extent, $\mathcal{T}$-*solver* plays also the role of CHOOSEUNSATISFIEDCLAUSE on $\varphi^p \wedge \tau^p$ when no unsatisfied clause is found in φ^p (to this extent, see also "Multiple Learning" in §4).

Algorithm 2. WALKSMT (φ)

Require: SMT($\mathcal{T}$) CNF formula φ, MAX_TRIES, MAX_FLIPS

```
 1: if (T-PREPROCESS (φ) == CONFLICT) then
 2:     return UNSAT
 3: end if
 4: for i = 1 to MAX_TRIES do
 5:     μ^p ← INITIALTRUTHASSIGNMENT (φ^p)
 6:     for j = 1 to MAX_FLIPS do
 7:         if (μ^p ⊨ φ^p) then
 8:             ⟨status, c^p⟩ ← T-solver(φ^p, μ^p)
 9:             if (status == SAT) then
10:                 return SAT
11:             end if
12:             c^p ← UNIT-SIMPLIFICATION(φ^p, c^p)
13:             φ^p ← φ^p ∧ c^p
14:             μ^p ← NEXTTRUTHASSIGNMENT (φ^p, c^p)
15:         else
16:             c^p ← CHOOSEUNSATISFIEDCLAUSE (φ^p)
17:             μ^p ← NEXTTRUTHASSIGNMENT (φ^p, c^p)
18:         end if
19:     end for
20: end for
21: return UNKNOWN
```

Example 1. Suppose WALKSMT is invoked on the formula φ^p in Fig. 1, generating the total truth assignment μ_1^p that satisfies φ^p. Then $\mathcal{T}$*-solver* is invoked on μ_1, which is $\mathcal{T}$-inconsistent due to the the literals $\{(x - z > 4), (x + y = 1), (z + y = 2)\}$, returning UNSAT and the conflict clause $c_1^p = \{\neg B_1 \vee \neg B_5 \vee \neg B_7\}$ (i.e. c_{12} in τ^p). Then NEXTTRUTHASSIGNMENT will flip one of the literals B_1, B_5 or B_7.

Remark: Efficient $\mathcal{T}$-Solvers for Local Search. In CDCL-Based SMT solvers, the interaction with $\mathcal{T}$*-solvers* is *stack-based*: the truth assignment μ is incrementally extended when performing unit propagation, $\mathcal{T}$-propagation, and when picking an unassigned literal for branching, and it is partly undone upon backtracking, when the most-recently-assigned literals are removed from it. Consequently, $\mathcal{T}$*-solvers* designed for interaction with a CDCL SAT solver are typically optimized for such stack-based invocation. In particular, they are typically *incremental* —when they have to check the consistency of a truth assignment μ' that is an extension of a previously-checked μ, they don't need to restart the computation from scratch— and *backtrackable* —when backtracking occurs, the most-recently-assigned literals that need to be unassigned can be efficiently removed, and the internal state can be efficiently restored to a previous configuration (see [13,5]).

In local search, instead, a new assignment μ' is obtained from the previous one μ by flipping *an arbitrary* literal (according to some heuristics). In this setting, the conventional backtrackability feature of $\mathcal{T}$*-solvers* is of little use, since there is no notion of most-recently-assigned literals to remove. Instead, it is very desirable to be able to

remove *arbitrary* literals from a $\mathcal{T}$*-solver* without the need of resetting its internal state. Such requirement might seem unrealistic, or at least difficult to fulfill. However, at least two state-of-the-art $\mathcal{T}$*-solvers* have this capability: the $\mathcal{T}$*-solver* for $\mathcal{DL}$ of [8] and the $\mathcal{T}$*-solver* for $\mathcal{LA}(\mathbb{Q})$ of [9], which are therefore natural candidates for integration with a SLS-based SAT solver. The MATHSAT solver implements both.

4 Enhancements to the Basic WalkSMT Procedure

The WALKSMT algorithm described above is very naive. Here we analyze the interaction of a $\mathcal{T}$-solver with a SLS SAT solver, and we present a group of optimization techniques aimed at improving the synergy of their interaction.

4.1 Preprocessing

Before entering the main WALKSMT routine, we apply a *preprocessing* step to the input formula φ in order to make it simpler to solve (steps 1-3 in algorithm 19). This preprocessing consists mainly of two techniques: *Initial BCP* and *Static Learning*.

Initial BCP. Often SMT formulas contain lots of "structural" atomic propositions whose truth value is assigned deterministically (e.g., when the formula derives from a CNF-ization step). Unlike a CDCL solver, an SLS one cannot handle them efficiently. Thus, during preprocessing we first perform a run of BCP to the input formula, simplifying the formula accordingly. In order to preserve correctness, we keep as unit clauses the $\mathcal{T}$-literals $l_1, .., l_n$ which have been assigned to true by BCP. If during this process one of the clauses of ϕ^p is falsified, or if the set of $\mathcal{T}$-literals $l_1, .., l_n$ above is $\mathcal{T}$-inconsistent, the algorithm can exit returning UNSAT. Otherwise, $l_1, .., l_n$ are tagged "unflippable", so that the SLS engine initially assigns them to true and never flips their value.

Static Learning. During preprocessing we also conjoin to the formula φ/φ^p short and "obvious" $\mathcal{T}$-lemmas on the atoms occurring in φ, which can be generated without explicitly invoking the $\mathcal{T}$-solver. (Examples of such $\mathcal{T}$-lemmas are mutual-exclusion lemmas like c_9 in Fig. 1. See also [13].) Thus the $\mathcal{T}$*-solver* is invoked on an assignment μ only if μ^p verifies also these $\mathcal{T}$-lemmas (row 7 in Alg. 2). This prevents WALKSMT from invoking $\mathcal{T}$*-solver* on obviously-$\mathcal{T}$-inconsistent assignments.

Example 2. Consider as input the formula ϕ of Fig. 1 (top). The preprocessing step generates the formula φ of Fig. 1 (bottom). In fact, BCP unit-propagates the literals $A_1, B_1, \neg A_2$, simplifying clause c_7 and eliminating clauses c_1, c_3 and c_4. Clause c_2 survives as an unit clause because B_1 is (the label of) a $\mathcal{T}$-literal. Notice that the $\mathcal{T}$-atom $B_2 \stackrel{\text{def}}{=} (y \geq 0)$ disappears from the formula because c_3 is satisfied by the unit-propagation of A_1. The $\mathcal{T}$-lemma c_9 is then added to the simplified formula by static learning.

4.2 Single and Multiple Learning

Learning. SLS SAT solvers typically do not implement learning. This is potentially a major problem with SLS-based SMT, because the SLS solver may generate many total

assignments $\mu_1^p, ..., \mu_k^p$ each containing the same $\mathcal{T}$-inconsistent subset η^p, causing thus $k-1$ useless calls to $\mathcal{T}$-*solver*. Thus, like in standard CDCL-based SMT solvers, we conjoin to φ^p the $\mathcal{T}$-lemma c^p returned by the $\mathcal{T}$-solver (step 13). Henceforth $\mathcal{T}$-*solver* is no more invoked on assignments violating c^p.

Unit Resolution. Before learning a $\mathcal{T}$-lemma c, we remove from it all the $\mathcal{T}$-literals whose negation occurs as unit clauses in the input problem (step 12). (Notice that after this step c may be no longer a $\mathcal{T}$-lemma.) We do this in both static and dynamic learning.

Example 3. Consider the scenario of Example 1, assuming learning is implemented. Because of the unit clause c_2 of φ^p, we remove from the conflict clause c_1^p the literal $\neg B_1$, obtaining $c_1^{p\prime} \stackrel{\text{def}}{=} \{\neg B_5 \vee \neg B_7\}$ (i.e., a unit-resolved version of c_{12} in τ^p.), which we add to φ^p. Then NEXTTRUTHASSIGNMENT will flip one of the literals B_5 or B_7. $\mathcal{T}$-*solver* will never be invoked again on assignments containing both B_5 and B_7.

Multiple Learning. Unlike with CDCL-based SMT solvers, which typically use some form of early pruning to check partial truth assignments for $\mathcal{T}$-consistency, in an SLS-based approach $\mathcal{T}$-solvers operate always on *complete* truth assignments μ. In this setting, it is likely that μ contains many different $\mathcal{T}$-inconsistent subsets, often independent from each another. This is the idea at the basis of our *multiple learning* technique, which allows for learning more than one $\mathcal{T}$-lemma for every $\mathcal{T}$-inconsistent assignment. When a conflict set η is found (and simplified via unit-resolution), a given percentage p of its literals are randomly removed from μ, and $\mathcal{T}$-*solver* is invoked again on the resulting set. This process is repeated until no more conflict is found. We then learn all the $\mathcal{T}$-lemmas $c_1^p, ..., c_k^p$ generated during the process. Also, if $k > 1$, then one clause c^p among $c_1^p, ..., c_k^p$ is chosen by CHOOSEUNSATISFIEDCLAUSE to be fed to NEXTTRUTHASSIGNMENT.

Example 4. Consider the scenario of Example 1 and 3, assuming multiple learning is implemented, with $p = 100\%$. After learning the clause $c_1^{p\prime}$, we drop B_5, B_7 from μ_1^p and re-invoke $\mathcal{T}$-*solver* on the set of $\mathcal{T}$-literals $\mu_2 \stackrel{\text{def}}{=} \mu_1 \setminus \{(x+y=1), (z+y=2)\}$, returning UNSAT and the conflict clause $c_2^p \stackrel{\text{def}}{=} \{\neg B_1 \vee \neg B_3 \vee \neg B_4\}$, from which $\neg B_1$ is removed by unit-resolution, so that also the clause $c_2^{p\prime} \stackrel{\text{def}}{=} \{\neg B_3 \vee \neg B_4\}$ is learned (a unit-resolved version of clause c_{10}). After further removing B_3 and B_4 from μ_2 the set of $\mathcal{T}$-literals is found $\mathcal{T}$-consistent by $\mathcal{T}$-*solver*, so that no further clause is learned. Then $c_1^{p\prime}, c_2^{p\prime}$ are fed to CHOOSEUNSATISFIEDCLAUSE which selects one and feed it to NEXTTRUTHASSIGNMENT, which flips one literal among B_5, B_7, B_3 and B_4.

4.3 Literal Filterings

Pure-literal Filtering. If some $\mathcal{T}$-atoms occur only positively [resp. negatively] in the original formula (learned clauses and statically-learned clauses are not considered), then we can safely drop every negative [resp. positive] occurrence of them from the assignment μ to be checked by the $\mathcal{T}$-solver [13]. (Intuitively, since such occurrences play no role in satisfying the formula, the resulting partial assignment $\mu^{p\prime}$ still satisfies φ^p.) The benefits of this action is twofold:

(i) reduces the workload for the $\mathcal{T}$-*solver* by feeding it smaller sets;
(ii) increases the chance of finding a $\mathcal{T}$-consistent satisfying assignment by removing "useless" $\mathcal{T}$-literals which may cause the $\mathcal{T}$-inconsistency of μ.

Example 5. Consider the formula φ^p in Fig. 1 and the total truth assignment

$$\mu_4^p = \{B_1, \neg A_3, \neg A_4, \neg A_5, \neg B_6, \neg B_5, B_3, B_4, \neg B_7\}$$

that satisfies φ^p, but is $\mathcal{T}$-inconsistent because of its subset $\{B_1, B_3, B_4\}$ (clause c_{10} in τ^p). Without pure-literal filtering, $\mathcal{T}$-*solver* detects the inconsistency, WALKSMT learns the clause and looks for another assignment. If pure-literal filtering is implemented, instead, since the $\mathcal{T}$-literals $\neg B_5$, B_4 and $\neg B_7$ occur only negatively in the original formula ϕ, they are filtered out from μ_4^p, resulting in the *partial* assignment

$$\eta_4^p = \{B_1, \neg A_3, \neg A_4, \neg A_5, \neg B_6, B_3\},$$

which still satisfies φ^p. $\mathcal{T}$-*solver* is invoked on the corresponding set of $\mathcal{T}$-literals:

$$\eta_4 = \{(x - z > 4), \neg(x + y = 0), (x - y \leq 3)\}.$$

which is $\mathcal{T}$-consistent, from which we can conclude that φ (and ϕ) is $\mathcal{T}$-consistent.

Ghost-literal Filtering. We further enforce the benefits of pure-literal filtering as follows. When a truth assignment μ is found s.t. $\mu^p \models \varphi^p$, before invoking $\mathcal{T}$-*solver* on μ, we check whether any $\mathcal{T}$-atom occurring only positively [resp. negatively] in the original formula and being assigned true [resp. false] in μ can be flipped without falsifying any clause. (This test can be performed very efficiently inside an SLS solver.) If this is the case, then the atom is flipped. This step is repeated until no more such atoms are found, after which the resulting set μ is passed to $\mathcal{T}$-*solver*. This allows for further removing useless $\mathcal{T}$-literals from μ by pure-literal filtering. (Since such literals are a particular case of "ghost literals" [13], we call this enhancement *ghost-literal filtering*.)

Example 6. Consider the formula φ^p in Fig. 1 and the total truth assignment

$$\mu_5^p = \{B_1, A_3, \neg A_4, \neg A_5, \neg B_6, \neg B_5, B_3, \neg B_4, B_7\}$$

that satisfies φ^p. If we apply pure-literal filtering on μ_5^p, then we can filter out only the literal $\neg B_5$ before invoking $\mathcal{T}$-*solver*. By ghost-literal filtering, the literals B_3, $\neg B_4$ and $\neg B_6$ are flipped without falsifying φ^p, resulting in the total truth assignment:

$$\mu_5^{p\prime} = \{B_1, A_3, \neg A_4, \neg A_5, B_6, \neg B_5, \neg B_3, B_4, B_7\}.$$

Now, by pure-literal filtering, we remove from $\mu_5^{p\prime}$ the literals B_3, $\neg B_4$, $\neg B_5$ and $\neg B_6$.

5 Experimental Evaluation

We have implemented two versions of the WALKSMT procedure described above to work for the $\mathcal{LA}(\mathbb{Q})$ theory. The implementation is done on top of MATHSAT4 [7], using part of its preprocessor its $\mathcal{LA}(\mathbb{Q})$-solver [9] and lots of its features. We have

implemented two versions, each using one between two SLS-based SAT solvers: UBCSAT [2] [17] and UBCSAT++ [3] [6]. UBCSAT is a SLS platform providing a very-wide range of SLS algorithms for SAT (including the WalkSAT family), with a very flexible architecding the WalkSAT family), with a very flexible architecture. Among the various SLS procedures provided by UBCSAT, we have chosen to use the Adaptive Novelty$^+$ variant of the WalkSAT family because it was the best-performing in a previous extensive empirical evaluation [15]. UBCSAT++ is built on top of UBCSAT and extends its implementation of Adaptive Novelty$^+$ with the Trimming Variable Selection and Literal Commitment Strategy techniques described in §2.1. We partition the enhancements of WALKSMT of §4 into three groups:

- *Preprocessing and Learning (PL)*, including preprocessing (Initial BCP and Static Learning), Learning and Unit Resolution;
- *Multiple Learning (ML)*;
- *Filtering (FI)*, including both Pure-Literal and Ghost-Literal filterings.

Notationally, we use a "+" [resp. "–"] symbol to denote that an option is enabled [resp. disabled]: e.g., "UBCSAT++ BASIC+PL-ML+FI" denotes WALKSMT based on UBCSAT++ with PL and FI enabled and ML disabled. (Notice that ML requires PL, so that we cannot have "...-PL+ML..." configurations.)

In this section, we evaluate the performance of WALKSMT by comparing its two versions (those based on UBCSAT and UBCSAT++ respectively) against the CDCL-based SMT solver MATHSAT4. We ran MATHSAT4 with all the optimizations enabled (the most important ones are early pruning and $\mathcal{T}$-propagation). [4] We performed our comparison over two distinct sets of instances, which are described in the next two sections: the first consists of the set of all satisfiable $\mathcal{LA}(\mathbb{Q})$ formulas in the SMT-LIB 1.2 (`www.smtlib.org`), whereas the second is composed of randomly-generated problems. All tests were executed on 2.66 GHz Xeon machines running Linux, using a timeout of 600 seconds. The correctness of the models found by WALKSMT have been cross-checked by MATHSAT4. In order to make the experiments reproducible, the full-size plots, the tools, the problems, and the results are available at [1].

5.1 WALKSMT on SMT-LIB Instances

In the first part of our experiments, we compare WALKSMT against MATHSAT on all the satisfiable $\mathcal{LA}(\mathbb{Q})$-formulas (QF_LRA) in the SMT-LIB 1.2. These instances are all classified as "industrial", because they come from the encoding of different real-world problems in formal verification, planning and optimization, and they are divided into six categories: `sc`, `uart`, `sal`, `TM`, `tta_startup` ("tta" hereafter), and `miplib`. [5]

[2] UBCSAT is publicly available at `http://www.satlib.org/ubcsat/`.

[3] UBCSAT++ was kindly provided to us by the developers, Belov and Stachniak.

[4] Although more efficient SMT ($\mathcal{LA}(\mathbb{Q})$) solvers exist, including the recent MATHSAT5, here the choice of MATHSAT4 is aimed at minimizing the differences in performance due to the implementation, because WALKSMT is implemented on top of MATHSAT4 (in particular it uses its preprocessor and $\mathcal{T}$-solver for $\mathcal{LA}(\mathbb{Q})$), so that to better highlight the differences between SLS- and CDCL-based approaches.

[5] Notice that other SMT-LIB categories like `spider_benchmarks` and `clock_synchro` do not contain satisfiable instances and are thus not reported here.

Table 1. Comparison of the number of instances solved within the 600s timeout by the various configurations of WALKSMT and MATHSAT4. Notice that instances solved by the different solvers might not be the same.

Solver	SMT-LIB Instances						Total
	sc	uart	sal	TM	tta	miplib	
Total # of Instances	108	36	11	24	24	22	225
WalkSMT UBCSAT Basic–PL–ML–FI	0	0	0	0	0	0	0
WalkSMT UBCSAT++ Basic–PL–ML–FI	0	0	0	0	0	1	1
WalkSMT UBCSAT Basic+PL–ML–FI	59	10	6	13	5	3	96
WalkSMT UBCSAT++ Basic+PL–ML–FI	46	6	7	17	10	1	87
WalkSMT UBCSAT Basic+PL+ML–FI	103	15	6	12	6	3	145
WalkSMT UBCSAT++ Basic+PL+ML–FI	61	6	7	15	9	1	99
WalkSMT UBCSAT Basic+PL–ML+FI	59	32	10	14	9	3	127
WalkSMT UBCSAT++ Basic+PL–ML+FI	62	12	8	18	10	1	111
WalkSMT UBCSAT Basic+PL+ML+FI	78	35	10	14	9	3	149
WalkSMT UBCSAT++ Basic+PL+ML+FI	63	14	8	19	10	2	116
MATHSAT4	108	36	11	21	24	8	208

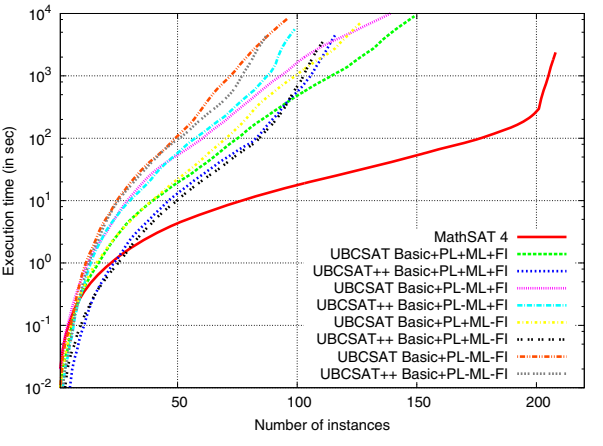


Fig. 2. Cumulative plots of WALKSMT and MATHSAT4 on all SMT-LIB instances

The results of the experiments are reported in Figures 2, 3, 4, 5 and in Table 1. Figure 2 shows the cumulative plots of the execution time for the different configurations of WALKSMT and MATHSAT4 on SMT-LIB instances. (The plots for BASIC-PL-ML-FI are not reported since no formula was solved within the timeout.) Figure 3 compares the best configurations of WALKSMT (BASIC+PL+ML+FI) with UBCSAT (left) and with UBCSAT++ (right) against MATHSAT4 on all instances. Figure 4 shows the relative effects of the different optimizations for WALKSMT with UBCSAT. Figure 5 compares WALKSMT UBCSAT against WALKSMT UBCSAT++ on BASIC+PL+ML+FI versions. The results suggest a list of considerations.

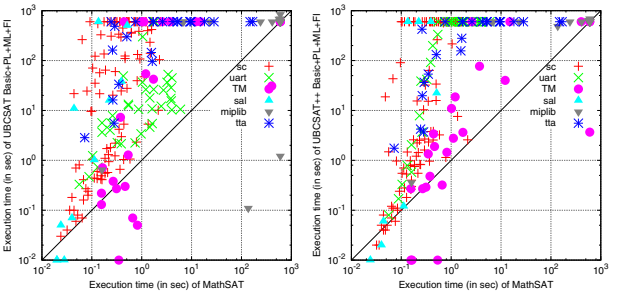


Fig. 3. Comparison of the best configurations of WALKSMT (BASIC+PL+ML+FI) against MATHSAT4 on SMT-LIB instances. Left: with UBCSAT; Center: with UBCSAT++; Right: with UBCSAT++, considering only `miplib` and `TM` benchmarks.

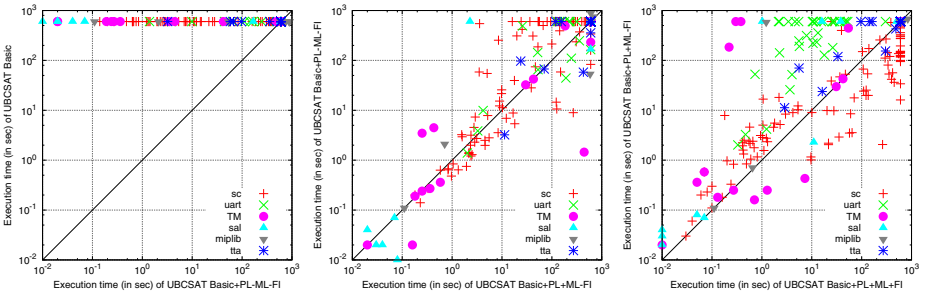


Fig. 4. Pairwise comparison between different configurations of WALKSMT with UBCSAT on SMT-LIB instances, adding increasingly PL, ML and FI to basic WALKSMT.
Left: BASIC-PL-ML-FI vs. BASIC+PL-ML-FI (benefits of adding PL to Basic);
Center: BASIC+PL-ML-FI vs. BASIC+PL+ML-FI (benefits of further adding ML);
Right: BASIC+PL+ML-FI vs. BASIC+PL+ML+FI (benefits of further adding FI).

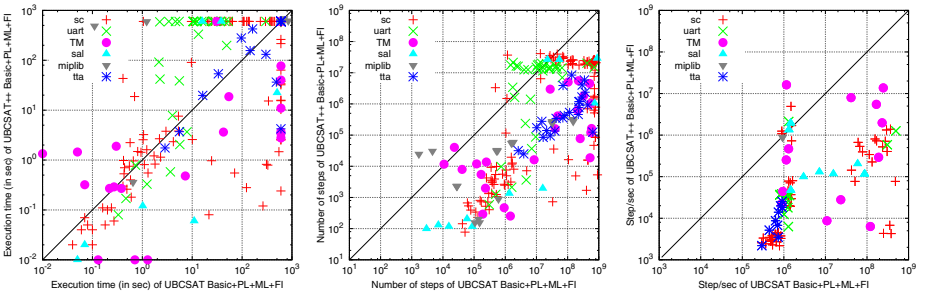


Fig. 5. Comparison between WALKSMT UBCSAT and WALKSMT UBCSAT++ on BASIC+PL+ML+FI versions. Left: CPU time. Center: flip# (on commonly solved instances). Right: average ratio flips#/sec (on commonly solved instances).

First, the optimizations described in §4 lead to dramatic improvements in performance, sometimes by orders of magnitude. Without them, WALKSMT times out on all instances. (See Table 1 and Figures 2 and 4.):

- PL is crucial for performance, since with PL disabled almost no problem is solved within the timeout. In particular, from our data we see that a key role is played by learning. (Which perhaps is not surprising from an SMT perspective, but we believe may be of interest from an SLS perspective.)
- ML produces significant improvements overall, except for a few cases where it may worsen performances (e.g., with `miplib`).
- FI produces strong improvements in performance in all problem categories, (apparently with the exception of the `sc` benchmarks).

Second, globally WALKSMT seems to perform better with UBCSAT than with UBCSAT++, with some exceptions (`TM`, `tta`). From Figure 5, considering the problems solved by both configurations, we see that the total number of flips performed by UBCSAT++ is dramatically smaller than that performed by UBCSAT, but the average cost of each flip is dramatically higher.

Third, globally MATHSAT4 performs much better than WALKSMT, often by orders of magnitude. This mirrors the typical performance gap between CDCL and SLS SAT solvers on industrial benchmarks.

5.2 WALKSMT on Random Instances

Unlike with SAT, in SMT there is very-limited tradition in testing on random problems (e.g., [2,3]). However, for a matter of scientific curiosity and/or to leverage to SMT a popular test for SLS SAT procedures, here we present also a brief comparison of WALKSMT vs. MATHSAT4 on randomly-generated, unstructured 3-CNF $\mathcal{LA}(\mathbb{Q})$-formulas. Each 3-CNF formula is randomly generated according to three integer parameters $\langle m, n, a\rangle$ as follows. First, a distinct $\mathcal{T}$-atoms $\psi_1, ..., \psi_a$ are created, s.t. each atom ψ_j is in the form $(\sum_{i=1}^{4} c_{ji} x_{ji} \leq c_j)$, it is generated by randomly picking four distinct variables x_{ji} out of n variables $\{x_1, ..., x_n\}$, and five integer values $c_{j1}, ..., c_{j4}, c_j$ in the interval $[-100, 100]$. Then, m 3-CNF clauses are randomly generated, each by randomly picking 3 distinct $\mathcal{T}$-atoms in $\{\psi_1, ..., \psi_a\}$, negating each with probability 0.5.

Figure 6 shows the run times of several versions of WALKSMT and MATHSAT4 on the generated formulas, for $n = 20$. Each graph shows curves for WALKSMT (in particular, UBCSAT and UBCSAT++ with the best configuration BASIC+PL+ML+FI) and MATHSAT4 on a group of instances with a fixed number a of $\mathcal{T}$-atoms, for $a = 30, 40, 50, 60$. The plots represent the execution time versus the ratio $r = m/a$ of clauses/$\mathcal{T}$-atoms. Each point in the graphs corresponds to the median run-time of each algorithm on 100 different instances of the same size. (For WALKSMT, each value is itself a median value of 3 runs with different seeds.) The plots show also the satisfiability percentage of each group of instances, defined as the ratio between the satisfiable instances generated and the total number of instances generated, for each value of r. E.g., in the plot in the first column of the first row of Figure 6 the percentage 0.01% for $r = 6$ means that we had to generate and test 10514 formulas (using MATHSAT4 with a timeout of 600 seconds) in order to obtain 100 satisfiable instances.

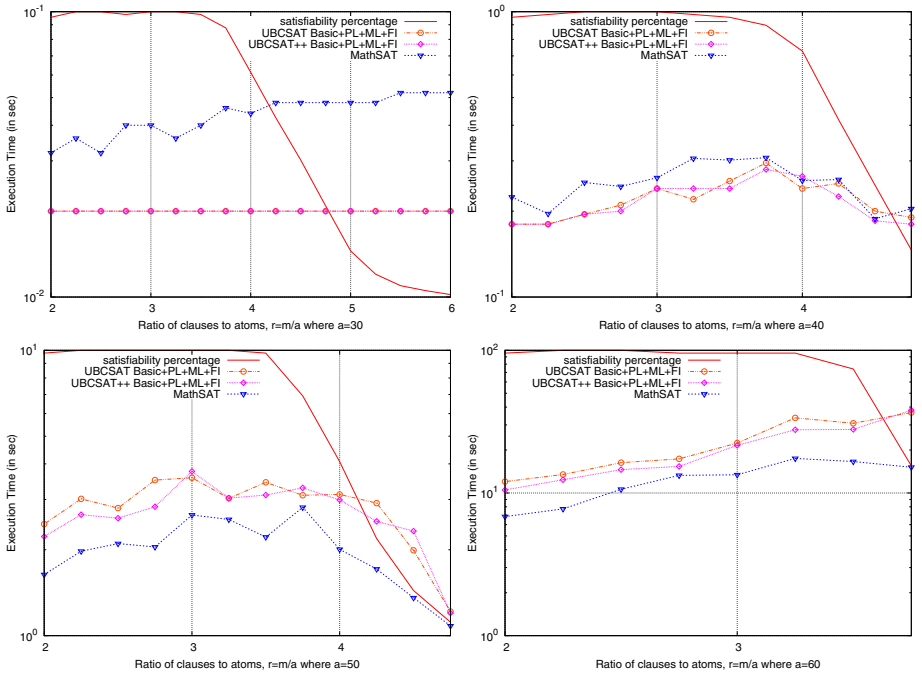


Fig. 6. Comparison of different configurations of WALKSMT and MATHSAT4 on randomly-generated instances with 20 theory variables and atoms $a = 30, 40, 50, 60$

The results show that, unlike with SMT-LIB formulas, on randomly-generated instances there is a very small difference between the performance of UBCSAT BASIC+PL+ML+FI, UBCSAT++ BASIC+PL+ML+FI and MATHSAT4.

6 Conclusions and Future Work

In this paper we have investigated the possibility of using an SLS SAT solver instead of a conventional CDCL-based one as propositional engine for a lazy SMT solver. We have presented and discussed several optimizations to the basic architecture proposed, which allowed WALKSMT to solve a significant amount of industrial SMT problems, although it is still much less efficient that the corresponding CDCL-based SMT solver. We believe that the latter fact is not surprising, since optimization techniques for CDCL-based SMT solvers have been investigated and optimized for the last ten years, whilst to the best of our knowledge this is the first attempt of building a SLS-based one.

This research opens the possibility for several interesting future directions. The first obvious option is to port the implementation to the more-efficient MATHSAT5 and to extend the present work to cover other theories typically used in SMT. We would like to concentrate in particular on "hard" theories such as $\mathcal{LA}(\mathbb{Z})$. Second, we plan to investigate the use of SLS techniques for solving/approximating optimization problems, such as Max-SMT. Third, we will explore the possibility of tightening the synergy between

the SLS SAT solver and $\mathcal{T}$-solvers, for instance by better exploiting information that can be provided by $\mathcal{T}$-solvers when deciding which variables to flip, or by considering architectures in which the search is more driven by the theory part of the formula rather than by the SAT engine. Finally, we plan to work on the integration/combination between SLS-based and CDCL-based SMT solvers, both using a portfolio-like approach and investigating more tightly-coupled solutions.

References

1. http://disi.unitn.it/~rseba/frocos11/tests.tar.gz
2. Armando, A., Castellini, C., Giunchiglia, E.: SAT-based procedures for temporal reasoning. In: Proc. European Conference on Planning, CP 1999 (1999)
3. Audemard, G., Bertoli, P., Cimatti, A., Korniłowicz, A., Sebastiani, R.: A SAT Based Approach for Solving Formulas over Boolean and Linear Mathematical Propositions. In: Voronkov, A. (ed.) CADE 2002. LNCS (LNAI), vol. 2392, pp. 195–210. Springer, Heidelberg (2002)
4. Audemard, G., Lagniez, J.-M., Mazure, B., Sais, L.: Boosting local search thanks to CDCL. In: Fermüller, C.G., Voronkov, A. (eds.) LPAR-17. LNCS, vol. 6397, pp. 474–488. Springer, Heidelberg (2010)
5. Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability Modulo Theories. In: Handbook of Satisfiability. ch. 26, pp. 825–885. IOS Press, Amsterdam (2009)
6. Belov, A., Stachniak, Z.: Improving variable selection process in stochastic local search for propositional satisfiability. In: Kullmann, O. (ed.) SAT 2009. LNCS, vol. 5584, pp. 258–264. Springer, Heidelberg (2009)
7. Bruttomesso, R., Cimatti, A., Franzén, A., Griggio, A., Sebastiani, R.: The MATHSAT 4 SMT solver. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 299–303. Springer, Heidelberg (2008)
8. Cotton, S., Maler, O.: Fast and Flexible Difference Constraint Propagation for DPLL(T). In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 170–183. Springer, Heidelberg (2006)
9. Dutertre, B., de Moura, L.: A Fast Linear-Arithmetic Solver for DPLL(T). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer, Heidelberg (2006)
10. Hoos, H.H., Stutzle, T.: Local Search Algorithms for SAT: An Empirical Evaluation. Journal of Automated Reasoning 24(4) (2000)
11. Hoos, H.H., Stutzle, T.: Stochastic Local Search Foundation And Application. Morgan Kaufmann, San Francisco (2005)
12. Minton, S., Johnston, M.D., Philips, A.B., Laird, P.: Minimizing Conflicts: A Heuristic Repair Method for Constraint-Satisfaction and Scheduling Problems. Artificial Intelligence 58(1) (1992)
13. Sebastiani, R.: Lazy Satisfiability Modulo Theories. Journal on Satisfiability, Boolean Modeling and Computation, JSAT 3 (2007)
14. Selman, B., Kautz, H.A., Cohen, B.: Noise strategies for improving local search. In: Proc. AAAI. MIT Press, Cambridge (1994)
15. Tomasi, S.: Stochastic Local Search for SMT. Technical report, DISI-10-060, DISI, University of Trento (2010), http://eprints.biblio.unitn.it/
16. Tompkins, D., Hoos, H.: Novelty+ and Adaptive Novelty+. In: SAT 2004 Competition Booklet (2004)
17. Tompkins, D., Hoos, H.: UBCSAT: An Implementation and Experimentation Environment for SLS Algorithms for SAT and MAX-SAT. In: Hoos, H., Mitchell, D.G. (eds.) SAT 2004. LNCS, vol. 3542, pp. 306–320. Springer, Heidelberg (2005)

Controlled Term Rewriting*

Florent Jacquemard[1], Yoshiharu Kojima[2,3], and Masahiko Sakai[2]

[1] INRIA & LSV, ENS Cachan, 61 av. Pdt Wilson 94230 Cachan, France
florent.jacquemard@inria.fr
[2] Graduate School of Information Science, Nagoya University
Furo-cho, Chikusa-ku, Nagoya, 464-8603 Japan
{kojima@trs.cm.,sakai@}is.nagoya-u.ac.jp
[3] Research Fellow of the Japan Society for the Promotion of Science

Abstract. Motivated by the problem of verification of imperative tree transformation programs, we study the combination, called controlled term rewriting systems (CntTRS), of term rewriting rules with constraints selecting the possible rewrite positions. These constraints are specified, for each rewrite rule, by a selection automaton which defines a set of positions in a term based on tree automata computations.

We show that reachability is PSPACE-complete for so-called monotonic CntTRS, such that the size of every left-hand-side of every rewrite rule is larger or equal to the size of the corresponding right-hand-side, and also for the class of context-free non-collapsing CntTRS, which transform Context-Free (CF) tree language into CF tree languages.

When allowing size-reducing rules, reachability becomes undecidable, even for flat CntTRS (both sides of rewrite rules are of depth at most one) when restricting to words (*i.e.* function symbols have arity at most one), and for ground CntTRS (rewrite rules have no variables).

We also consider a restricted version of the control such that a position is selected if the sequence of symbols on the path from that position to the root of the tree belongs to a given regular language. This restriction enables decision results in the above cases.

Introduction

Term rewriting is a rule-based formalism for describing computations in ranked terms. In the context of formal verification, term rewriting systems (TRS) can be used to provide a finite abstraction of the dynamics of a system whose configurations are represented by ranked terms. In this case, the rewrite relation represents the transitions between configurations. For instance, functional programs manipulating (tree) structured data values with pattern matching can be described by rewrite rules [18] such that the rewriting relation represents the program evaluation. This approach can also be applied to systems [1] or imperative programs [4,17] modifying some parts of tree shaped data structures in place, while leaving the rest unchanged. These update operations can be modeled by rewrite rules or similarly, tree transducers. A crucial problem for the

* This work has been supported by the ERC research project FoX under grant agreement FP7-ICT-233599 and by the INRIA ARC project ACCESS.

C. Tinelli and V. Sofronie-Stokkermans (Eds.): FroCoS 2011, LNAI 6989, pp. 179–194, 2011.

automatic reachability and flow analysis of programs is to find finite and decidable representations of the closure the above TRS representations of sets of terms representing configurations. This approach is related to *static type checking* of XML transformations (see *e.g.* [22]), which is the problem to verify that a program always converts valid source trees (documents) into valid output trees (where types are defined by TA). It has also been shown useful for the analysis of consistency of XML read and write access control policies [17].

A rewrite rule is an oriented equation, whose left-hand-side (*lhs*) describe a pattern to be replaced in a term, and whose right-hand-side (*rhs*) is the new term for replacement. It can be applied at any position in a term, providing that the *lhs* matches the subterm at this position. For instance, a rule with *lhs* $a(x_1, x_2)$ can be applied at any position labelled by a. This simple approach is in general relevant in domains like theorem proving or algebraic computation. For some applications however, like the analysis of programs for XML document transformations or of access-control policies, it is important to be able to express explicitly some *context conditions* to be checked before applying a rewrite rule. For instance, one may want to rename the label at some position with a rewrite rule $a(x) \to b(x)$ providing that there is no occurrence of b above the position to be rewritten (position labeled with a). Some standard XML transformation languages like XSLT or XQuery update [6], use the path specification language XPath or a XQuery expressions in order to define the position where the transformation can be applied.

In this paper, we study the so called *controlled term rewriting systems* (CntTRS) in the context of regular tree model checking. They are defined by the combination of term rewriting rules with some constraints (called control) specifying the possible rewrite positions. In order to define the constraints, we have chosen a model similar to the selection tree automata (SA) of [13], which, intuitively, select positions in a term based on the computations of a tree automaton. This gives a powerful selection mechanism, with the same expressiveness as the monadic second order logic of the tree, or monadic Datalog [14]. We consider also a restriction of the SA where a position p in a term t is selected if the sequence of symbols on the path from p to the root of t belongs to a given regular language. The corresponding restricted rewriting model is called prefix CntTRS, or pCntTRS. It turns out quickly (Examples at the end of Section 1) that even the restricted pCntTRS are actually too powerful for preserving regularity, even for very simple rewrite rules.

Therefore, we consider in Section 2 the classes of *context-free* (CF) and *context-sensitive* (CS) tree languages, which are both strictly larger than the class of TA languages (also called *regular* tree languages). We also define the so called CF and *monotonic* classes of rewrite systems (without control). A rewrite rule is CF if its *lhs* is of the form $f(x_1, \ldots, x_n)$ where $x_1, \ldots, x_n$ are distinct variables, and monotonic if the size of its *lhs* is larger or equal to the size of its *rhs*. We show that CF and monotonic TRS respectively preserve CF tree languages and CS tree languages.

The monotonic uncontrolled rewrite rules already have the full power of CS tree grammars. Adding control with SA does not improve their expressiveness, and it follows that reachability is PSPACE-complete and model checking undecidable for monotonic CntTRS (Section 3.1). Similar results also hold for CF CntTRS without collapsing rules (projection rules of the form $f(x_1, \ldots, x_n) \to x_i$), even when restricting to prefix control (Section 3.2).

When allowing depth-reducing rules (Section 3.3), reachability becomes undecidable, even for flat CntTRS (*lhs* and *rhs* of rewrite rules are of depth at most one) and in the case of words (*i.e.* function symbols of arity at most one). Similarly, reachability is undecidable for ground CntTRS (rewrite rules have no variables). When restricting to words, prefix control and flat rewrite rules, reachability is decidable in PSPACE.

Finally, we consider in Section 4 a relaxed form of the prefix control of rewrite rules, where the selection is done by considering the term in input modulo the rewrite relation. We obtain a regularity preservation result for this recursive form of prefix controlled rewriting, using alternating tree automata with ε-transitions.

Related Work. Controlled rewrite systems have been introduced in the case of word rewriting, see [28] for a survey, and also [9] for the case of conditional context-free (word) grammars, which are mentioned in Section 3.

Regarding term rewriting, there have been many studies for finding syntactical restriction on term rewrite rules ensuring the preservation of regularity, see *e.g.* [12]. This is the case for instance of linear and flat as well as ground TRS without control, in contrast to the results of Section 3.3 for their controlled counterpart.

Many strategies have been proposed for term rewriting, most often for efficiency purposes (our goal here is rather to study the expressiveness and decidability results for controlled rewriting). We cannot mention all of them, and will just give some elements of comparison with controlled rewriting. The *innermost* strategy, which is the analogous of *call-by-value* for functional languages (a subterm can be rewritten only when all its proper subterms are no more rewritable) can be expressed in controlled rewriting for left-linear TRS, because the set of non-rewritable terms (the *normal forms*) for such TRS are recognizable by tree automata (see *e.g.* [8]). Some results of preservation of regularity for the innermost and the outermost and leftmost rewrite strategies can be found in [26] In the *context sensitive* strategy [21] (which is not related to the context-sensitive term languages presented in this paper), the rewriting positions are selected according to a mapping μ associating to every symbol of the signature the subset of the indexes of its argument that can be rewritten. More precisely, the positions selected for rewriting in a term $f(t_1, \ldots, t_n)$ are the root position and all the positions selected in every t_i such that $i \in \mu(f)$. This set of positions can be defined by SA, *i.e.* context-sensitive rewriting is a particular case of controlled rewriting. It is a strict subcase because with the context-sensitive strategy, the root position is always rewritable whereas this is not the case for controlled rewriting. A result of preservation of regularity for this rewrite strategy was established in [19]. Another related topic is the optimal *call-by-need* rewrite strategies for

TRS. In [7] and [10], it is shown how to select the *needed redexes* (positions at which rewriting must be performed in order to transform a term to its normal form) using monadic second order logic formulae, which is equivalent as using SA.

Top-down tree transducers with regular look-ahead [11] are an extension of top-down tree transducers, where a transition can be fired provided that the current subtree belongs to some given regular tree language. This is similar to our notion of control for rewrite systems. A notable difference however is that the transducers transform terms in one (top-down) pass, whereas we consider here the terms computed by an arbitrary iteration of controlled rewrite rules.

1 Preliminaries

Terms. We use the standard notations for terms and positions, see [2]. A *signature* Σ is a finite set of function symbols with arity. We denote the subset of function symbols of Σ of arity n as Σ_n. Given an infinite set $\mathcal{X}$ of variables, the set of terms built over Σ and $\mathcal{X}$ is denoted $\mathcal{T}(\Sigma, \mathcal{X})$, and the subset of *ground* terms (terms without variables) is denoted $\mathcal{T}(\Sigma)$. The set of variables occurring in a term $t \in \mathcal{T}(\Sigma, \mathcal{X})$ is denoted $vars(t)$. A signature is called *unary* if it contains only symbols of arity 1 and one symbol $\perp$ of arity 0. We make no distinction below between ground terms over a unary signature Σ and words of Σ_1^*. More precisely, a term $a_1(a_2(\dots a_n(\perp)))$ will be represented by the string $a_1\, a_2 \dots a_n$ (the constant symbol $\perp$ is forgotten in this representation).

A term $t \in \mathcal{T}(\Sigma, \mathcal{X})$ can be seen as a function from its set of positions $\mathcal{P}os(t)$ into $\Sigma \cup \mathcal{X}$. Positions in terms are denoted by sequences of natural numbers, ε is the empty sequence (root position), and $p.p'$ denotes the concatenation of positions p and p'. The set $\mathcal{P}os(t)$ of positions of the term t is defined recursively as $\mathcal{P}os\big(f(t_1, \dots, t_m)\big) = \{\varepsilon\} \cup \{i.p \mid i \in \{1, \dots, m\} \wedge p \in \mathcal{P}os(t_i)\}$. The *height* of a term t, denoted $h(t)$, is the maximal length of a position of $\mathcal{P}os(t)$. The *size* $\|t\|$ of a term t is the cardinality of $\mathcal{P}os(t)$.

The *subterm* of t at position p is denoted $t|_p$ defined by $t|_\varepsilon = t$ and $f(t_1, \dots, t_m)|_{i.p} = t_i|_p$. The term obtained from t by replacing subterm of t at position p by s, is denoted $t[s]_p$. The notation $t = t[s]_p$ may also be used to emphasize that $t|_p$ is s.

A *substitution* is a mapping $\mathcal{X} \to \mathcal{T}(\Sigma, \mathcal{X})$. Substitutions can also be applied to arbitrary terms by homomorphically extending its application to variables. The application of a substitution σ to a term t, denoted as $t\sigma$, is defined as follows for non-variable terms: $f(t_1, \dots, t_n)\sigma = f(t_1\sigma, \dots, t_n\sigma)$. A *variable renaming* is a substitution from variables to variables.

A term is *linear* if no variable occurs more than once in it. A term is *shallow* if each occurrence of variables is at most depth one, and *flat* if its height is at most one.

A *context* of dimension n is a linear term $C \in \mathcal{T}(\Sigma, \{x_1, \dots, x_n\})$. When $C = x_1$, it is called the *empty context*. Given a context C of dimension n and n terms $t_1, \dots, t_n \in \mathcal{T}(\Sigma, \mathcal{X})$, we write $C[t_1, \dots, t_n]$ to denote $C\sigma$ where σ is the substitution $\{x_1 \mapsto t_1, \dots, x_n \mapsto t_n\}$.

Tree Automata. A *tree automaton* (TA) $\mathcal{A}$ over a signature Σ is a tuple $\langle Q, F, \Delta\rangle$ where Q is a finite set of nullary state symbols, disjoint from Σ, F is a set of states of Q called final states, Δ is a set of transition rules of the form: $f(q_1, \ldots, q_n) \to q$, where $f \in \Sigma_n$, and $q_1, \ldots, q_n, q \in Q$. Sometimes, we shall refer to $\mathcal{A}$ as a subscript of its components, like in $Q_\mathcal{A}$ to indicate that Q is the state set of $\mathcal{A}$. The size of $\mathcal{A}$ is $\|\mathcal{A}\| = \sum_{f(q_1,\ldots,q_n)\to q\in\Delta}(n+2)$. Transition from s to t in one step by a TA $\mathcal{A}$ is denoted by $s \xrightarrow[\mathcal{A}]{} t$, and its reflexive and transitive closure is denoted by $s \xrightarrow[\mathcal{A}]{*} t$.

A *run* of $\mathcal{A}$ on a term $t \in \mathcal{T}(\Sigma)$ is a mapping ρ from $\mathcal{P}os(t)$ into $Q_\mathcal{A}$ such that for all $p \in \mathcal{P}os(t)$, $t(p)\big(\rho(p.1), \ldots, \rho(p.n)\big) \to \rho(p)$ is in $\Delta_\mathcal{A}$, where n is the arity of the symbol $t(p)$ in Σ. The run ρ is called *successful* (or *accepting*) if $\rho(\varepsilon)$ is in $F_\mathcal{A}$. The set of successful runs of $\mathcal{A}$ on t is denoted $sruns(\mathcal{A}, t)$. For the sake of conciseness, we shall sometimes apply term-like notations (subterm, replacement...) to runs. The language $\mathcal{L}(\mathcal{A})$ of $\mathcal{A}$ is the set of terms t for which $sruns(\mathcal{A}, t)$ is not empty.

Selection Automata. Besides being able to recognize terms, tree automata can also be used to select positions in a term [13,24]. We propose here a definition of position selection by TA very close to [13].

A *selection automaton* (SA) $\mathcal{A}$ over a signature Σ is a tuple $\langle Q, F, S, \Delta\rangle$ where $\langle Q, F, \Delta\rangle$ is a tree automaton denoted $ta(\mathcal{A})$ and S is a set of states of Q called selection states. Given a SA $\mathcal{A}$ and a term $t \in \mathcal{T}(\Sigma)$, the set of positions of t selected by $\mathcal{A}$ is defined as

$$sel(\mathcal{A}, t) = \{p \in \mathcal{P}os(t) \mid \exists \rho \in sruns(ta(\mathcal{A}), t), \rho(p) \in S\}.$$

Note that it is required that t is recognized by $\mathcal{A}$ in order to select positions.

We shall consider below a restricted kind of selection by TA, where a position p in a term t is selected only according to its strict prefix (*i.e.* the sequence of symbols labeling the path from the root of t down to the immediate ancestor of p), which is tested for membership to a regular (word) language. More precisely, a selection automaton $\mathcal{A} = \langle Q, F, S, \Delta\rangle$ is called *prefix* if Q contains two special states: q_0 (universal state) and q_s (selection state), $F \subseteq Q\backslash\{q_0\}$, $S = \{q_s\}$, and Δ contains $f(q_0, \ldots, q_0) \to q_0$ and $f(q_0, \ldots, q_0) \to q_s$ for all $f \in \Sigma$, and Δ contains some other transition rules of the form $f(q_1, \ldots, q_n) \to q$ where $q \in Q \setminus \{q_0, q_s\}$ and there exists exactly one $i \leq n$ such that $q_i \neq q_0$. Intuitively, assume that we are given a finite automaton $\mathcal{B}$ defining the strict prefixes of selected positions. Then q_s is the initial state of $\mathcal{B}$, F is the set of final states of $\mathcal{B}$, and for all $a \in \Sigma_n$, Δ contains n rules $a(q_0, \ldots, q_0, q', q_0, \ldots, q_0) \to q$ for each transition $q' \xrightarrow{a} q$ of $\mathcal{B}$. Note that with this definition, the root position is always selected by a by a prefix selection automaton.

Controlled Term Rewriting Systems. We propose a formalism that strictly extends standard term rewriting systems [2] by the restricting the possible rewrite positions to positions selected by a given SA. Formally, a *controlled term rewriting system* (CntTRS) $\mathcal{R}$ over Σ is a finite set of *controlled rewrite rules*

of the form $\mathcal{A} : \ell \to r$, made of a SA $\mathcal{A}$ over Σ and a *rewrite rule* $\ell \to r$ such that $\ell \in \mathcal{T}(\Sigma, \mathcal{X}) \setminus \mathcal{X}$ (the left-hand side of the rule), and $r \in \mathcal{T}(\Sigma, vars(\ell))$ (the right-hand side). Ths size of $\mathcal{R}$ is the number of the rewrite rules inn $\mathcal{R}$.

A term s rewrites to t in one step by an CntTRS $\mathcal{R}$, denoted by $s \xrightarrow[\mathcal{R}]{} t$, if there exists a controlled rewrite rule $\mathcal{A} : \ell \to r \in \mathcal{R}$, a position $p \in sel(\mathcal{A}, s)$, and a substitution σ such that $s|_p = \ell\sigma$ and $t = s[r\sigma]_p$. In this case, s is said to be $\mathcal{R}$-*reducible*, and otherwise s is called an $\mathcal{R}$-*normal form*. The reflexive and transitive closure, and reflexive, symmetric and transitive closure of $\xrightarrow[\mathcal{R}]{}$ are denoted as $\xrightarrow[\mathcal{R}]{*}$ and $\xleftrightarrow[\mathcal{R}]{*}$, and $\xrightarrow[\mathcal{R}]{=}$ denotes the union of $\xrightarrow[\mathcal{R}]{}$ and $=$.

Example 1. Let us consider the CntTRS $\mathcal{R} = \{(1)\, \mathcal{A}_1 : a \to c, (2)\, \mathcal{A}_2 : b \to c, (3)\, \mathcal{A}_3 : f(x, y) \to g(x, y)\}$ where each SA is as follows ($Q = \{q_1, q_2, q_f\}$):

$$\mathcal{A}_1 = \langle Q, \{q_f\}, \{q_1\}, \{a \to q_1, b \to q_2, f(q_1, q_2) \to q_f\}\rangle$$
$$\mathcal{A}_2 = \langle Q, \{q_f\}, \{q_2\}, \{c \to q_1, b \to q_2, g(q_1, q_2) \to q_f\}\rangle$$
$$\mathcal{A}_3 = \langle Q, \{q_f\}, \{q_f\}, \{c \to q_1, b \to q_2, f(q_1, q_2) \to q_f\}\rangle$$

Then, possible rewriting from the term $f(a, b)$ by $\mathcal{R}$ is $f(a, b) \xrightarrow[(1)]{} f(c, b)$ since $sel(\mathcal{A}_1, f(a, b)) = \{1\}$ and the subterm at the position 1 is a. Similarly, we have $f(c, b) \xrightarrow[(3)]{} g(c, b) \xrightarrow[(2)]{} g(c, c)$ where $sel(\mathcal{A}_3, f(c, b)) = \{\varepsilon\}$, and $sel(\mathcal{A}_2, g(c, b)) = \{2\}$. □

We call *prefix controlled term rewriting system* (pCntTRS), resp. *term rewriting systems* (TRS), the special cases of CntTRS such that every SA in a controlled rewrite rule is a prefix SA, resp. is the *universal* SA $\mathcal{A}_0 = \langle\{q_0\}, \{q_0\}, \{q_0\}, \{f(q_0, \ldots, q_0) \to q_0 \mid f \in \Sigma\}\rangle$. In the latter case, $\mathcal{A}_0$ may be dropped when defining the rewrite rules, *i.e.* we present a TRS as a finite set of uncontrolled rewrite rules, as usual.

A controlled rewrite rule $\mathcal{A} : \ell \to r$ is ground, *flat*, *linear*, *shallow* if ℓ and r are so. It is *collapsing* if r is a variable. A CntTRS is *flat*, *linear*, etc if all its rules are so.

Decision Problems. The *closure* of a ground term set L by a CntTRS $\mathcal{R}$ is $\{t \mid \exists s \in L, s \xrightarrow[\mathcal{R}]{*} t\}$ (it is sometimes denoted $\mathcal{R}^*(L)$).
Ground reachability is the problem to decide, given two ground terms $s, t \in \mathcal{T}(\Sigma)$ and a CntTRS $\mathcal{R}$ whether $s \xrightarrow[\mathcal{R}]{*} t$. *Regular Model checking* (RMC) is the problem to decide, given two TA languages L_{in} and L_{err} and a CntTRS $\mathcal{R}$ whether $\mathcal{R}^*(L_{\mathsf{in}}) \cap L_{\mathsf{err}} = \emptyset$. The name of the problem is coined after state exploration techniques for checking safety properties. In this setting, L_{in} and L_{err} represent (possibly infinite) sets of initial, respectively error, states.

Example 2. Let us consider the following CntTRS $\mathcal{R}$ over the unary signature Σ with $\Sigma_1 = \{a, b, c, d\}$ and $\Sigma_0 = \{\bot\}$. Let $\mathcal{R}$ be the CntTRS containing the following controlled rewrite rules. The SA of these rules select one position per term, and they are represented by a regular expression where the selected letter is underlined.

$$(1)\ c^*a^*d^*\underline{d}\,b^* \ : d(x) \to b'(x) \quad (2)\ c^*\underline{c}\,a^*d^*b'b^* : \ c(x) \to a'(x)$$
$$(3)\ c^*a'a^*d^*\underline{b'}\,b^* : b'(x) \to b(x) \quad (4)\ c^*\underline{a'}\,a^*d^*b^* \ : a'(x) \to a(x)$$

More precisely, the SA for the above rules are respectively (Q is the state set $\{q_a, q_b, q_c, q_d, q\}$)

$$\begin{aligned}
\mathcal{A}_1 &= \langle Q, \{q_c, q_a, q_d, q\}, \{q\}, \{\bot \to q_b, b(q_b) \to q_b, d(q_b) \to q, d(q|q_d) \to q_d,\\
&\quad a(q|q_d|q_a) \to q_a, c(q_a|q_c|q) \to q_c\}\rangle\\
\mathcal{A}_2 &= \langle Q, \{q_c, q\}, \{q\}, \{\bot \to q_b, b(q_b) \to q_b, b'(q_b) \to q_d, d(q_d) \to q_d,\\
&\quad a(q_d|q_a) \to q_a, c(q_a) \to q, c(q|q_c) \to q_c\}\rangle\\
\mathcal{A}_3 &= \langle Q, \{q_c\}, \{q\}, \{\bot \to q_b, b(q_b) \to q_b, b'(q_b) \to q, d(q|q_d) \to q_d, a(q|q_d) \to q_a,\\
&\quad a'(q|q_d|q_a) \to q_c, c(q_c) \to q_c\}\rangle\\
\mathcal{A}_4 &= \langle Q, \{q_c, q\}, \{q\}, \{\bot \to q_b, b(q_b) \to q_b, d(q_b|q_d) \to q_d, a(q_b|q_d|q_a) \to q_a,\\
&\quad a'(q_b|q_d|q_a) \to q, c(q|q_c) \to q_c\}\rangle
\end{aligned}$$

Note that these SA are all deterministic. The SA $\mathcal{A}_1$ selects the last d (starting from the top), $\mathcal{A}_2$ selects the last c when there is a b', $\mathcal{A}_3$ selects the b' when there is a a', and $\mathcal{A}_4$ selects the a' when there is no b'. The closure of the regular tree language $L = c^+(d^+(\bot))$ by the CntTRS is such that $R^*(L) \cap a^*(b^*(\bot)) = \{a^n b^n \leq n \geq 0\}$. Therefore, $R^*(L)$ is a non regular tree language (it is a context-free tree language).

Example 3. Using the same signature as in Example 2, we can obtain a context-free set of descendants with a flat pCntTRS. Indeed, intersection of a^*b^* and the closure of c^*d^* by the following set of rewrite rules is $\{a^n b^m \mid n \geq m\}$ which is CF and not regular. In the rewrite rules, we use an informal description of the languages of the prefix allowed, instead of giving explicitly the prefix SA.

no a', no a : $c(x) \to a'(x)$, exactly one a' : $d(x) \to b'(x)$,
no a', no a : $a'(x) \to a(x)$, no a', no b', no b : $b'(x) \to b(x)$.

It is not difficult to generalize the construction of Example 3 in order to obtain a context sensitive rewrite closure of the form $\{a^n b^m c^p \mid n \geq m \geq p\}$, starting from a regular set of the form $c^*d^*e^*$ and using a flat pCntTRS.

2 CF and CS Tree Languages and TRS

In this section, we define the context-free and context-sensitive sets of terms, and give properties of their closure under term rewriting.
A rewrite rule over Σ is called

context-free (CF) if it is of the form $f(x_1, \ldots, x_n) \to r$ where $r \in \mathcal{T}\big(\Sigma, \{x_1, \ldots, x_n\}\big)$, $x_1, \ldots, x_n$ are distinct variables and $f \in \Sigma_n$. Recall that when $r = x_i$ for some $i \leq n$, then the rule is called collapsing.

monotonic if it is of the form $C[x_1, \ldots, x_n] \to D[x_1, \ldots, x_n]$ where C and D are two contexts over Σ and such that $\|C\| \leq \|D\|$ and $x_1, \ldots x_n$ are distinct variables (note that it implies that the rule is linear).

A *tree grammar* (TG, see *e.g.* [8]) is a tuple $\mathcal{G} = \langle \mathcal{N}, S, \Sigma, P\rangle$ where $\mathcal{N}$ is a finite set of *non-terminal* symbols, each with an arity, $S \in \mathcal{N}$ has arity 0, it is called the *axiom* of $\mathcal{G}$, Σ is a signature disjoint from $\mathcal{N}$, (its elements are also

called *terminal* symbols) and P is a set of (uncontrolled) rewrite rules, called *production rules*, of the form $\ell \to r$ where ℓ, r are terms of $\mathcal{T}(\Sigma \cup \mathcal{N}, \mathcal{X})$ such that ℓ contains at least one non-terminal. The tree grammar $\mathcal{G}$ is *regular* if all non-terminal symbols of $\mathcal{N}$ have arity 0 and all production rules of P have the form $A \to r$, with $A \in \mathcal{N}$ and $r \in \mathcal{T}(\Sigma \cup \mathcal{N})$. It is *context-free* (CFTG), resp. *context-sensitive* (CSTG) if all production rules are CF, resp. monotonic. In the two later cases, we assume from now on *wlog* that every production rule either has the form $A(x_1, \ldots, x_n) \to a(x_1, \ldots, x_n)$ where $A \in \mathcal{N}$ and $a \in \Sigma_n$, or it does not contain terminal symbols of Σ, by introducing the non-terminal symbol $\langle b \rangle$, the production rule $\langle b \rangle \to b$, and replace all b in the other production rules by $\langle b \rangle$.

The *language* generated by $\mathcal{G}$, denoted by $L(\mathcal{G})$, is the set of terms of $\mathcal{T}(\Sigma)$ which can be reached by successive applications of the production rules, starting from the axiom, *i.e.* $L(\mathcal{G}) = \{t \in \mathcal{T}(\Sigma) \mid S \xrightarrow[P]{*} t\}$. A tree language is called regular (resp. CF, CS) if it is the language of a regular (resp. CF, CS) grammar.

Note that the classical cases of word languages are particular cases of the above, if the symbols of $\mathcal{N} \cup \Sigma$ are unary symbols of a unary signature.

The regular tree grammars are equivalent in expressiveness to TA. There exists a model of pushdown TA equivalent to the CF tree grammars [15].
The *membership* problem is, given a tree grammar $\mathcal{G}$ over Σ and a term $t \in \mathcal{T}(\Sigma)$, to decide whether $t \in \mathcal{L}(\mathcal{G})$.
The *emptiness* problem is, given a tree grammar $\mathcal{G}$, to decide whether $\mathcal{L}(\mathcal{G}) = \emptyset$.

Proposition 1. *Membership and emptiness are decidable in PTIME for CFTG.*

The following result (perhaps a folklore knowledge) is almost immediate from the above definitions.

Proposition 2. *Given a CFTG $\mathcal{G}$ and a CF TRS $\mathcal{R}$, one can construct in PTIME a CFTG generating the closure of $L(\mathcal{G})$ by $\mathcal{R}$, and whose size is polynomial in the size of $\mathcal{G}$ and $\mathcal{R}$.*

Proof. Let $\mathcal{G} = \langle \mathcal{N}, S, \Sigma, P \rangle$ be a CFTG and $\mathcal{R}$ a CF TRS over Σ. For all $a \in \Sigma$, we create a new non-terminal N_a with the same arity as a. Let $\mathcal{N}' = \mathcal{N} \cup \{N_a \mid a \in \Sigma\}$, and let P' be obtained from P by replacing every production rule $A(x_1, \ldots, x_n) \to a(x_1, \ldots, x_n)$, with $A \in \mathcal{N}$ and $a \in \Sigma$, by $A(x_1, \ldots, x_n) \to N_a(x_1, \ldots, x_n)$. Moreover, we add to P' the rules obtained from the rules of $\mathcal{R}$ by replacing every symbol $a \in \Sigma$ by N_a. The CFTG $\mathcal{G}' = \langle \mathcal{N}', S, \Sigma, P' \rangle$ generates the closure of $L(\mathcal{G})$ by $\mathcal{R}$. □

Corollary 1. *Reachability and RMC are decidable in PTIME for CF TRS.*

Proof. For the RMC, we use the fact that the intersection of the languages of a CF tree grammar $\mathcal{G}$ and a TA $\mathcal{A}$ is the language of a CF tree grammar whose size is the product of the respective sizes of $\mathcal{G}$ and $\mathcal{A}$. □

Note however that *joinability*, the problem to decide, given two ground terms $s, t \in \mathcal{T}(\Sigma)$ and a TRS $\mathcal{R}$, whether there exists $u \in \mathcal{T}(\Sigma)$ such that $s \xrightarrow[\mathcal{R}]{*}$

$u \xleftarrow[\mathcal{R}]{*} t$, is undecidable for CF TRS [5], because the emptiness of intersection is undecidable for CF tree languages.

We can also observe that the CF TRS are left linear, but in general not right linear. They are the symmetric the so called right-linear, monadic and non-collapsing TRS, whose rules have the form $f(x_1, \ldots, x_n) \to r$, where $r \in \mathcal{T}(\Sigma, \{x_1, \ldots, x_n\}) \backslash \mathcal{X}$. It has been shown that these TRS preserve regularity [23]: the closure of a regular tree language by such a TRS is a regular tree language. The decidability of reachability for CF TRS is already a consequence of this former result. It has been observed, see *e.g.* [16], that in several cases, one class of word rewrite system preserves regularity and its symmetric class preserves CF languages. We have here an example of such a situation in the case of terms.

To our knowledge, the case of CSTG and monotonic TRS was not studied before but it is not very surprising.

Proposition 3. *Membership is PSPACE-complete for CSTG.*

Proof. The hardness is an immediate consequence of the same result for CS (word) grammars [20], which are a particular case of CSTG. For the decision algorithm, let $\mathcal{G}$ be a CSTG over Σ and let $t \in \mathcal{T}(\Sigma)$ be given. We can observe that if two terms s and s' over the non-terminal and terminal symbols of the CSTG $\mathcal{G}$ are successive in a derivation starting from the axiom S of $\mathcal{G}$, then the size $\|s'\|$ is larger or equal to $\|s\|$. Hence, if there is a derivation from S to t by $\mathcal{G}$, then all the terms in this derivation have a size smaller or equal to $\|t\|$. Hence, it is possible to construct a linear bounded automata which, starting from t, will search backward (non deterministically) a derivation from S. The detailed construction is given in the long version of this paper. □

Proposition 4. *Emptiness is undecidable for CSTG.*

Proof. It is a consequence of the same result for CS (word) grammars. □

Proposition 5. *Given a CSTG $\mathcal{G}$ and a monotonic TRS $\mathcal{R}$, one can construct in PTIME a CSTG generating the closure of $L(\mathcal{G})$ by $\mathcal{R}$, and whose size is polynomial in the size of $\mathcal{G}$ and $\mathcal{R}$.*

Proof. The construction is similar as the one in the proof of Proposition 2. □

3 Controlled Term Rewriting

3.1 Monotonic CntTRS

The result of Proposition 5 can be extended from uncontrolled to controlled monotonic TRS. Intuitively, monotonic TRS are powerful enough to be able to simulate a control with uncontrolled rewrite rules.

Theorem 1. *Given a CSTG $\mathcal{G}$ and a monotonic CntTRS $\mathcal{R}$, one can construct in PTIME a CSTG generating the closure of $L(\mathcal{G})$ by $\mathcal{R}$, and whose size is linear in the size of $\mathcal{G}$ and $\mathcal{R}$.*

Proof. (sketch) In order to prove this theorem, we show how to construct a CSTG $\mathcal{G}_*$ that accepts the set of terms reachable by $\mathcal{R}$ from the terms in $L(\mathcal{G})$. The production rules P_* of $\mathcal{G}_*$ consists in two sets: the rules P of $\mathcal{G}$ and P_1, that simulate rewriting by $\mathcal{R}$. The basic idea for the construction of $\mathcal{G}_*$ is the introduction of non-terminals of the form $\langle f, q\rangle$ where f is a symbol and q is a state of some SA.

First, we produce the term $\langle t\rangle$ where $t \in L(\mathcal{G})$ and $\langle t\rangle$ is the term obtained by replacing each symbol f by the non-terminal $\langle f\rangle$. Next, we simulate the rewriting of $\mathcal{R}$ by P_1. We simulate a transition $f(q_1, \ldots, q_n) \to q$ of a SA by some production rules in P_1 of the form $\langle f\rangle(\langle f_1, q_1\rangle(\overline{x_1}), \ldots, \langle f_n, q_n\rangle(\overline{x_n})) \to \langle f, q\rangle(\langle f_1, q_1\rangle(\overline{x_1}), \ldots, \langle f_n, q_n\rangle(\overline{x_n}))$. Finally, if a final state occurs at the root position of a term and a rewrite rule matches the subterm where a selection state appears, then we rewrite the term. □

Example 4. Consider the CntTRS $\mathcal{R}$ in Example 1 and the CSG $\mathcal{G}$ such that $L(\mathcal{G}) = \{f(a, b)\}$. We construct the CSG $\mathcal{G}_*$ such that $L(\mathcal{G}_*) = \{f(a, b), f(c, b), g(c, b), g(c, c)\}$. Let the set of production rules P of $\mathcal{G}$ be $\{S \to \langle f\rangle(\langle a\rangle, \langle b\rangle), \langle a\rangle \to a, \langle b\rangle \to b, \langle f\rangle(x_1, x_2) \to f(x_1, x_2)\}$, and mark i to each component of SA to distinguish each SA $\mathcal{A}_i$. Let the axiom of $\mathcal{G}_*$ be S^λ. We define the set of production rules P_* of $\mathcal{G}_*$ as $P_* = P \cup P' \cup P_\mathcal{A} \cup P_{\mathsf{fin}} \cup P_\mathcal{R} \cup P_{\mathsf{re}}$ where

$$
\begin{aligned}
P' &= \{S^\lambda \to \langle f\rangle^\lambda(\langle a\rangle, \langle b\rangle), \langle f\rangle^\lambda(x_1, x_2) \to f(x_1, x_2)\}\\
P_\mathcal{A} &= \{\langle c_1\rangle \to \langle c_1, q^i\rangle \mid c_1 \to q^i \in \Delta_i \text{ for some } i\} \cup\\
&\quad \left\{ \begin{array}{l} \langle f_1\rangle^\lambda(\langle c_1, q_1^i\rangle, \langle c_2, q_2^i\rangle)\\ \to \langle f_1, q^i\rangle^\lambda(\langle c_1, q_1^i\rangle, \langle c_2, q_2^i\rangle) \end{array} \middle| \begin{array}{l} c_1, c_2 \in \{a, b, c\},\\ f_1(q_1^i, q_2^i) \to q^i \in \Delta_i \text{ for some } i \end{array} \right\}\\
P_{\mathsf{fin}} &= \{\langle f_1, q^i\rangle^\lambda(x_1, x_2) \to \langle f_1, q^i\rangle^\lambda_{\mathsf{fin}}(x_1, x_2) \mid f_1 \in \{f, g\}, q^i \in F_i \text{ for some } i\} \cup\\
&\quad \left\{ \begin{array}{l} \langle f_1, q^i\rangle^\lambda_{\mathsf{fin}}(\langle c_1, q_1^i\rangle, \langle c_2, q_2^i\rangle)\\ \to \langle f_1\rangle^\lambda(\langle c_1, q_1^i\rangle_{\mathsf{fin}}, \langle c_2, q_2^i\rangle_{\mathsf{fin}}) \end{array} \middle| \begin{array}{l} f_1 \in \{f, g\},\\ c_1 \in \{a, b, c\},\\ f(q_1^i, q_2^i) \to q \in \Delta_i \text{ for some } i \end{array} \right\}\\
P_\mathcal{R} &= \{\langle a, q_1^1\rangle_{\mathsf{fin}} \to \langle c\rangle\} \cup \{\langle b, q_2^2\rangle_{\mathsf{fin}} \to \langle c\rangle\} \cup \{\langle f, q_f^3\rangle^\lambda_{\mathsf{fin}}(x_1, x_2) \to \langle g\rangle^\lambda(x_1, x_2)\}\\
P_{\mathsf{re}} &= \{\langle c_1, q^i\rangle \to \langle c_1\rangle, \langle c_1, q^i\rangle_{\mathsf{fin}} \to \langle c_1\rangle \mid c_1 \in \{a, b, c\}, q^i \in Q_i \text{ for some } i\} \cup\\
&\quad \left\{ \begin{array}{l} \langle f_1, q^i\rangle^\lambda(x_1, x_2) \to \langle f_1\rangle^\lambda,\\ \langle f_1, q^i\rangle^\lambda_{\mathsf{fin}}(x_1, x_2) \to \langle f_1\rangle^\lambda(x_1, x_2) \end{array} \middle| \begin{array}{l} f_1 \in \{f, g\},\\ q^i \in Q_i \text{ for some } i \end{array} \right\}
\end{aligned}
$$

The term $f(c, b)$ is produced by $\mathcal{G}_*$ with the production $S^\lambda \xrightarrow[P']{} \langle f\rangle^\lambda(\langle a\rangle, \langle b\rangle) \xrightarrow[P_\mathcal{A}]{*} \langle f, q_f\rangle^\lambda(\langle a, q_1\rangle, \langle b, q_2\rangle) \xrightarrow[P_{\mathsf{fin}}]{*} \langle f\rangle^\lambda(\langle a, q_1\rangle_{\mathsf{fin}}, \langle b, q_2\rangle_{\mathsf{fin}}) \xrightarrow[P_\mathcal{R}]{} \langle f\rangle^\lambda(\langle c\rangle, \langle b, q_2\rangle_{\mathsf{fin}}) \xrightarrow[P_{\mathsf{re}}]{} \langle f\rangle^\lambda(\langle c\rangle, \langle b\rangle) \xrightarrow[P \cup P']{*} f(c, b)$. □

Corollary 2. *Reachability is PSPACE-complete for monotonic CntTRS.*

From Proposition 4, it immediately holds that regular model checking is undecidable for monotonic CntTRS. Moreover, the following lower bounds already hold in the very restricted case of controlled rewrite rules over words, and where each side of every rule has depth exactly one.

Proposition 6. *Reachability is NLINSPACE-complete and regular model checking is undecidable for monotonic and flat CntTRS over unary signatures.*

Proof. We reduce the acceptance (for reachability) and emptiness (for regular model checking) problems for a linear bounded automaton (LBA) $\mathcal{M}$ [20]. Let us present below the general idea of the reduction. Every configuration of $\mathcal{M}$ will be represented by a term of the form $\|{:}\, a_1 \dots a_{j-1} a_j^p a_{j+1} \dots a_n {:}\|$, where $\|{:}$, ${:}\|$ are the symbols for left and right endmarkers, $a_1 \dots a_n$ is the content of the tape of $\mathcal{M}$, p is its current state and a_j^p marks the current position (j) of the head.

To every transition θ of $\mathcal{M}$ stating that in state p, reading a, $\mathcal{M}$ changes state to p', write b and moves left, we associate the four following monotonic and flat controlled rules (Γ is the input alphabet of $\mathcal{M}$, the selected position in the regular expression is obvious, it is the *lhs* of the rule)

$$\begin{array}{llll} \|{:}\, \Gamma^* c\, a^p \Gamma^* {:}\| & : a^p(x) \to \langle a^p, \theta \rangle(x), & \|{:}\, \Gamma^* \langle c, \theta \rangle \langle a^p, \theta \rangle \Gamma^* {:}\| & : \langle a^p, \theta \rangle(x) \to b(x), \\ \|{:}\, \Gamma^* c \langle a^p, \theta \rangle \Gamma^* {:}\| & : c(x) \to \langle c, \theta \rangle(x), & \|{:}\, \Gamma^* \langle c, \theta \rangle\, b \Gamma^* {:}\| & : \langle c, \theta \rangle(x) \to c^{p'}(x). \end{array}$$

The rules for a transition moving to the right are similar. □

Some remarks about the above result. In the above construction, the selection of the rewrite position by the SA is not necessary. Only the selection of the rewritable terms by TA is needed (a weaker condition). Note also that linear and flat TRS preserve regularity, with a PTIME construction of the TA recognizing the closure (see *e.g.* [27]). Hence reachability is decidable in PTIME in the uncontrolled case.

The *conditional grammars* of [9] can be redefined in our settings as (word) grammars whose production rules are CF controlled rewrite rules (and derivations are defined using the controlled rewrite relation). It is shown in [9] that the class of languages of conditional grammars without collapsing rules coincide with CS (word) languages. Hence, it also holds that reachability is PSPACE-complete and regular model checking is undecidable for CF non-collapsing CntTRS over unary signatures.

3.2 Prefix Control

Some other former results in the case of words imply that the above lower bounds still hold when control is limited to prefix SA. It is shown in [25] that every CS word language can be generated by a CS grammar with production rules of the form $AB \to AC$, $A \to BC$, $A \to a$ (where A, B, C are non-terminal and a is a terminal). It follows that every CS word language is the closure of a constant symbol under a CF non-collapsing pCntTRS (over a unary signature).

Proposition 7. *For all CS tree language L over a unary signature Σ, there exists a CF non-collapsing pCntTRS $\mathcal{R}$ over $\Sigma' \supset \Sigma$ such that $L = \mathcal{R}^*(\{c\}) \cap \mathcal{T}(\Sigma)$ for some constant $c \in \Sigma'_0 \setminus \Sigma$.*

Proof. Since L is the language over unary symbols, L can be regarded as a word language. Moreover, we can easily construct a pCntTRS that has the rule $c \to S(\bot)$ where S is the start symbol of the grammar for L and inverse of every production rule. □

Corollary 3. *Reachability is PSPACE-complete and regular model checking undecidable for CF non-collapsing pCntTRS over unary signatures.*

Another consequence of Proposition 7 is that deterministic top-down SA (which are incomparable with prefix SA in general but more general than prefix SA in unary signatures) already capture CS languages, for unary signatures.

To add a final remark, we can observe that following Example 3, there is no hope of regularity preservation even for very simple CF CntTRS containing only flat and monotonic rules, and even restricting to prefix control.

3.3 Non-monotonic Rewrite Rules

It is also shown in [9] that the class of languages of conditional grammars with collapsing rules coincide with recursively enumerable languages. As a consequence, reachability is undecidable for CF CntTRS (with collapsing rules) already in the case of unary signatures. Actually, the following propositions shows that flat (but not monotonic) controlled rewrite rules are sufficient for the simulation of Turing machines.

Proposition 8. *Reachability is undecidable for flat CntTRS over unary signatures.*

Proof. Flat CntTRS is a super class of monotonic and flat CntTRS. We can extend the flat CntTRS $\mathcal{R}$ that simulates the moves of LBA in the proof for Proposition 6, by adding some flat rules of the form $:\| \to \flat:\|$ and $\flat:\| \to :\|$ to $\mathcal{R}$ ($:\|$ denotes the right endmarker), in order to simulate all the moves of a TM. □

Note that when the signature is unary, all the rewrite rules are necessary linear. Again, this result is in contrast with the case of uncontrolled rewriting, because linear and flat TRS preserve regularity, and hence have a decidable reachability problem. Restricting the control to prefix permits to obtain a decidability result for non-monotonic rewrite rules, as long as they are not collapsing.

Theorem 2. *Reachability is decidable in PSPACE for flat non-collapsing pCntTRS over unary signatures.*

Proof. We show the following claim: u rewrites to v iff $u = u_0 \xrightarrow[\mathcal{R}]{} u_1 \xrightarrow[\mathcal{R}]{} \cdots \xrightarrow[\mathcal{R}]{} u_k = v$ with $\|u_0\|, \ldots, \|u_k\| \leq max(\|u\|, \|v\|)$.

The "if" direction is trivial. For the "only if" direction, assume given a reduction $u = w_1 \xrightarrow[\mathcal{R}]{*} w_n = v$, and let $max(\|u\|, \|v\|) = M$. We make an induction on the number of strings w_i longer than M. Suppose that the reduction contains one w_i such that $||w_i|| > M$. Then there exists a sub-sequence $w_k \xrightarrow[\mathcal{R}]{+} w_m$ such that $\|w_k\| = \|w_m\| = M$, with $k < m$. It holds that $w_k = w_k[\bot]_M \xrightarrow[\mathcal{R}]{*} w_m[\bot]_M = w_m$ because we consider only prefix control. This reduces the number of string w_i longer than M. □

Non-monotonicity is also a source of undecidability of reachability for CntTRS even in the case of ground controlled rewrite rules.

Proposition 9. *Reachability is undecidable for ground CntTRS.*

Proof. By representing the words $a_1, \ldots, a_n$ as right combs $f(a_1, f(\cdots f(a_n, \bot)))$, we can construct a ground CntTRS that simulates the moves of Turing Machine. Like in the proof of Propositions 8 and 6, one move of the of the TM is simulated by several rewrite steps, controlling the context left or right of the current position of the TM's head. Here, the controlled rewrite rule will have the form $\mathcal{A} : a \to a'$ were a and a' are constant symbols, and $\mathcal{A}$ controls c and d in a configuration $f(\ldots f(c, f(a, f(d(, \ldots)))))$, where a is at the rewrite position. □

This result is in contrast to uncontrolled ground TRS, for which reachability is decidable in PTIME.

4 Recursive Prefix Control

We propose a relaxed form of control, where, in order to select the positions of application of a controlled rewrite rule, the term to be rewritten is tested for membership in the closure of a regular language L, instead of membership to L directly. The idea is somehow similar to conditional rewriting (see *e.g.* [2]) where the conditions are equations that have to be solved by the rewrite system.

The definition is restricted to control with prefix SA, and a recursive pCntTRS $\mathcal{R}$ is defined as a pCntTRS. In order to define formally the rewrite relation, let us recall first that in the computations of a prefix SA $\mathcal{A}$, the states below a selection state q_s are universal (q_0), *i.e.* that we can have any subterm at a selected position (only the part of the term *above* the selected position matters). Following this observation, we say that the variable position p in a context $C[x_1]$ over Σ is selected by the prefix SA $\mathcal{A}$ if p is selected in $C[c]$ where c is an arbitrary symbol of Σ_0. A term s rewrites to t in one step by a recursive pCntTRS $\mathcal{R}$, denoted by $s \xrightarrow[\mathcal{R}]{} t$, if there exists a controlled rewrite rule $\mathcal{A} : \ell \to r \in \mathcal{R}$, where $\mathcal{A}$ is a prefix SA, a substitution σ, a position $p \in \mathcal{P}os(s)$, and a context $C[x_1]$ such that $C[x_1] \xrightarrow[\mathcal{R}]{*} s[x_1]_p$ and the position of x_1 is selected in $C[x_1]$ by $\mathcal{A}$, such that $s|_p = \ell\sigma$ and $t = s[r\sigma]_p$. This definition is well-founded because of the restriction to prefix control (remember that the root position is always selected by prefix SA).

Example 5. Let $\Sigma = \{a, b, c, d, \bot\}$ be a unary signature, and let $\mathcal{R}$ be the flat recursive pCntTRS containing the rules $\mathcal{C}_1 : a(a(x)) \to b(x)$, and $\mathcal{C}_2 : c(x) \to d(x)$, where the SA $\mathcal{C}_1$ selects the position after a prefix aa, and $\mathcal{C}_2$ selects the position after a prefix $aaaa$. Then we have with $\mathcal{R}$ (we omit the parentheses and the tail $\bot$, and underline the part of the term which is rewritten)

$$aa\underline{aa}c \xrightarrow[\mathcal{R}]{} aab\underline{c} \xrightarrow[\mathcal{R}]{} aabd$$

Note that for the last step, we have use the fact that $aaaa \xrightarrow[\mathcal{R}]{} aab$, *i.e.* there exists $C[x_1] = aaaa(x_1)$ with $C[x_1] \xrightarrow[\mathcal{R}]{} aab(x_1)$ and the position of x_1 in $C[x_1]$ is selected by $\mathcal{C}_2$. The last rewrite step would not be possible if $\mathcal{R}$ would not be recursive, because aab is not a prefix admitted by $\mathcal{C}_2$.

Theorem 3. *Regular model-checking is decidable in EXPTIME for linear and right-shallow recursive pCntTRS.*

Proof. (sketch) We show that, given a right-shallow and linear recursive pCntTRS $\mathcal{R}$ and given the language $L \subseteq \mathcal{T}(\Sigma)$ of a TA $\mathcal{A}_L$ we can construct an alternating tree automaton with epsilon-transitions (ε-ATA) $\mathcal{A}'$ recognizing the rewrite closure $\mathcal{R}^*(L)$. Intuitively, an alternating tree automata $\mathcal{A}$ is a top-down tree automaton that can spawn in several copies during computation on a term t. Formally, an ε-ATA over a signature Σ is a tuple $\mathcal{A} = \langle Q, q_0, \delta \rangle$ where Q is a finite set of states, $q_0 \in Q$ is the initial state and δ is a function which associates to every state $q \in Q$ a disjunction of conjunctions of propositional variables of the following form $a \in \Sigma$, or $\langle q', \varepsilon \rangle$, for $q' \in Q \setminus \{q\}$, or $\langle q', i \rangle$, for $q' \in Q$ and $1 \leq i \leq m$ where m is the maximal arity of a symbol in Σ.

A *run* of $\mathcal{A}$ on $t \in \mathcal{T}(\Sigma)$ is a function ρ from $\mathcal{P}os(t)$ into 2^Q such that for all position $p \in \mathcal{P}os(t)$, with $t(p) = a \in \Sigma_n$ $(n \geq 0)$, and for all state $q \in \rho(p)$, it holds that $a, \langle \rho(p.1), 1 \rangle, \ldots, \langle \rho(p.n), n \rangle, \langle \rho(p), \varepsilon \rangle \models \delta(q)$ where $\langle S, p \rangle$ is a notation for all the variables $\langle q, p \rangle$ with $q \in S$, and $\models$ denotes propositional satisfaction, while assigning true to the propositional variables on the left of $\models$.

The language $L(\mathcal{A})$ of $\mathcal{A}$ is the set of terms $t \in \mathcal{T}(\Sigma)$ on which there exists a run ρ of $\mathcal{A}$ such that $q_0 \in \rho(\varepsilon)$ (terms *recognized* by $\mathcal{A}$).

Roughly, the principle of the construction of $\mathcal{A}'$ is to start with $\mathcal{A}_L$ and the SA of $\mathcal{R}$, casted into ATA and merged, and to complete the transition functions in order to reflect the effect of the possible rewrite steps. □

Conclusion

We have proposed a definition of controlled term rewrite systems based on selection automata for the specification of authorized rewrite position, and a restriction where the selection of a position depends only on the labels on its prefix path. We have shown that reachability is PSPACE-complete for controlled monotonic (non-size-reducing) rewrite rules, using context-sensitive tree languages, and for prefix-controlled context-free non-collapsing rewrite rules. When allowing size decreasing rules, reachability becomes undecidable, even for flat and linear or ground rules. Finally, we have presented a relaxed form of prefix control called recursive prefix control which permits to obtain preservation of regular tree languages, hence decidability of reachability and regular model checking (in EXPTIME). The proof involves the construction of alternating tree automata with ε-transitions, and we believe that this technique could be useful for computing the rewrite closure of other classes of automata with local constraints.

The proof of undecidability for ground CntTRS (Proposition 9) does not work when restricting to prefix control. It could be interesting to know whether reachability is decidable for ground pCntTRS. Also, we are interested in knowing how the decidability result of Theorem 2 can be generalized to non-unary signatures.

In [9], the conditional grammars (*i.e.* controlled (in the above sense) context-free word grammars) are related to grammars with a restriction on the possible production sequences (the list of names of production rules used must belong

to a regular language). It could be interesting to establish a similar comparison for term rewriting. In particular, results on the restriction defined by authorized sequences of rewrite rules could be useful for the analysis of languages for the extensional specification of the set of possible rewrite derivations like in [3].

Rewriting of unranked ordered labeled tree has been much less studied that its counterpart for ranked terms. We would like to study controlled rewriting in this case, in particular in the context of update rules for XML [6,17].

Acknowledgements. The authors wish to thanks Olivier Ly for his suggestion for the proof of Proposition 9, and Sylvain Schmitz, Géraud Senizergues and Hubert Comon-Lundh for their useful remarks and recommendations.

References

1. Abdulla, P.A., Jonsson, B., Mahata, P., d'Orso, J.: Regular tree model checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 555–568. Springer, Heidelberg (2002)
2. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press, New York (1998)
3. Borovansky, P., Kirchner, H., Kirchner, C., Ringeissen, C.: Rewriting with strategies in elan: a functional semantics. International Journal of Foundations of Computer Science (IJFCS) 12(1), 69–95 (2001)
4. Bouajjani, A., Habermehl, P., Rogalewicz, A., Vojnar, T.: Abstract regular tree model checking of complex dynamic data structures. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 52–70. Springer, Heidelberg (2006)
5. Chabin, J., Réty, P.: Visibly pushdown languages and term rewriting. In: Konev, B., Wolter, F. (eds.) FroCos 2007. LNCS (LNAI), vol. 4720, pp. 252–266. Springer, Heidelberg (2007)
6. Chamberlin, D., Robie, J.: Xquery update facility 1.0. W3C Candidate Recommendation (2009), http://www.w3.org/TR/xquery-update-10/
7. Comon, H.: Sequentiality, second order monadic logic and tree automata. In: Proc. 10th IEEE Symposium on Logic in Computer Science (LICS), pp. 508–517. IEEE Computer Society, Los Alamitos (1995)
8. Comon, H., Dauchet, M., Gilleron, R., Jacquemard, F., Löding, C., Lugiez, D., Tison, S., Tommasi, M.: Tree Automata Techniques and Applications (2007), http://tata.gforge.inria.fr
9. Dassow, J., Paun, G., Salomaa, A.: Grammars with Controlled Derivations. In: Handbook of Formal Languages, vol. 2, pp. 101–154. Springer, Heidelberg (1997)
10. Durand, I., Middeldorp, A.: Decidable call-by-need computations in term rewriting. Information and Computation 196(2), 95–126 (2005)
11. Engelfriet, J.: Top-down tree transducers with regular look-ahead. Mathematical Systems Theory 10, 289–303 (1977)
12. Feuillade, G., Genet, T., Tong, V.V.T.: Reachability analysis over term rewriting systems. J. Autom. Reasoning 33(3-4), 341–383 (2004)
13. Frick, M., Grohe, M., Koch, C.: Query evaluation on compressed trees. In: Proc. of the 18th Annual IEEE Symposium on Logic in Computer Science. IEEE Computer Society, Los Alamitos (2003)

14. Gottlob, G., Koch, C.: Monadic datalog and the expressive power of languages for web information extraction. Journal of the ACM 51(1), 74–113 (2004)
15. Guessarian, I.: Pushdown tree automata. Mathematical Systems Theory 16(1), 237–263 (1983)
16. Hofbauer, D., Waldmann, J.: Deleting string rewriting systems preserve regularity. Theor. Comput. Sci. 327(3), 301–317 (2004)
17. Jacquemard, F., Rusinowitch, M.: Rewrite-based verification of XML updates. In: Proc. 12th ACM SIGPLAN International Symposium on Principles and Practice of Declarative Programming (PPDP), pp. 119–130. ACM, New York (2010)
18. Jones, N.D., Andersen, N.: Flow analysis of lazy higher-order functional programs. Theoretical Computer Science 375(1-3), 120–136 (2007)
19. Kojima, Y., Sakai, M.: Innermost reachability and context sensitive reachability properties are decidable for linear right-shallow term rewriting systems. In: Voronkov, A. (ed.) RTA 2008. LNCS, vol. 5117, pp. 187–201. Springer, Heidelberg (2008)
20. Kuroda, S.Y.: Classes of languages and linear-bounded automata. Information and Control 7, 207–223 (1964)
21. Lucas, S.: Context-sensitive rewriting strategies. Inf. Comput. 178, 294–343 (2002)
22. Milo, T., Suciu, D., Vianu, V.: Typechecking for XML transformers. J. of Comp. Syst. Sci. 66(1), 66–97 (2003)
23. Nagaya, T., Toyama, Y.: Decidability for left-linear growing term rewriting systems. Inf. Comput. 178(2), 499–514 (2002)
24. Neven, F., Schwentick, T.: Query automata over finite trees. Theoretical Computer Science 275(1-2), 633–674 (2002)
25. Penttonen, M.: One-sided and two-sided context in formal grammars. Information and Control 25, 371–392 (1974)
26. Réty, P., Vuotto, J.: Tree automata for rewrite strategies. J. Symb. Comput. 40, 749–794 (2005)
27. Salomaa, K.: Deterministic tree pushdown automata and monadic tree rewriting systems. J. Comput. Syst. Sci. 37(3), 367–394 (1988)
28. Sénizergues, G.: Formal languages and word-rewriting. In: Comon, H., Jouannaud, J.-P. (eds.) TCS School 1993. LNCS, vol. 909, pp. 75–94. Springer, Heidelberg (1995)

Sharing Is Caring: Combination of Theories*

Dejan Jovanović and Clark Barrett

New York University

Abstract. One of the main shortcomings of the traditional methods for combining theories is the complexity of guessing the arrangement of the variables shared by the individual theories. This paper presents a reformulation of the Nelson-Oppen method that takes into account explicit equality propagation and can ignore pairs of shared variables that the theories do not care about. We show the correctness of the new approach and present care functions for the theory of uninterpreted functions and the theory of arrays. The effectiveness of the new method is illustrated by experimental results demonstrating a dramatic performance improvement on benchmarks combining arrays and bit-vectors.

1 Introduction

The seminal paper of Nelson and Oppen [15] introduced a general framework for combining quantifier-free first-order theories in a modular fashion. Using the Nelson-Oppen framework, decision procedures for two individual theories can be used as black boxes to create a decision procedure for the combined theory. Although the Nelson-Oppen combination method as originally formulated requires stably-infinite theories, it can be extended to handle non-stably-infinite theories using an approach based on polite theories [12,13,18].

The core idea driving the method (and ensuring its correctness) is the exchange of equalities and disequalities over the interface variables between the theories involved in the combination. Interface variables are the problem variables that are shared by both theories (or an extended set of variables in the polite combination framework), and both theories must agree on an arrangement over these variables. Most modern satisfiability modulo theories (SMT) solvers perform the search for such an arrangement by first using aggresive theory propagation to determine as much of the arrangement as possible and then relying on an efficient SAT solver to guess the rest of the arrangement, backtracking and learning lemmas as necessary [1,3,6].

In some cases, if the theories that are being combined have additional properties, such as convexity and/or complete and efficient equality propagation, there are more efficient ways of obtaining a suitable arrangement. But, in general, since the number of shared variables can be substantial, guessing an arrangement over the shared variables can have an exponential impact on the running time [16]. Trying to minimize the burden of non-deterministic guessing is thus

* This work was funded in part by SRC contract 2008-TJ-1850.

C. Tinelli and V. Sofronie-Stokkermans (Eds.): FroCoS 2011, LNAI 6989, pp. 195–210, 2011.

of the utmost importance for a practical and efficient combination mechanism. For example, a recent model-based theory combination approach [7], in which the solver keeps a model for each theory, takes the optimistic stance of eagerly propagating all equalities that hold in the model (whether or not they are truly implied), obtaining impressive performance improvements.

In this paper we tackle the problem of minimizing the amount of non-deterministic guessing by equipping the theories with an *equality propagator* and a *care function*. The role of the theory-specific equality propagator is, given a context, to propagate entailed equalities and disequalities over the interface variables. The care function, on the other hand, provides information about which variable pairs among the interface variables are important for maintaining the satisfiability of a given formula. With the information provided by these two functions we can, in many cases, drastically reduce the search space for finding a suitable arrangement. We present a reformulation of the Nelson-Oppen method that uses these two functions to decide a combination of two theories. The method can easily be adapted to the combination method for polite theories, where reducing the number of shared variables is even more important (the polite theory combination method requires extending the set of interface variables significantly).

2 Preliminaries

We start with a brief overview of the syntax and semantics of many-sorted first-order logic. For a more detailed exposition, we refer the reader to [11,21].

A *signature* Σ is a triple (S, F, P) where S is a set of *sorts*, F is a set of *function symbols*, and P is a set of *predicate symbols*. For a signature $\Sigma = (S, F, P)$, we write $\Sigma^{\mathbb{S}}$ for the set S of sorts, $\Sigma^{\mathbb{F}}$ for the set F of function symbols, and $\Sigma^{\mathbb{P}}$ for the set P of predicates. Each predicate and function symbol is associated with an *arity*, a tuple constructed from the sorts in S. Functions whose arity is a single sort are called *constants*. We write $\Sigma_1 \cup \Sigma_2 = (S_1 \cup S_2, F_1 \cup F_2, P_1 \cup P_2)$ for the union[1] of signatures $\Sigma_1 = (S_1, F_1, P_1)$ and $\Sigma_2 = (S_2, F_2, P_2)$. Additionally, we write $\Sigma_1 \subseteq \Sigma_2$ if $S_1 \subseteq S_2$, $F_1 \subseteq F_2$, $P_1 \subseteq P_2$, and the symbols of Σ_1 have the same arity as those in Σ_2. We assume the standard notions of a Σ-*term*, Σ-*literal*, and Σ-*formula*. In the following, we assume that all formulas are quantifier-free, if not explicitly stated otherwise. A literal is called *flat* if it is of the form $x = y$, $x \neq y$, $x = f(y_1, \ldots, y_n)$, $p(y_1, \ldots, y_n)$, or $\neg p(y_1, \ldots, y_n)$, where $x, y, y_1, \ldots, y_n$ are variables, f is a function symbol, and p is a predicate symbol. If ϕ is a term or a formula, we will denote by $vars_\sigma(\phi)$ the set of variables of sort σ that occur (free) in ϕ. We overload this function in the usual way, $vars_S(\phi)$ denoting variables in ϕ of the sorts in S, and $vars(\phi)$ denoting all variables in ϕ. We also sometimes refer to a set Φ of formulas as if it were a single formula, in

[1] In this paper, we always assume that function and predicate symbols from different theories do not overlap, so that the union operation is well-defined. On the other hand, two different theories are allowed to have non-disjoint sets of sorts.

which case the intended meaning is the conjunction $\bigwedge \Phi$ of the formulas in the set.

Let Σ be a signature, and let X be a set of variables whose sorts are in $\Sigma^{\mathbb{S}}$. A *Σ-interpretation* $\mathcal{A}$ over X is a map that interprets each sort $\sigma \in \Sigma^{\mathbb{S}}$ as a non-empty domain A_σ, each variable $x \in X$ of sort σ as an element $x^{\mathcal{A}} \in A_\sigma$, each function symbol $f \in \Sigma^{\mathbb{F}}$ of arity $\sigma_1 \times \cdots \times \sigma_n \times \tau$ as a function $f^{\mathcal{A}} : A_{\sigma_1} \times \cdots \times A_{\sigma_n} \to A_\tau$, and each predicate symbol $p \in \Sigma^{\mathbb{P}}$ of arity $\sigma_1 \times \cdots \times \sigma_n$ as a subset $p^{\mathcal{A}}$ of $A_{\sigma_1} \times \cdots \times A_{\sigma_n}$. A *$\Sigma$-structure* is a Σ-interpretation over an empty set of variables. As usual, the interpretations of terms and formulas in an interpretation $\mathcal{A}$ are defined inductively over their structure. For a term t, we denote with $t^{\mathcal{A}}$ the evaluation of t under the interpretation $\mathcal{A}$. Likewise, for a formula ϕ, we denote with $\phi^{\mathcal{A}}$ the truth-value (true or false) of ϕ under interpretation $\mathcal{A}$. A Σ-formula ϕ is *satisfiable* iff it evaluates to true in some Σ-interpretation over (at least) $vars(\phi)$. Let $\mathcal{A}$ be an Ω-interpretation over some set V of variables. For a signature $\Sigma \subseteq \Omega$, and a set of variables $U \subseteq V$, we denote with $\mathcal{A}^{\Sigma,U}$ the interpretation obtained from $\mathcal{A}$ by restricting it to interpret only the symbols in Σ and the variables in U.

We will use the definition of theories as classes of structures, rather than sets of sentences. We define a theory formally as follows (see e.g. [20] and Definition 2 in [18]).

Definition 1 (Theory). *Given a set of Σ-sentences* $\mathbf{Ax}$ *a* Σ-theory $T_{\mathbf{Ax}}$ *is a pair* $(\Sigma, \mathbf{A})$ *where Σ is a signature and* $\mathbf{A}$ *is the class of Σ-structures that satisfy* $\mathbf{Ax}$.

Given a theory $T = (\Sigma, \mathbf{A})$, a *T-interpretation* is a Σ-interpretation $\mathcal{A}$ such that $\mathcal{A}^{\Sigma,\emptyset} \in \mathbf{A}$. A Σ-formula ϕ is T-satisfiable iff it is satisfiable in some T-interpretation $\mathcal{A}$. This is denoted as $\mathcal{A} \vDash_T \phi$, or just $\mathcal{A} \vDash \phi$ if the theory is clear from the context.

As theories in our formalism are represented by classes of structures, a combination of two theories is represented by those structures that can interpret both theories (Definition 3 in [18]).

Definition 2 (Combination). *Let* $T_1 = (\Sigma_1, \mathbf{A}_1)$ *and* $T_2 = (\Sigma_2, \mathbf{A}_2)$ *be two theories. The* combination *of* T_1 *and* T_2 *is the theory* $T_1 \oplus T_2 = (\Sigma, \mathbf{A})$ *where* $\Sigma = \Sigma_1 \cup \Sigma_2$ *and* $\mathbf{A} = \{\Sigma\text{-structures } \mathcal{A} \mid \mathcal{A}^{\Sigma_1,\emptyset} \in \mathbf{A_1} \text{ and } \mathcal{A}^{\Sigma_2,\emptyset} \in \mathbf{A_2}\}$.

The set of Σ-structures resulting from the combination of two theories is indeed a theory in the sense of Definition 1. If $\mathbf{Ax}_1$ is the set of sentences defining theory T_1, and $\mathbf{Ax}_2$ is the set of sentences defining theory T_2, then $\mathbf{A}$ is the set of Σ-structures that satisfy the set $\mathbf{Ax} = \mathbf{Ax}_1 \cup \mathbf{Ax}_2$ (see Proposition 4 in [18]).

Given decision procedures for the satisfiability of formulas in theories T_1 and T_2, we are interested in constructing a decision procedure for satisfiability in $T_1 \oplus T_2$ using these procedures as black boxes. The Nelson-Oppen combination method [15,20,21] gives a general mechanism for doing this. Given a formula ϕ over the combined signature $\Sigma_1 \cup \Sigma_2$, the first step is to *purify* ϕ by constructing an equisatisfiable set of formulas $\phi_1 \cup \phi_2$ such that each ϕ_i consists of only Σ_i-formulas. This can easily be done by finding a pure (i.e. Σ_i- for some i) subterm t,

replacing it with a new variable v, adding the equation $v = t$, and then repeating this process until all formulas are pure. The next step is to force the decision procedures for the individual theories to agree on whether variables appearing in both ϕ_1 and ϕ_2 (called *shared* or *interface* variables) are equal. This is done by introducing an *arrangement* over the shared variables [18,20].

Here we will use a more general definition of an arrangement that allows us to restrict the pairs of variables that we are interested in. We do so by introducing the notion of a *care graph*. Given a set of variables V, we will call any graph $\mathbf{G} = \langle V, E \rangle$ a care graph, where $E \subseteq V \times V$ is the set of care graph edges. If an edge $(x, y) \in E$ is present in the care graph, it means that we are interested in the relationship between the variables x and y.

Definition 3 (Arrangement). *Given a care graph $\mathbf{G} = \langle V, E \rangle$ where sorts of variables in V range over a set of sorts S, with $V_\sigma = vars_\sigma(V)$, we call $\delta_\mathbf{G}$ an* arrangement over $\mathbf{G}$ *if there exists a family of equivalence relations*

$$\mathcal{E} = \{\ E_\sigma \subseteq V_\sigma \times V_\sigma \ \mid\ \sigma \in S\ \}\ ,$$

such that the equivalence relations restricted to E induce $\delta_\mathbf{G}$, i.e. $\delta_\mathbf{G} = \bigcup_{\sigma \in S} \delta_\sigma$, where each δ_σ is an individual arrangement of V_σ (restricted to E):

$$\delta_\sigma = \{\ x = y \ \mid\ (x, y) \in E_\sigma \cap E\ \} \cup \{\ x \neq y \ \mid\ (x, y) \in \overline{E}_\sigma \cap E\ \}\ ,$$

where $\overline{E}_\sigma$ denotes the complement of E_σ (i.e. $V_\sigma \times V_\sigma \setminus E_\sigma$). If the care graph $\mathbf{G}$ is a complete graph over V, we will denote the arrangement simply as δ_V.

The Nelson-Oppen combination theorem states that ϕ is satisfiable in $T_1 \oplus T_2$ iff there exists an arrangement δ_V of the shared variables $V = vars(\phi_1) \cap vars(\phi_2)$ such that $\phi_i \cup \delta_V$ is satisfiable in T_i. However, as mentioned earlier, some restrictions on the theories are necessary in order for the method to be complete. Sufficient conditions for completeness are: the two signatures have no function or predicate symbols in common; and the two theories are *stably-infinite* over (at least) the set of common sorts $\Sigma_1^\mathbb{S} \cap \Sigma_2^\mathbb{S}$. Stable-infiniteness was originally introduced in a single-sorted setting [16]. In the many-sorted setting stable-infiniteness is defined with respect to a subset of the signature sorts (see Definition 6 from [21]).

3 New Combination Method

In this section we present a new method for combining two signature-disjoint theories. The method is based on Nelson-Oppen, but it makes equality propagation explicit and also includes a *care function* for each theory, enabling a more efficient mechanism for determining equalities and dis-equalities among the shared variables. Another notable difference from the original method is that we depart from viewing the combination problem as symmetric. Instead, as in the method for combining polite theories [12,13,18], one of the theories is

designated to take the lead in selecting which variable pairs are going to be part of the final arrangement.

We first define the equality propagator and the care function, and then proceed to presenting and proving the combination method.

Definition 4 (Equality Propagator). *For a Σ-theory T we call a function $\mathfrak{P}_T^{=}[\![\cdot]\!]$ an* equality propagator *for T if, for every set V of variables, it maps every set ϕ of flat Σ-literals into a set of equalities and dis-equalities between variables:*

$$\mathfrak{P}_T^{=}[\![V]\!](\phi) = \{x_1 = y_1, \ldots, x_m = y_m\} \cup \{z_1 \neq w_1, \ldots, z_n \neq w_n\} \ ,$$

where $vars(\mathfrak{P}_T^{=}[\![V]\!](\phi)) \subseteq V$ and

1. *for each equality $x_i = y_i \in \mathfrak{P}_T^{=}[\![V]\!](\phi)$ it holds that $\phi \vDash_T x_i = y_i$;*
2. *for each dis-equality $z_i \neq w_i \in \mathfrak{P}_T^{=}[\![V]\!](\phi)$ it holds that $\phi \vDash_T z_i \neq w_i$;*
3. *$\mathfrak{P}_T^{=}[\![V]\!]$ is monotone, i.e. $\phi \subseteq \psi \implies \mathfrak{P}_T^{=}[\![V]\!](\phi) \subseteq \mathfrak{P}_T^{=}[\![V]\!](\psi)$; and*
4. *$\mathfrak{P}_T^{=}[\![V]\!]$ contains at least those equalities and dis-equalities, over variables in V, that appear in ϕ.*

An equality propagator, given a set of theory literals, returns a set of entailed equalities and dis-equalities between the variables in V. It does not need to be complete (i.e. it does not need to return *all* entailed equalities and dis-equalities), but the more complete it is, the more helpful it is in reducing the arrangement search space.

When combining two theories, the combined theory can provide more equality propagation than just the union of the individual propagators. The following construction defines an equality propagator that reuses the individual propagators in order to obtain a propagator for the combined theory. This is achieved by allowing the propagators to incrementally exchange literals until a fix-point is reached.

Definition 5 (Combined Propagator). *Let T_1 and T_2 be two theories over the signatures Σ_1 and Σ_2, equipped with equality propagators $\mathfrak{P}_{T_1}^{=}[\![\cdot]\!]$ and $\mathfrak{P}_{T_2}^{=}[\![\cdot]\!]$, respectively. Let $T = T_1 \oplus T_2$ and $\Sigma = \Sigma_1 \cup \Sigma_2$. Let V be a set of variables and ϕ a set of flat Σ-literals partitioned into a set ϕ_1 of Σ_1-literals and a set ϕ_2 of Σ_2-literals. We define the combined propagator $\mathfrak{P}_T^{=}[\![\cdot]\!]$ for the theory T as*

$$\mathfrak{P}_T^{=}[\![V]\!](\phi) = (\mathfrak{P}_{T_1}^{=} \oplus \mathfrak{P}_{T_2}^{=})[\![V]\!](\phi) = \psi_1^* \cup \psi_2^* \ ,$$

where $\langle \psi_1^*, \psi_2^* \rangle$ is the least fix-point of the following operator $\mathcal{F}$

$$\mathcal{F}\langle \psi_1, \psi_2 \rangle = \left\langle \mathfrak{P}_{T_1}^{=}[\![V]\!](\phi_1 \cup \psi_2), \mathfrak{P}_{T_2}^{=}[\![V]\!](\phi_2 \cup \psi_1) \right\rangle \ .$$

The fix-point exists as the propagators are monotone and the set V is finite. Moreover, the value of the fix-point is easily computable by iteration from $\langle \emptyset, \emptyset \rangle$. Also, it's clear from the definition that the combined propagator is at least as strong as the individual propagators, i.e. $\mathfrak{P}_{T_1}^{=}[\![V]\!](\phi_1) \subseteq \mathfrak{P}_T^{=}[\![V]\!](\phi_1) \subseteq \mathfrak{P}_T^{=}[\![V]\!](\phi)$, $\mathfrak{P}_{T_2}^{=}[\![V]\!](\phi_2) \subseteq \mathfrak{P}_T^{=}[\![V]\!](\phi_2) \subseteq \mathfrak{P}_T^{=}[\![V]\!](\phi)$.

Definition 6 (Care Function). *For a Σ-theory T we call a function $\mathfrak{C}[\![\cdot]\!]$ a* care function *for T with respect to a T-equality propagator $\mathfrak{P}^{=}_{T}[\![\cdot]\!]$ when for every set V of variables and every set ϕ of flat Σ-literals*

1. *$\mathfrak{C}[\![V]\!]$ maps ϕ to a care graph $\mathbf{G} = \langle V, E\rangle$;*
2. *if $x = y$ or $x \neq y$ are in $\mathfrak{P}^{=}_{T}[\![V]\!](\phi)$ then $(x, y) \notin E$;*
3. *if $\mathbf{G} = \langle V, \emptyset\rangle$ and ϕ is T-satisfiable then, for any arrangement δ_V such that $\mathfrak{P}^{=}_{T}[\![V]\!](\phi) \subseteq \delta_V$, it holds that $\phi \cup \delta_V$ is also T-satisfiable.*

For any Σ-theory T and a set of variables V, the *trivial care function* $\mathfrak{C}_0[\![\cdot]\!]$ is the one that maps a set of variables to a complete graph over the pairs of variables that are not yet decided. i.e.

$$\mathfrak{C}_0[\![V]\!](\phi) = \langle V, \{(x, y) \in V \times V \mid x = y, x \neq y \notin \mathfrak{P}^{=}_{T}[\![V]\!](\phi)\}\rangle \quad .$$

Notice that $\mathfrak{C}_0[\![\cdot]\!]$ trivially satisfies the conditions of Definition 6 with respect to any equality propagator. To see this, the only case to consider is when the care graph returned has no edges and ϕ is satisfiable. But in this case, if $\mathfrak{P}^{=}_{T}[\![V]\!](\phi) \subseteq \delta_V$, then we must have $\mathfrak{P}^{=}_{T}[\![V]\!](\phi) = \delta_V$, and so clearly $\phi \cup \delta_V$ is satisfiable.

3.1 Combination Method

Let T_i be a Σ_i-theory, for $i = 1, 2$ and let $S = \Sigma_1^{\mathbb{S}} \cap \Sigma_2^{\mathbb{S}}$. Further, assume that each T_i is stably-infinite with respect to S_i, decidable, and equipped with a theory propagator $\mathfrak{P}^{=}_{T_i}[\![\cdot]\!]$. Additionally, let T_2 be equipped with a care function $\mathfrak{C}_{T_2}[\![\cdot]\!]$ operating with respect to $\mathfrak{P}^{=}_{T_2}[\![\cdot]\!]$. We are interested in deciding the combination theory $T = T_1 \oplus T_2$ over the signature $\Sigma = \Sigma_1 \cup \Sigma_2$. We denote the combined theory propagator with $\mathfrak{P}^{=}_{T}[\![\cdot]\!]$.The combination method takes as input a set ϕ of Σ-literals and consists of the following steps:

Purify: The output of the purification phase is two new sets of literals, ϕ_1 and ϕ_2 such that $\phi_1 \cup \phi_2$ is equisatisfiable (in T) with ϕ and each literal in ϕ_i is a flat Σ_i-literal, for $i = 1, 2$.

Arrange: Let $V = \mathit{vars}(\phi_1) \cap \mathit{vars}(\phi_2)$ be the set of all variables shared by ϕ_1 and ϕ_2. Let the care graph $\mathbf{G}_2$ be a fix-point of the following operator

$$\mathcal{G}\langle \mathbf{G}\rangle = \mathbf{G} \cup \mathfrak{C}_{T_2}[\![V]\!](\phi_2 \cup \mathfrak{P}^{=}_{T}[\![V]\!](\phi_1 \cup \phi_2 \cup \delta_{\mathbf{G}})) \quad , \tag{1}$$

where we choose the arrangement $\delta_{\mathbf{G}}$ non-deterministically.

Check: Check the following formulas for satisfiability in T_1 and T_2 respectively

$$\phi_1 \cup \mathfrak{P}^{=}_{T}[\![V]\!](\phi_1 \cup \phi_2 \cup \delta_{\mathbf{G}_2}) \quad , \qquad \phi_2 \cup \mathfrak{P}^{=}_{T}[\![V]\!](\phi_1 \cup \phi_2 \cup \delta_{\mathbf{G}_2}) \quad .$$

If both are satisfiable, output satisfiable, otherwise output unsatisfiable.

Notice that above, since the graph is finite, and the the operator $\mathcal{G}$ is increasing, the fix-point always exists. Moreover, it is in our interest to choose the minimal such fix-point, which we can obtain by doing a fix-point iteration starting from

$\mathbf{G}_0 = \langle V, \emptyset \rangle$. Another important fact is that for any fix-point $\mathbf{G}$, with respect to the $\delta_{\mathbf{G}}$ we have chosen, of the operator $\mathcal{G}$ above, we must have that the care function from (11) returns an empty graph. This follows from the fact that the propagator must return all the equalities and dis-equalities from $\delta_{\mathbf{G}}$, by definition, and the care function then must ignore them, also by definition.

Example 1. Consider the case of combining two theories T_1 and T_2 equipped with trivial care functions and propagators $\mathfrak{P}^{=}_{T_i}[\![V]\!]$ that simply return those input literals that are either equalities or dis-equalities over variables in V. Assume that ϕ_1 and ϕ_2 are the output of the purification phase, and let V be the set of variables shared by ϕ_1 and ϕ_2. Since $\mathfrak{C}_{T_2}[\![\cdot]\!]$ is a trivial care function, we will choose a arrangement $\delta_{\mathbf{G}_2}$ over the set V of shared variables that completes the set of equalities and dis-equalities over V. Since equality propagators simply keep the input equalities and dis-equalities over V, and all other relationships between variables in V are determined by $\delta_{\mathbf{G}_2}$, the combined propagator will return a complete arrangement δ_V and we will then check $\phi_1 \cup \delta_V$ and $\phi_2 \cup \delta_V$ for satisfiability. This shows that our method can effectively simulate the standard Nelson-Oppen combination method. We now show the correctness of the method.

Theorem 1. *Let T_i be a Σ_i-theory, stably-infinite with respect to the set of sorts S_i, and equipped with equality propagators $\mathfrak{P}^{=}_{T_i}[\![\cdot]\!]$, for $i = 1, 2$. Additionally, let T_2 be equipped with a care function $\mathfrak{C}_{T_i}[\![\cdot]\!]$ operating with respect to $\mathfrak{P}^{=}_{T_2}[\![\cdot]\!]$. Let $\Sigma = \Sigma_1 \cup \Sigma_2$, $T = T_1 \oplus T_2$ and let ϕ be a set of flat Σ-literals, which can be partitioned into a set ϕ_1 of Σ_1-literals and a set ϕ_2 of Σ_2-literals, with $V = vars(\phi_1) \cap vars(\phi_2)$. If $\Sigma_1^{\mathbb{S}} \cap \Sigma_2^{\mathbb{S}} = S_1 \cap S_2$, then following are equivalent*

1. *ϕ is T-satisfiable;*
2. *there exists some care-graph $\mathbf{G}_2$, and a corresponding arrangement $\delta_{\mathbf{G}_2}$, that are fix-point solutions of (11), such that the following sets are T_1- and T_2-satisfiable respectively*

$$\phi_1 \cup \mathfrak{P}^{=}_{T}[\![V]\!](\phi_1 \cup \phi_2 \cup \delta_{\mathbf{G}_2}) \ , \qquad \phi_2 \cup \mathfrak{P}^{=}_{T}[\![V]\!](\phi_1 \cup \phi_2 \cup \delta_{\mathbf{G}_2}) \ .$$

Moreover, T is stably-infinite with respect to $S_1 \cup S_2$.

Proof. (1) $\Rightarrow$ (2) : Suppose $\phi = \phi_1 \cup \phi_2$ is T-satisfiable in a T-interpretation $\mathcal{A}$. Let δ_V be the full arrangement over V satisfied by $\mathcal{A}$. Since δ_V trivially is a fix-point of (11), $\mathcal{A}$ satsifies δ_V, and the propagator only adds formulas that are entailed, it is clear that $\mathcal{A}$ satisfies both sets of formulas, which proves one direction.

(2) $\Leftarrow$ (1) : Assume that there is a T_1-interpretation $\mathcal{A}_1$ and a T_2-interpretation $\mathcal{A}_2$ (and assume wlog that both interpret all the variables in V) such that $\mathcal{A}_1 \vDash_{T_1} \phi_1 \cup \mathfrak{P}^{=}_{T}[\![V]\!](\phi_1 \cup \phi_2 \cup \delta_{\mathbf{G}_2})$ and $\mathcal{A}_2 \vDash_{T_2} \phi_2 \cup \mathfrak{P}^{=}_{T}[\![V]\!](\phi_1 \cup \phi_2 \cup \delta_{\mathbf{G}_2})$. Let δ_V be the arrangement over the complete graph on V satisfied by $\mathcal{A}_1$, so

$$\delta_{\mathbf{G}_2} \subseteq \mathfrak{P}^{=}_{T_2}[\![V]\!](\phi_2 \cup \delta_{\mathbf{G}_2}) \subseteq \mathfrak{P}^{=}_{T}[\![V]\!](\phi_1 \cup \phi_2 \cup \delta_{\mathbf{G}_2}) \subseteq \delta_V \ .$$

Because $\mathbf{G}_2$ is a fix-point, we know that $\mathfrak{C}_{T_2}[\![V]\!](\phi_2 \cup \mathfrak{P}^{=}_{T}[\![V]\!](\phi_1 \cup \phi_2 \cup \delta_{\mathbf{G}})) = \langle V, \emptyset \rangle$. We then know, by property 3 of the care function, that there is a T_2-interpretation $\mathcal{B}_2$ such that $\mathcal{B}_2 \vDash_{T_2} \phi_2 \cup \delta_V$. Since δ_V is a complete arrangement

over all the shared variables and we also have that $\mathcal{A}_1 \models_{T_1} \phi_1 \cup \delta_V$, we can now appeal to the correctness of the standard Nelson-Oppen combination method to obtain a T-interpretation $\mathcal{C}$ that satisfies $\phi_1 \cup \phi_2 = \phi$. The proof that the combined theory is stably-infinite can be found in [12]. □

3.2 Extension to Polite Combination

The method described in Section 3 relies on the correctness argument for the standard Nelson-Oppen method, meaning that the theories involved should be stably-infinite for completeness. A more general combination method based on the notion of *polite* theories (and not requiring that both theories be stably-infinite) was introduced in [18] and clarified in [12,13]. Here, we assume familiarity with the concepts appearing in those papers, and show how they can be integrated into the combination method of this paper.

Assume that the theory T_2 is polite with respect to the set of sorts S_2 such that $\Sigma_1 \cap \Sigma_2 \subseteq S_2$, and is equipped with a witness function $witness_2$. We modify the combination method of Section 3.1 as follows:

1. In the **Arrange** and **Check** phases, instead of using ϕ_2, we use the formula produced by the witness function, i.e. $\phi_2' = witness_2(\phi_2)$.
2. We define $V = vars_S(\phi_2')$ instead of $V = vars(\phi_1) \cap vars(\phi_2)$.

Theorem 2. *Let T_i be a Σ_i-theory polite with respect to the set of sorts S_i, and equipped with equality propagators $\mathfrak{P}^{=}_{T_i}[\![\cdot]\!]$, for $i = 1, 2$. Additionally, let T_2 be equipped with a care function $\mathfrak{C}_{T_2}[\![\cdot]\!]$ operating with respect to $\mathfrak{P}^{=}_{T_2}[\![\cdot]\!]$. Let $\Sigma = \Sigma_1 \cup \Sigma_2$, $T = T_1 \oplus T_2$ and let ϕ be a set of flat Σ-literals, which can be partitioned into a set ϕ_1 of Σ_1-literals and a set ϕ_2 of Σ_2-literals. Let $\phi_2' = witness_{T_2}(\phi_2)$ and $V = vars_S(\phi_2')$. If $S \subseteq S_1 \cap S_2$, then following are equivalent*

1. *ϕ is T-satisfiable;*
2. *there exists a care-graph $\mathbf{G}_2$ and arrangement $\delta_{\mathbf{G}_2}$, fix-point solutions of (11), such that the following sets are T_1- and T_2-satisfiable respectively*

$$\phi_1 \cup \mathfrak{P}^{=}_{T}[\![V]\!](\phi_1 \cup \phi_2' \cup \delta_{\mathbf{G}_2}) \ , \qquad \phi_2' \cup \mathfrak{P}^{=}_{T}[\![V]\!](\phi_1 \cup \phi_2' \cup \delta_{\mathbf{G}_2}) \ .$$

Moreover, T is polite with respect to $S_1 \cup (S_2 \setminus \Sigma_1)$.[2]

4 Theory of Uninterpreted Functions

The theory of uninterpreted functions over a signature $\Sigma_{\mathtt{euf}}$ is the theory $T_{\mathtt{euf}} = (\Sigma_{\mathtt{euf}}, \mathbf{A})$, where $\mathbf{A}$ is simply the class of all $\Sigma_{\mathtt{euf}}$-structures. Conjunctions of literals in this theory can be decided in polynomial time by congruence closure algorithms (e.g. [19]). We make use of insights from these algorithms in defining both the equality propagator and the care function. For simplicity, we assume $\Sigma_{\mathtt{euf}}$ contains no predicate symbols, but the extension to the case with predicate symbols is straightforward.

[2] The remaining proofs are relegated to the technical report [14] due to space constraints.

Equality Propagator. Let ϕ be a set of flat literals, let V be a set of variables, and let $\sim_c$ be the smallest congruence relation[3] over terms in ϕ containing $\{(x,t) \mid x = t \in \phi\}$. We define a dis-equality relation $\neq_c$ as the smallest relation satisfying

$$x \sim_c x' \wedge y \sim_c y' \wedge x' \neq y' \in \phi \Longrightarrow x \neq_c y \ .$$

Now, we define the equality propagator as

$$\mathfrak{P}^{=}_{\mathtt{euf}}[\![V]\!](\phi) = \{x = y \mid x, y \in V, x \sim_c y\} \cup \{x \neq y \mid x, y \in V, x \neq_c y\}.$$

It is easy to see that $\mathfrak{P}^{=}_{\mathtt{euf}}[\![\cdot]\!]$ is indeed an equality propagator. Moreover, it can easily be implemented as part of a decision procedure based on congruence closure.

Example 2. Given the set $\phi = \{\ x = z, y = f(a), z \neq f(a)\ \}$, the equality propagator would return $\mathfrak{P}^{=}_{\mathtt{euf}}[\![\{\ x, y\ \}]\!](\phi) = \{\ x = x,\ y = y,\ x \neq y,\ y \neq x\ \}$.

Care Function. The definition of the care function is based on the fact that during congruence closure, we only care about equalities between pairs of variables that occur as arguments in the same position of the same function symbol. Again, let V be a set of variables and let ϕ be a set of flat literals, such that ϕ only contains function symbols from $F = \{f_1, f_2, \ldots, f_n\}$.

For a set of formulas ϕ, let $\mathcal{E}(\phi)$ denote the smallest equivalence relation over the terms occurring in ϕ containing $\{(x,t) \mid x = t \in \phi\}$. For an equivalence relation E, let E^* denote the *congruence closure* of E (i.e. the smallest congruence relation containing E). In order to make our care-function more precise, we will first approximate the implications that possible equalities over variables in V could trigger. We do so by taking all possible equalities over V, i.e. let $\delta^{=}_V$ be the full arrangement over the shared variables where all variables of the same sort are equal. Now, to see what these equalities could imply, we let $E^{=}_{\phi} = \mathcal{E}(\phi \cup \delta^{=}_V)^*$.

For each function symbol $f \in F$ of arity $\sigma_1 \times \sigma_2 \times \cdots \times \sigma_k \mapsto \sigma$, let E_f be a set containing pairs of variables that could trigger an application of congruence because of two terms that are applications of f. More precisely, let $E_f \subseteq V \times V$ be a maximal set of pairs $(x,y) \in V \times V$, that are not already decided by the propagator ($x \nsim_c y$ and $\neg x \neq_c y$), such that for each $(x,y) \in E_f$ we have:

1. there are x_i and y_i such that $x \sim_c x_i$ and $y \sim_c y_i$;
2. there are terms $f(x_1, \ldots, x_i, \ldots, x_k) \nsim_c f(y_1, \ldots, y_i, \ldots, y_k)$ in ϕ;
3. for $1 \leq j \leq k$, variables x_j and y_j could become equal, $(x_j, y_j) \in E^{=}_{\phi}$;
4. for $1 \leq j \leq k$, variables x_j and y_j are not known to be disequal, $\neg(x_j \neq_c y_j)$.

Now, we let $E = \bigcup_{f \in F} E_f$, and define the care function mapping ϕ to the care graph $\mathbf{G}$ as $\mathfrak{C}_{\mathtt{euf}}[\![V]\!](\phi) = \mathbf{G} = \langle V, E \rangle$.

[3] In this context, a congruence relation is an equivalence relation that also satisfies the congruence property: if $f(x_1, \ldots, x_n)$ and $f(y_1, \ldots, y_n)$ are terms in ϕ, and if for each $1 \leq i \leq n$, $x_i \sim_c y_i$, then $f(x_1, \ldots, x_n) \sim_c f(y_1, \ldots, y_n)$.

Example 3. Consider the following sets of literals

$$\phi_1 = \{f(x_1) \neq f(y_1), y_1 = x_2\} \ ,$$
$$\phi_2 = \{z_1 = f(x_1), z_2 = f(y_1), g(z_1, x_2) \neq g(z_2, y_2)\} \ ,$$
$$\phi_3 = \{y_1 = f(x_1), y_2 = f(x_2), z_1 = g(x_1), z_2 = g(x_2), h(y_1) \neq h(z_1)\} \ .$$

and corresponding sets of shared variables $V_1 = \{x_1, x_2\}$, $V_2 = \{x_1, x_2, y_1, y_2\}$, $V_3 = \{x_1, x_2, y_2, z_2\}$. The care function above would return the care graphs $\mathbf{G}_1 = \langle V, \{(x_1, x_2)\}\rangle$, $\mathbf{G}_2 = \langle V, \{(x_1, y_1), (x_2, y_2)\}\rangle$, and $\mathbf{G}_2 = \langle V, \{(x_1, x_2)\}\rangle$.

Note that the the care function for ϕ_3 does not return the pair (y_2, z_2), which is important in case x_1 and x_2 become equal. This is remedied in the procedure itself, by computing the fix-point, which, in case we choose $x_1 = x_2$, will add the pair (y_2, z_2) to the care graph in the second step.

Theorem 3. *Let $T_{\mathtt{euf}}$ be the theory of uninterpreted functions with equality over the signature $\Sigma_{\mathtt{euf}}$. $\mathfrak{C}_{\mathtt{euf}}[\![\cdot]\!]$ is a care function for $T_{\mathtt{euf}}$ with respect to the equality propagator $\mathfrak{P}^{=}_{\mathtt{euf}}[\![\cdot]\!]$.*

5 Theory of Arrays

The extensional theory of arrays $T_{\mathtt{arr}}$ operates over the signature $\Sigma_{\mathtt{arr}}$ that contains the sorts $\{\mathsf{array}, \mathsf{index}, \mathsf{elem}\}$ and function symbols

$$\mathsf{read} : \mathsf{array} \times \mathsf{index} \mapsto \mathsf{elem} \ , \qquad \mathsf{write} : \mathsf{array} \times \mathsf{index} \times \mathsf{elem} \mapsto \mathsf{array} \ ,$$

where read represents reading from an array at a given index, and write represents writing a given value to an array at an index. The semantics of the theory are given by the three axioms:

1. $\forall\, a{:}\mathsf{array}.\ \forall\, i{:}\mathsf{index}.\ \forall\, v{:}\mathsf{elem}.\ \mathsf{read}(\mathsf{write}(a, i, v), i) = v$,
2. $\forall\, a{:}\mathsf{array}.\ \forall\, i, j{:}\mathsf{index}.\ \forall\, v{:}\mathsf{elem}.\ i \neq j \rightarrow \mathsf{read}(\mathsf{write}(a, i, v), j) = \mathsf{read}(a, j)$,
3. $\forall\, a, b{:}\mathsf{array}.\ (\forall\, i{:}\mathsf{index}.\ \mathsf{read}(a, i) = \mathsf{read}(b, i)) \rightarrow a = b$.

The flat literals of the theory are of the form $x = \mathsf{read}(a, i)$, $a = \mathsf{write}(b, i, x)$, $i = j$, $i \neq j$, $x = y$, $x \neq y$, $a = b$, $a \neq b$, where here and below we use the convention that x, y, v are variables of sort elem, i, j are variables of sort index, a, b, and c are variables of sort array, and w, z are variables of any sort. For a set ϕ of flat $T_{\mathtt{arr}}$-literals, we also define $\alpha(\phi)$ to be the subset of ϕ that does not contain literals of the form $a = \mathsf{write}(b, i, v)$.

Decision Procedure. Before presenting the equality propagator and care function, it will be helpful to present a simple rule-based decision procedure for $T_{\mathtt{arr}}$ based on [9].[4] Given a set Γ of flat $T_{\mathtt{arr}}$-literals, we define $\approx^{\Gamma}_{a}$ as $\mathcal{E}(\alpha(\Gamma))^*$ and the corresponding disequality relation $\neq^{\Gamma}_{a}$ as the smallest relation satisfying:

[4] The main difference is that in our procedure, we exclude literals containing write from the $T_{\mathtt{euf}}$-satisfiability check as they are not needed and this allows us to have a simpler care function.

$$w \approx_a^\Gamma w' \wedge z \approx_a^\Gamma z' \wedge w \neq z \in \Gamma \implies w' \neq_a^\Gamma z' \ .$$

Additionally, let $\Gamma[l_1, \ldots, l_n]$ denote that literals $l_i, 1 \leq i \leq n$ appear in Γ, and for every pair (a, b) of variables in $vars_{\mathsf{array}}(\Gamma)$, let $k_{a,b}$ be a distinguished fresh variable of sort index. Let $\mathcal{D}_{\mathtt{arr}}$ be the following set of inference rules.

$$\mathsf{RIntro1}\ \frac{\Gamma[a = \mathsf{write}(b,i,v)]}{\Gamma, v = \mathsf{read}(a,i)}\ \text{if}\ \ v \not\approx_a^\Gamma \mathsf{read}(a,i)$$

$$\mathsf{RIntro2}\ \frac{\Gamma[a = \mathsf{write}(b,i,v), x = \mathsf{read}(c,j)]}{\Gamma, i = j \qquad \Gamma, \mathsf{read}(a,j) = \mathsf{read}(b,j)}\ \text{if}\ \begin{array}{l} a \approx_a^\Gamma c \text{ or } b \approx_a^\Gamma c, \\ i \not\approx_a^\Gamma j, \\ \mathsf{read}(a,j) \not\approx_a^\Gamma \mathsf{read}(b,j) \end{array}$$

$$\mathsf{ArrDiseq}\ \frac{\Gamma[a \neq b]}{\Gamma, \mathsf{read}(a,k_{a,b}) \neq \mathsf{read}(b,k_{a,b})}\ \text{if}\ \ \neg(\mathsf{read}(a,k_{a,b}) \neq_a^\Gamma \mathsf{read}(b,k_{a,b}))$$

Note that non-flat literals appear in the conclusions of rules RIntro2 and ArrDiseq. We use this as a shorthand for the flattened version of these literals. For example, $\mathsf{read}(a,j) = \mathsf{read}(b,j)$ is shorthand for $x = \mathsf{read}(a,j) \wedge y = \mathsf{read}(b,j) \wedge x = y$, where x and y are fresh variables (there are other possible flattenings, especially if one or more of the terms appears already in Γ, but any of them will do). We say that a set Γ of literals is $\mathcal{D}_{\mathtt{arr}}$-*saturated* if no rules from $\mathcal{D}_{\mathtt{arr}}$ can be applied.

Theorem 4. *The inference rules of $\mathcal{D}_{\mathtt{arr}}$ are sound and terminating.*

Theorem 5. *Let Γ be a $\mathcal{D}_{\mathtt{arr}}$-saturated set of flat $T_{\mathtt{arr}}$-literals. Then Γ is $T_{\mathtt{arr}}$-satisfiable iff $\alpha(\Gamma)$ is $T_{\mathtt{euf}}$-satisfiable.*

Equality Propagator. Let ϕ be a set of flat literals and V a set of variables. Consider the following modified versions of RIntro2 that are enabled only if one of the branches can be ruled out as unsatisfiable:

$$\mathsf{RIntro2a}\ \frac{\Gamma[a = \mathsf{write}(b,i,v), x = \mathsf{read}(c,j)]}{\Gamma, i = j}\ \text{if}\ \begin{array}{l} a \approx_a^\Gamma c \text{ or } b \approx_a^\Gamma c, \\ i \not\approx_a^\Gamma j, \\ \mathsf{read}(a,j) \neq_a^\Gamma \mathsf{read}(b,j) \end{array}$$

$$\mathsf{RIntro2b}\ \frac{\Gamma[a = \mathsf{write}(b,i,v), x = \mathsf{read}(c,j)]}{\Gamma, \mathsf{read}(a,j) = \mathsf{read}(b,j)}\ \text{if}\ \begin{array}{l} a \approx_a^\Gamma c \text{ or } b \approx_a^\Gamma c, \\ i \neq_a^\Gamma j, \\ \mathsf{read}(a,j) \not\approx_a^\Gamma \mathsf{read}(b,j) \end{array}$$

Let $\mathcal{D}'_{\mathtt{arr}}$ be obtained from $\mathcal{D}_{\mathtt{arr}}$ by replacing RIntro2 with the above rules. Since these rules mimic RIntro2 when they are enabled, but are enabled less often, it is clear that $\mathcal{D}'_{\mathtt{arr}}$ remains sound and terminating. Let Γ' be the result of applying $\mathcal{D}'_{\mathtt{arr}}$ until no more apply (we say that Γ' is $\mathcal{D}'_{\mathtt{arr}}$-saturated). We define the equality propagator as:

$$\mathfrak{P}^{=}_{\mathtt{arr}}[\![V]\!](\phi) = \{w = z \mid w, z \in V, w \approx_a^{\Gamma'} z\} \cup \{w \neq z \mid w, z \in V, w \neq_a^{\Gamma'} z\}.$$

It is easy to see that $\mathfrak{P}^{=}_{\mathtt{arr}}[\![\cdot]\!]$ satisfies the requirements for a propagator. Though not necessary for the care function we present here, a more powerful propagator can be obtained by additionally performing congruence closure over write terms.

Care Function. Let ϕ be a set of flat literals and V a set of variables. First, since a simple propagator cannot compute all equalities between array variables, we will ensure that the relationships between all pairs of array variables in V have been determined. To do so we define the set E_a^ϕ of pairs of array variables in V that are not yet known equal or dis-equal

$$E_a^\phi = \{(a, b) \in V \times V \mid a \not\approx_a^\phi b \wedge \neg(a \neq_a^\phi b)\} \ .$$

Next, since the inference rules can introduce new read terms, we compute the smallest set R^ϕ with possible such terms, i.e

- if $x = \mathsf{read}(a, i) \in \phi$ or $a = \mathsf{write}(b, i, v) \in \phi$, then $\mathsf{read}(a, i) \in \mathsf{R}^\phi$,
- if $a = \mathsf{write}(b, i, v) \in \phi$, $\mathsf{read}(c, j) \in \mathsf{R}^\phi$, $i \not\approx_a^\phi j$, and $a \approx_a^\phi c \vee b \approx_a^\phi c$, then both $\mathsf{read}(a, j) \in \mathsf{R}^\phi$ and $\mathsf{read}(b, j) \in \mathsf{R}^\phi$,
- if $a \neq b \in \phi$, then both $\mathsf{read}(a, k_{a,b}) \in \mathsf{R}^\phi$ and $\mathsf{read}(b, k_{a,b}) \in \mathsf{R}^\phi$.

Crucial in the introduction of the above read terms, is the set of index variables whose equality could affect the application of the RIntro2 rule. We capture these variables by defining the set E_i^ϕ as the set of all pairs (i, j) such that:

- $i \not\approx_a^\phi j$ and $\neg(i \neq_a^\phi j)$
- $\exists\, a, b, c, v.\, a = \mathsf{write}(b, i, v) \in \phi, \mathsf{read}(c, j) \in \mathsf{R}^\phi,$ and $a \approx_a^\phi c \vee b \approx_a^\phi c$.

Finally, we claim that with the variables in E_a^ϕ and E_i^ϕ decided, we can essentially use the same care function as for $T_{\mathtt{euf}}$, treating read as uninterpreted. We therefore define the third set E_r^ϕ to be the set of all pairs $(i, j) \in V \times V$ of undecided indices, $i \not\approx_a^\phi j$ and $\neg(i \neq_a^\phi j)$, such that there are a, b, i', j' with $a \approx_a^\phi b$, $i \approx_a^\phi i'$, $j \approx_a^\phi j'$, $\mathsf{read}(a, i') \in \mathsf{R}^\phi$, $\mathsf{read}(b, j') \in \mathsf{R}^\phi$, $\mathsf{read}(a, i') \not\approx_a^\phi \mathsf{read}(b, j')$.

With the definitions above, we can define the care graph as $\mathfrak{C}_{\mathtt{arr}}[\![V]\!](\phi) = \mathbf{G} = \langle V, E\rangle$, where the set of edges is defined as

$$E = \begin{cases} E_a^\phi & \text{if } E_a^\phi \neq \emptyset, \\ E_i^\phi & \text{if } E_i^\phi \neq \emptyset, \text{ and} \\ E_r^\phi & \text{otherwise.} \end{cases}$$

Note that as defined, E_i^ϕ may include pairs of index variables, one or more of which are not in V. Unfortunately, the care function fails if E_i^ϕ is not a subset of $V \times V$. We can ensure that it is either by expanding the set V until it includes all variables in E_i^ϕ or doing additional case-splitting up front on pairs in E_i^ϕ, adding formulas to ϕ, until $E_i^\phi \subseteq V \times V$.

Theorem 6. *Let $T_{\mathtt{arr}}$ be the theory of arrays. $\mathfrak{C}_{\mathtt{arr}}[\![\cdot]\!]$ is a care function for $T_{\mathtt{arr}}$ with respect to the equality propagator $\mathfrak{P}^{=}_{\mathtt{arr}}[\![\cdot]\!]$ for all sets ϕ of literals and V of variables such that $E_i^\phi \subseteq V \times V$.*

Example 4. Consider the following constraints involving arrays and bit-vectors of size m, where $\times_m$ denotes unsigned bit-vector multiplication:

$$\bigwedge_{k=1}^{n} (\mathsf{read}(a_k, i_k) = \mathsf{read}(a_{k+1}, i_{k+1}) \wedge i_k = x_k \times_m x_{k+1}) \ . \tag{2}$$

Assume that only the index variables are shared, i.e. $V = \{i_1, \ldots, i_{n+1}\}$. In this case, both E_a^ϕ and E_i^ϕ will be empty and the only read terms in R^ϕ will be those appearing in the formula. Since none of these are reading from equivalent arrays, the empty care graph is a fix-point for our care function, and we do not need to guess an arrangement.

Note that in the case when V contains array variables, the care graph requires us to split on all pairs of these variables (i.e. the care function is trivial over these variables). Fortunately, in practice it appears that index and element variables are typically shared, and only rarely are array variables shared.

6 Experimental Evaluation

We implemented the new method in the Cvc3 solver [2], and in the discussion below, we denote the new implementation as Cvc3+C. We focused our attention on the combination of the theory of arrays and the theory of fixed-size bit-vectors (QF_AUFBV). This seemed like a good place to start because there are many benchmarks which generate a non-trivial number of shared variables, and additional splits on shared bit-vector variables can be quite expensive. This allowed us to truly examine the merits of the new combination method. In order to evaluate our method against the current state-of-the-art, we compared to Boolector [4], Yices [10], Cvc3, and MathSAT [5], the top solvers in the QF_AUFBV category from the 2009 SMT-COMP competition (in order). Additionally, we included the Z3 solver [8] so as to compare to the model-based theory combination method [7]. All tests were conducted on a dedicated Intel Pentium E2220 2.4 GHz processor with 4GB of memory. Individual runs were limited to 15 minutes.

We crafted a set of new benchmarks based on Example 4 from Section 5, taking $n = 10, \ldots, 100$, with increments of 10, and $m = 32, \ldots, 128$, with increments of 32. We also included a selection of problems from the QF_AUFBV division of the SMT-LIB library. Since most of the benchmarks in the library come from model-checking of software and use a flat memory model, they mostly operate over a single array representing the heap. Our method is essentially equivalent to the standard Nelson-Oppen approach for such benchmarks, so we selected only the benchmarks that involved constraints over at least two arrays. We anticipate that such problems will become increasingly important as static-analysis tools become more precise and are able to infer separation of the heap (in the style of Burstall, e.g. [17]). All the benchmarks and the Cvc3 binaries used in the experiments are available from the authors' website.[5]

The combined results of our experiments are presented in Table 1, with columns reporting the total time (in seconds) that a solver used on the problem instances it solved (not including time spent on problem instances it was unable to solve), and the number of solved instances. Compared to Cvc3, the new implementation Cvc3+C performs uniformly better. On the first four classes of

[5] http://cs.nyu.edu/~dejan/sharing-is-caring/

Table 1. Experimental results

	Boolector		Yices		MathSAT		Z3		Cvc3		Cvc3+C	
`crafted` (40)	2100.13	40	6253.32	34	468.73	30	112.88	40	388.29	9	**14.22**	**40**
`matrix` (11)	1208.16	10	683.84	6	474.89	4	927.12	11	831.29	11	**45.08**	**11**
`unconstr` (10)	**3.00**	**10**		0	706.02	3	54.60	2	185.00	5	340.27	8
`copy` (19)	11.76	19	**1.39**	**19**	1103.13	19	4.79	19	432.72	17	44.75	19
`sort` (6)	691.06	6	557.23	4	82.21	4	248.94	3	**44.89**	**6**	**44.87**	**6**
`delete` (29)	**3407.68**	**18**	1170.93	10	2626.20	14	1504.46	10	1766.91	17	1302.32	17
`member` (24)	2807.78	24	185.54	24	217.35	24	**112.23**	**24**	355.41	24	320.80	24
	10229.57	**127**	8852.25	97	5678.53	98	2965.02	109	4004.51	89	2112.31	125

problems, Cvc3+C greatly outperforms Cvc3. On the last three classes of problems, the difference is less significant. After examining the benchmarks, we concluded that the multitude of arrays in these examples is artificial – the many array variables are just used for temporary storage of sequential updates on the same starting array – so there is not a great capacity for improvement using the care function that we described. A scatter-plot comparison of Cvc3 vs Cvc3+C is shown in Figure 1(a). Because the only difference between the two implementations is the inclusion of the method described in this paper, this graph best illustrates the performance impact this optimization can have.

When compared to the other solvers, we find that whereas Cvc3 is not particularly competitive, Cvc3+C is very competitive and in fact, for several sets of benchmarks, performs better than all of the others. This again emphasizes the strength of our results and suggests that combination methods can be of great importance for performance and scalability of modern solvers. Overall, on this set of benchmarks, Boolector solves the most (solving 2 more than Cvc3+C). However, Cvc3+C is significantly faster on the benchmarks it solves. Figure 1(b) shows a scatter-plot comparison of Cvc3+C against Boolector.

7 Conclusion

We presented a reformulation of the classic Nelson-Oppen method for combining theories. The most notable novel feature of the new method is the ability to leverage the structure of the individual problems in order to reduce the complexity of finding a common arrangement over the interface variables. We do this by defining theory-specific care functions that determine the variable pairs that are relevant in a specific problem. We proved the method correct, and presented care functions for the theories of uninterpreted functions and arrays. We draw intuition for the care functions and correctness proofs directly from the decision procedures for specific theories, leaving room for new care functions backed by better decision algorithms. Another benefit of the presented method is that it is orthogonal to the previous research on combinations of theories. For example, it would be easy to combine our method with a model-based combination approach–instead of propagating all equalities between shared variables implied by the model, one could restrict propagation to only the equalities that correspond to edges in the care graph, gaining advantages from both methods.

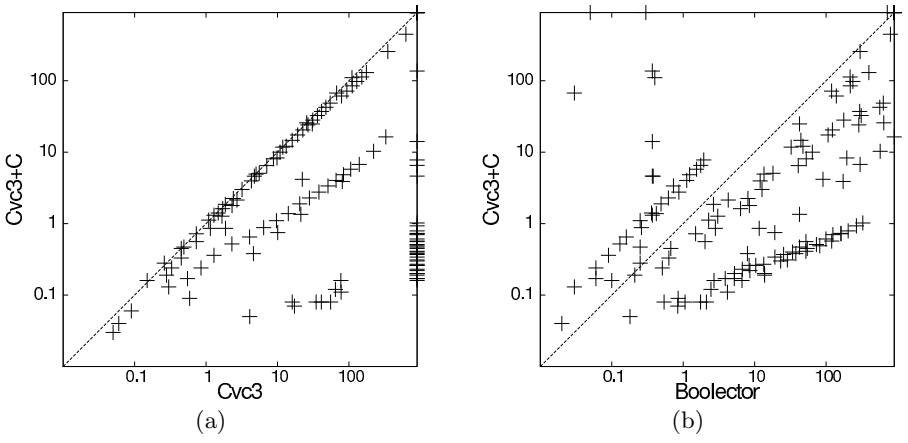


Fig. 1. Comparison of Cvc3, Cvc3+C and Boolector. Both axes use a logarithmic scale and each point represents the time needed to solve an individual problem.

We also presented an experimental evaluation of the method, comparing the new method to a standard Nelson-Oppen implementation and several state-of-the art solvers. Compared to the other solvers on a selected set of benchmarks, the new method performs competitively, and shows a robust performance increase over the standard Nelson-Oppen implementation.

References

1. Barrett, C.W., Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Splitting on Demand in SAT Modulo Theories. In: Hermann, M., Voronkov, A. (eds.) LPAR 2006. LNCS (LNAI), vol. 4246, pp. 512–526. Springer, Heidelberg (2006)
2. Barrett, C., Tinelli, C.: CVC3. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 298–302. Springer, Heidelberg (2007)
3. Bozzano, M., Bruttomesso, R., Cimatti, A., Junttila, T., Ranise, S., van Rossumd, P., Sebastiani, R.: Efficient theory combination via Boolean search. Information and Computation 204(10), 1493–1525 (2006)
4. 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)
5. Bruttomesso, R., Cimatti, A., Franzén, A., Griggio, A., Sebastiani, R.: The MATH-SAT 4 SMT solver. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 299–303. Springer, Heidelberg (2008)
6. Bruttomesso, R., Cimatti, A., Franzen, A., Griggio, A., Sebastiani, R.: Delayed theory combination vs. Nelson-Oppen for satisfiability modulo theories: A comparative analysis. Annals of Mathematics and Artificial Intelligence 55(1), 63–99 (2009)
7. de Moura, L., Bjørner, N.: Model-based Theory Combination. In: 5th International Workshop on Satisfiability Modulo Theories. Electronic Notes in Theoretical Computer Science, vol. 198, pp. 37–49. Elsevier, Amsterdam (2008)

8. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
9. de Moura, L., Bjørner, N.: Generalized, efficient array decision procedures. In: Formal Methods in Computer-Aided Design, 2009, pp. 45–52. IEEE, Los Alamitos (2009)
10. Dutertre, B., de Moura, L.: The YICES SMT Solver (2006) Tool paper at, http://yices.csl.sri.com/tool-paper.pdf
11. Enderton, H.B.: A mathematical introduction to logic. Academic Press, New York (1972)
12. Jovanović, D., Barrett, C.: Polite theories revisited. Technical Report TR2010-922, Department of Computer Science, New York University (January 2010)
13. Jovanović, D., Barrett, C.: Polite theories revisited. In: Fermüller, C.G., Voronkov, A. (eds.) LPAR-17. LNCS, vol. 6397, pp. 402–416. Springer, Heidelberg (2010)
14. Jovanović, D., Barrett, C.: Sharing is Caring: Combination of Theories. Technical Report TR2011-940, New York University (2011)
15. Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. ACM Transactions on Programming Languages and Systems 1(2), 245–257 (1979)
16. Oppen, D.C.: Complexity, convexity and combinations of theories. Theoretical Computer Science 12(3), 291–302 (1980)
17. Rakamarić, Z., Hu, A.J.: A Scalable Memory Model for Low-Level Code. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 290–304. Springer, Heidelberg (2009)
18. Ranise, S., Ringeissen, C., Zarba, C.G.: Combining Data Structures with Nonstably Infinite Theories Using Many-Sorted Logic. In: Gramlich, B. (ed.) FroCos 2005. LNCS (LNAI), vol. 3717, pp. 48–64. Springer, Heidelberg (2005)
19. Shostak, R.E.: An algorithm for reasoning about equality. In: 5th International Joint Conference on Artificial Intelligence, pp. 526–527. Morgan Kaufmann Publishers Inc., San Francisco (1977)
20. Tinelli, C., Harandi, M.T.: A new correctness proof of the Nelson–Oppen combination procedure. In: Frontiers of Combining Systems. Applied Logic, pp. 103–120. Kluwer Academic Publishers, Dordrecht (1996)
21. Tinelli, C., Zarba, C.: Combining decision procedures for sorted theories. In: Alferes, J.J., Leite, J. (eds.) JELIA 2004. LNCS (LNAI), vol. 3229, pp. 641–653. Springer, Heidelberg (2004)

Modular Termination and Combinability for Superposition Modulo Counter Arithmetic

Christophe Ringeissen and Valerio Senni*

LORIA-INRIA Nancy Grand Est
FirstName.LastName@loria.fr

Abstract. Modularity is a highly desirable property in the development of satisfiability procedures. In this paper we are interested in using a dedicated superposition calculus to develop satisfiability procedures for (unions of) theories sharing counter arithmetic. In the first place, we are concerned with the termination of this calculus for theories representing data structures and their extensions. To this purpose, we prove a modularity result for termination which allows us to use our superposition calculus as a satisfiability procedure for combinations of data structures. In addition, we present a general combinability result that permits us to use our satisfiability procedures into a non-disjoint combination method à la Nelson-Oppen without loss of completeness. This latter result is useful whenever data structures are combined with theories for which superposition is not applicable, like theories of arithmetic.

1 Introduction

Software verification tasks require the availability of solvers that are able to discharge proof obligations involving data-structures together with arithmetic constraints and other mathematical abstractions, such as size abstractions. Besides, the use of Satisfiability Modulo Theories (SMT) solvers allows us to focus on the development of satisfiability procedures for such mixed theories. In this setting, the problem of designing the satisfiability procedures is often addressed with success by using approaches based on combination [15].

Problems arise when we consider combinations involving theories whose signatures are non-disjoint. This is especially the case when we consider theories sharing some algebraic constraints [14,16,17,18,20,21]. In order to combine satisfiability procedures for the single theories to handle constraints in their non-disjoint union one needs to rely on powerful methods such as the combination framework of [9,10]. These methods are based on semantic properties of the considered theories, such as compatibility and computability of bases of the shared entailed equalities, which often require complex proofs.

A further issue concerns the development of correct and efficient satisfiability procedures for the single theories, possibly using a systematic approach. In this regard, the use of superposition calculus has proved to be effective

* The author acknowledges support from ERCIM during his stay at LORIA-INRIA.

C. Tinelli and V. Sofronie-Stokkermans (Eds.): FroCoS 2011, LNAI 6989, pp. 211–226, 2011.

to deal with classical data structures, which do not involve arithmetic constraints [1,2,4,5,6,13].

In this paper we address both aspects by: (1) considering a superposition calculus with a built-in theory of counter arithmetic [17,18] and (2) providing modularity results for termination and combinability, based on conditions on the saturations of the component theories that can be checked automatically.

Our contributions are twofold. First, we prove a modular termination result for extending the applicability of the superposition calculus to theories that share a theory of counter arithmetic. This generalizes, to the non-disjoint case, the results in [1], where the authors consider the standard superposition calculus and signature-disjoint theories. This result allows us to drop some of the complex conditions required by the combination framework when we deal with theories that can be treated uniformly through superposition.

Second, we prove a general compatibility result that allows us to use our superposition-based satisfiability procedures into the combination framework of [10]. We prove that any satisfiability procedure obtained by using our modular termination result is able to compute a finite basis of the shared entailed equalities. In addition, we provide a sufficient condition on the form of the saturations of the theories that allows us to conclude compatibility of the component theories with respect to the shared theory and, thus, completeness of their combination.

As an outcome, we have less and simpler restrictions on combinability and we are able to obtain satisfiability procedures both by a uniform approach for theories that can be treated well by superposition (e.g., data structures) and by combination with other solvers for theories which are not 'superposition-friendly' (such as theories of arithmetic).

To show the application of our results in practice, we introduce a class of new theories modeling data structures and equipped with a counting operator that allows us to keep track of the number of the modifications (writes, constructors, etc.) performed on a data structure. In these theories we are able to distinguish between versions of the same data structure obtained by some update.

The paper is organized as follows. In Section 2 we briefly introduce an example in which we use data structures equipped with a mechanism to count the update operations. In Section 3 we introduce some basic notions and recall the superposition calculus for counter arithmetic. In Section 4 we present our modular termination result. In Section 5 we present our general compatibility result. In Section 6 we discuss in details how these results can be applied to our motivating example. Section 7 concludes with some perspectives.

2 A Motivating Example

Let us now consider an example where we show the application of our technique to the analysis of a function *minmax*, defined as follows:

```
function minmax (l : LIST) : RECORD {
    while (l != nil) {
        e := car(l);
```

```
        if e < rselect1(r) then r := rstore1(r, e);
        if rselect2(r) < e then r := rstore2(r, e);
        l := cdr(l)
    };
    return r
}
```

The function *minmax* stores into a binary record the maximum and minimum elements of a given list of rational numbers. We consider a theory of lists T_{LV} (including the classic car and cons operators) and a theory of records T_{RV} (including the the classic $\mathsf{rselect}_i$ and rstore_i operators), both equipped with a counting operator: $\mathsf{count}_R(r)$, which denotes the number of updates performed on the record r, and $\mathsf{count}_L(l)$, which denotes the number of elements inserted into the list l and coincides with the size of the list. In order to verify the correctness of the *minmax* function, we will prove that $\psi\colon\ \forall l, r\,(r = minmax(l) \Rightarrow \mathsf{count}_R(r) \leq \mathsf{count}_L(l))$ holds. The meaning of ψ is that the record r will not be updated more than 'size of l' times.

We will prove the desired property by relying on an SMT solver modulo $(T_{LV} \cup T_{RV} \cup T_S) \cup T_{\mathbb{Q}}$, where T_S is a (shared) theory of counter arithmetic and $T_{\mathbb{Q}}$ is a theory extension of T_S corresponding to the (convex) theory of linear arithmetic over the rationals. We know from [17] that $T_{\mathbb{Q}}$ fulfills the requirements for using the non-disjoint combination framework of [10] when T_S is shared. We develop a satisfiability procedure for $(T_{LV} \cup T_{RV} \cup T_S) \cup T_{\mathbb{Q}}$ in two steps. In the first step we use our result on modular termination to obtain a superposition-based satisfiability procedure for $T_{LV} \cup T_{RV} \cup T_S$. Then, in the second step, we use our general compatibility result to show that the requirements needed to combine $T_{LV} \cup T_{RV} \cup T_S$ with $T_{\mathbb{Q}}$ by using the non-disjoint combination framework of [10] are fulfilled. In Section 6 we discuss these aspects in detail and we also show that, in order to prove ψ we need to add some extra assumptions, namely that r is a 'fresh' record (no update operations have been performed on it) and r has been initialized so that $\mathsf{rselect}_1(r) \leq \mathsf{rselect}_2(r)$.

3 Preliminaries

Let us consider a many-sorted language. A *signature* Σ is a set of sorts, function and predicate symbols (each endowed with the corresponding arity and sort). We assume that, for each sort s, the equality '$\simeq_s$' is a logical constant that does not occur in Σ and that is always interpreted as the identity relation over (the interpretation of) s; moreover, as a notational convention, we will often omit the subscript for sorts and we will use the symbol $\bowtie$ to denote either $\simeq$ or $\not\simeq$. The signature obtained from Σ by adding a set $\underline{a}$ of new constants (i.e., 0-ary function symbols, each of them equipped with its sort) is denoted by $\Sigma^{\underline{a}}$ and named a *constant expansion* of Σ. *Σ-terms*, *Σ-substitutions*, *Σ-atoms*, *Σ-literals*, *Σ-clauses*, and *Σ-formulae* are defined in the usual way (see, e.g., [8]). The empty clause is denoted by $\bot$. A set of Σ-literals is called a *Σ-constraint*. Terms, literals, clauses and formulae are said to be *ground* whenever no variable

appears in them; *sentences* are formulae in which free variables do not occur. Given a function symbol f, a *f-rooted* term is a term whose top-symbol is f. A *compound* term is a f-rooted term for a function symbol f of arity different from 0. Given a term t and a position p, $t_{|p}$ denotes the subterm of t at position p, and $t[l]_p$ denotes the term t in which l appears as the subterm at position p. The *depth* of a term t is defined as follows: $depth(t) = 0$, if t is a constant or a variable, and $depth(f(t_1, \ldots, t_n)) = 1 + \max\{depth(t_i) \mid 1 \leq i \leq n\}$. The *depth* of a literal $l \bowtie r$ is $depth(l \bowtie r) = depth(l) + depth(r)$. We write substitution applications in postfix notation. Substitutions are well-sorted: for any substitution σ and any variable x, $x\sigma$ is a term of the same sort as x.

In order to define models, we rely on the standard notion of a *Σ-structure* $\mathcal{M}$, which consists of: (1) a typed domain D, that is a domain partitioned into a (finite) set of (sub)domains, one for each sort, and (2) a sort- and arity-matching interpretation $\mathcal{I}$ of the function and predicate symbols from Σ. The truth of a Σ-formula in the structure $\mathcal{M}$ is defined in any of the standard ways.

A Σ-theory T is a collection of Σ-sentences, called the axioms of T. If every axiom is a sentence of the form $\forall \overline{x}\, A$, where A is a quantifier free formula, then we say that the theory is *universal*. An *equational theory* is a universal theory whose axioms are universally quantified equalities.

In this paper, we are concerned with the (*constraint*) *satisfiability problem* for a given theory T, also called the T-satisfiability problem, which is the problem of deciding whether a Σ-constraint is satisfiable in a model of T (and, if so, we say that the constraint is T-satisfiable). Note that a constraint may contain variables: since these can be replaced by fresh new constants (preserving satisfiability), we can reformulate the constraint satisfiability problem as the problem of deciding whether a finite conjunction of ground literals in a constant expansion Σ^a is true in a Σ^a-structure whose Σ-reduct is a model of T.

We consider inference systems using well-founded orderings on terms/literals that are total on ground terms/literals. An ordering $\succ$ on terms is a *simplification ordering* [7] if it is stable ($l \prec r$ implies $l\sigma \prec r\sigma$ for every substitution σ), monotonic ($l \prec r$ implies $t[l]_p \prec t[r]_p$ for every term t and position p), and has the subterm property (i.e., it contains the subterm ordering: if l is a strict subterm of r, then $l \prec r$). Simplification orderings are well-founded. A term t is *maximal* in a multiset S of terms if $t \not\preceq u$, for every $u \in S$ different from t. An ordering on terms is extended to literals by using its multiset extension on literals viewed as multisets of terms. Any positive literal $l \simeq r$ (resp. negative literal $l \not\simeq r$) is viewed as the multiset $\{l, r\}$ (resp. $\{l, l, r, r\}$). Also maximality is extended to literals, by defining a term l maximal in a literal whenever l is maximal in the corresponding multiset.

3.1 Superposition Calculus for Counter Arithmetic

Recent literature has focused on the use of superposition calculus to decide the satisfiability of ground formulae in theories extending the theory of Integer Offsets [1,4]. These techniques are based on a problem-specific reduction of the input set of clauses to a new (equisatisfiable) one that admits a finite axiomatization of

the successor function. Then, the standard superposition calculus [3] can be used as a decision procedure for the satisfiability of the obtained set of clauses. Moreover, these approaches allow the combination with other superposition-based decision procedures and ensure termination whenever the involved theories satisfy the so-called 'variable inactivity' property and are signature-disjoint.

In contrast, we are interested in a superposition-based calculus that is able to cope with *non-disjoint* extensions of a theory of Counter Arithmetic.

Theories of Counter Arithmetic. T_S is the theory of Increment, which defines the behavior of the successor function s and the constant 0. T_S has the monosorted signature $\Sigma_S := \{0 : \text{NUM}, \mathsf{s} : \text{NUM} \to \text{NUM}\}$, and it is axiomatized as follows[1]: $\{\mathsf{s}(x) \simeq \mathsf{s}(y) \to x \simeq y\} \cup \{x \not\simeq \mathsf{s}^n(x) \mid n \in \mathbb{N}, n > 0\}$. T_I is the theory of Integer Offsets defined as $T_S \cup \{\mathsf{s}(x) \not\simeq 0\}$. In the following we will generically denote as T_C any theory in the set $\{T_S, T_I\}$.

In order to deal with theories that are extensions of a theory of Counter Arithmetic we consider the superposition calculus of [18], presented in Figure 1, which extends the standard superposition calculus of [3] to take into account the axioms of the theories T_S or T_I. The difference between the classical calculus and the one we consider in this paper is twofold: (1) this calculus is specialized for reasoning over sets of *literals*, and (2) this calculus is augmented with four rules over ground terms, called Ground Reduction Rules, that encode directly into the calculus the axioms of the theory of Counter Arithmetic.

We will use this calculus to check the satisfiability of any set of ground 'flat' literals modulo a set of axioms. In the context of this paper, a literal is said to be *flat* if it is a Σ_S-literal or a positive literal of depth 1 which is not a Σ_S-literal.

Definition 1. *Let $\mathcal{SP}^{\succ}_I$ be the calculus presented in Figure 1. Let $\mathcal{SP}^{\succ}_S$ be the calculus obtained from $\mathcal{SP}^{\succ}_I$ by removing the rule C1. Let T_C be the generic name for a theory chosen between T_I and T_S and, analogously, let $\mathcal{SP}^{\succ}_C$ be the generic name for a calculus chosen between $\mathcal{SP}^{\succ}_I$ and $\mathcal{SP}^{\succ}_S$.*

The simplification ordering $\succ$ used in the conditions of the rules is total on ground terms.

We now introduce two crucial notions: goodness and safety. The first restricts the choice of a reduction ordering $\succ$ when using the $\mathcal{SP}^{\succ}_C$ calculus. The second is a property of the saturations obtained using $\mathcal{SP}^{\succ}_C$. In [18] it is shown that goodness and safety are sufficient to guarantee that $\mathcal{SP}^{\succ}_C$ is a decision procedure for the satisfiability problem of equational theories extending Counter Arithmetic. In this paper we show that these properties are also sufficient to ensure the modular termination of $\mathcal{SP}^{\succ}_C$ and combinability, when applied to unions of theories.

A simplification ordering $\succ$ that is total on ground terms on a signature containing Σ_S is s-*good* if (1) $t \succ c$ for every ground compound term t which is not s-rooted and every constant c, (2) 0 is minimal, and (3) whenever two terms t_1 and t_2 are not s-rooted we have $\mathsf{s}^m(t_1) \succ \mathsf{s}^n(t_2)$ iff either $t_1 \succ t_2$ or $(t_1 = t_2$ and $m > n)$.

[1] All the axioms are (implicitly) closed under universal quantification.

A *derivation* is a sequence $S_0, \ldots, S_i, \ldots$ such that each S_i is a set of literals obtained from S_{i-1} by applying an inference rule to literals in S_{i-1}. We denote S_ω the set of *persistent* literals $\bigcup_i \bigcap_{j>i} S_j$. When the derivation is finite, the set of persistent literals coincides with the last set of the derivation. A derivation is *fair* if whenever an inference is applicable it will be eventually applied unless one of the literals that would be involved in this inference is simplified, subsumed, or deleted (see, e.g. [18] for a formal definition). The set of persistent literals S_ω obtained by a fair derivation is called the *saturation* of the derivation. In the following, we will only consider fair derivations.

Definition 2. *The saturation S_ω of a fair derivation δ w.r.t. $\mathcal{SP}^{\succ}_{C}$ is* safe *if, whenever S_ω does not contain $\bot$, we have that, for every literal $L \in S_\omega$ and any maximal term t in L: (1) if L is an equality and t is a variable then t is not of sort* NUM*, and (2) if L is an equality and t is* s*-rooted then L is ground. S_ω is* proper *if in addition, we have that: (3) if t has an* s*-rooted subterm u then the direct subterm of u is a non-variable.*

Note that condition (1) in Definition 2 is related to *variable inactivity* [1].

Definition 3. *Consider an equational Σ-theory T such that $\Sigma \supseteq \Sigma_S$, and assume $\mathcal{SP}^{\succ}_{C}$ is used with a* s*-good ordering $\succ$. The theory T is* terminating *(resp.* safely terminating*/*properly terminating*) w.r.t. $\mathcal{SP}^{\succ}_{C}$ if, for any set G of ground flat literals (built out of symbols from Σ and possibly further free constants), we have that:*

1. *there exists a saturation S_ω of $T \cup G$ w.r.t. $\mathcal{SP}^{\succ}_{C}$ which is finite (resp. finite and safe/finite and proper),*
2. *for any set G' of ground Σ_S-literals (built out of symbols from Σ_S and possibly further free constants) such that $G' \cap G = \emptyset$, there exists a saturation of $S_\omega \cup G'$ w.r.t. $\mathcal{SP}^{\succ}_{C}$ which is finite (resp. finite and safe/finite and proper).*

In this paper, we consider two different ways to find theories satisfying Definition 3. In Section 6, we introduce theories for which all the ground saturations modulo T are of the expected forms. In Section 4, we consider unions of theories for which the saturations described in Definition 3 have the expected forms.

Theorem 1 ([18]). *if T is safely terminating w.r.t. $\mathcal{SP}^{\succ}_{C}$, then $\mathcal{SP}^{\succ}_{C}$ induces a decision procedure for the constraint satisfiability problem w.r.t. $T \cup T_C$.*

Note that Definition 3 is slightly stronger than the one of [18]. This is motivated by the assumptions we need to prove modular termination in Section 4.

3.2 Background on Non-disjoint Combination

Combination techniques are widely studied to build decision procedures for complex theories by using decision procedures for simpler component theories. The Nelson-Oppen method [15] applies to disjoint unions of theories that satisfy stably infiniteness. The combination framework we consider here is an extension of

Expansion Rules:

Superposition	$\dfrac{l[u'] \simeq r \quad u \simeq v}{(l[v] \simeq r)\sigma}$	if (i) and (ii)
Paramodulation	$\dfrac{l[u'] \not\simeq r \quad u \simeq v}{(l[v] \not\simeq r)\sigma}$	if (i) and (ii)
Reflection	$\dfrac{u \not\simeq u'}{\bot}$	

Where the substitution σ is the most general unifier of u and u', and u' is *not* a variable in *Superposition* and *Paramodulation*. Moreover, we have the following conditions: (i) $u\sigma \not\leq v\sigma$, and (ii) $l[u']\sigma \not\leq r\sigma$.

Contraction Rules:

Subsumption	$\dfrac{S \cup \{L, L'\}}{S \cup \{L\}}$	if $L\vartheta = L'$ for some substitution ϑ
Simplification	$\dfrac{S \cup \{L[l'], l \simeq r\}}{S \cup \{L[\vartheta(r)], l \simeq r\}}$	if $l' = l\vartheta$, $l\vartheta \succ r\vartheta$, and $L[l\vartheta] \succ (l\vartheta \simeq r\vartheta)$
Deletion	$\dfrac{S \cup \{t \simeq t\}}{S}$	

Ground Reduction Rules:

R1	$\dfrac{S \cup \{\mathsf{s}(u) \simeq \mathsf{s}(v)\}}{S \cup \{u \simeq v\}}$	if u and v are ground
R2	$\dfrac{S \cup \{\mathsf{s}(u) \simeq t, \mathsf{s}(v) \simeq t\}}{S \cup \{u \simeq v, \mathsf{s}(v) \simeq t\}}$	if u, v, and t are ground, and $\mathsf{s}(u) \succ t$, $\mathsf{s}(v) \succ t$, and $u \succ v$
C1	$\dfrac{S \cup \{\mathsf{s}(t) \simeq 0\}}{S \cup \{\mathsf{s}(t) \simeq 0\} \cup \{\bot\}}$	if t is ground
C2	$\dfrac{S \cup \{\mathsf{s}^n(t) \simeq t\}}{S \cup \{\mathsf{s}^n(t) \simeq t\} \cup \{\bot\}}$	if t is ground and $n > 0$

Fig. 1. The superposition calculus $\mathcal{SP}^{\succ}_{C}$ with built-in counter arithmetic

Nelson-Oppen to the non-disjoint case [9] and combines satisfiability procedures having the capability of deducing logical consequences over the shared signature Σ_0. In order to ensure the termination when deducing logical consequences over the shared signature, we assume that the shared Σ_0-theory is Noetherian [10]. Intuitively, a theory T_0 is Noetherian if there exists only a finite number of atoms that are not redundant when reasoning modulo T_0.

Proposition 1 ([18]). *T_C is Noetherian.*

Consider a theory $T \supseteq T_0$ with signatures $\Sigma \supseteq \Sigma_0$. Given an arbitrary set of ground clauses Θ over Σ, the notion of T_0-*basis* allows us to define a 'complete set' of positive logical consequences of Θ over Σ_0.

Definition 4 (T_0-basis). *Given a finite set Θ of ground clauses (built out of symbols from Σ and possibly further free constants) and a finite set of free constants $\underline{a}$, a* T_0-basis *modulo T for Θ w.r.t. $\underline{a}$ is a set Δ of* positive *ground $\Sigma_0^{\underline{a}}$-clauses, denoted by $T_0\text{-}\mathsf{basis}_T(\Theta)$, such that*

(i) $T \cup \Theta \models C$, for all $C \in \Delta$ and
(ii) if $T \cup \Theta \models C$ then $T_0 \cup \Delta \models C$, for every positive ground $\Sigma_0^{\underline{a}}$-clause C.

Note that in the definition of a basis we are interested only in positive ground clauses: the exchange of positive information is sufficient to ensure the completeness of the resulting procedure. The interest in Noetherian theories lies in the fact that, for every set of Σ-clauses Θ and for every set $\underline{a}$ of constants, a finite T_0-basis for Θ w.r.t. $\underline{a}$ always exists [10]. Note that if Θ is T-unsatisfiable then w.l.o.g. $\Delta = \{\bot\}$. Unfortunately, a basis for a Noetherian theory does not need to be computable; this motivates the following definition.

Definition 5. *A theory T is an* effectively Noetherian extension of T_0 *if and only if T_0 is Noetherian and a T_0-basis modulo T w.r.t. $\underline{a}$ is computable for every set of literals and every finite set $\underline{a}$ of free constants.*

Theorem 2 ([18]). *Let $\underline{a}$ be a finite set of free constants. Assume $\mathcal{SP}_C^{\succ}$ is used with a* s-good *ordering $\succ$ such that any $\Sigma_S^{\underline{a}}$-term is smaller than any term containing a function symbol not in $\Sigma_S^{\underline{a}}$. If T is safely terminating w.r.t. $\mathcal{SP}_C^{\succ}$, then $\mathcal{SP}_C^{\succ}$ is able to compute a T_C-basis modulo $T \cup T_C$ w.r.t. $\underline{a}$.*

The combination method works by exchanging the shared clauses obtained from procedures computing T_0-bases. To ensure completeness, we rely on the notion of T_0-compatibility [9] that extends the notion of stably infiniteness used in the disjoint case. We do not give here a general definition of T_0-compatibility, but we recall how it instantiates in the particular cases of T_I and T_S.

Proposition 2 ([18]). *A theory T such that $T \supseteq T_I$ is T_I-compatible iff every T-satisfiable constraint is satisfiable in a model of T in which $\forall x(x \not\simeq 0 \Rightarrow \exists y\, x \simeq \mathsf{s}(y))$ holds. A theory T such that $T \supseteq T_S$ is T_S-compatible iff every T-satisfiable constraint is satisfiable in a model of T in which $\forall x\ \exists y\ x \simeq \mathsf{s}(y)$ holds.*

The union of a Σ_1-theory T_1 and a Σ_2-theory T_2 *shares* the Σ_0-theory T_0 if $T_0 \subseteq T_1$, $T_0 \subseteq T_2$, and $\Sigma_1 \cap \Sigma_2 = \Sigma_0$. The following theorem states the modularity result we obtain by applying the Nelson-Oppen combination method extended to unions of theories sharing T_0.

Theorem 3 (Non-disjoint Nelson-Oppen [10]). *Let T_0 be a Noetherian Σ_0-theory. The class of theories which are T_0-compatible and effective Noetherian extensions of T_0 is closed under union sharing T_0.*

In [18], we studied how to apply Theorem 3 when the shared theory is T_C, by considering effectively Noetherian extensions for which $\mathcal{SP}^{\succ}_C$ terminates and is able to compute T_C-bases. This paper is the continuation of [18] with two new contributions. First, we show a modularity result for termination w.r.t. $\mathcal{SP}^{\succ}_C$. Second, we show a general T_C-compatibility result. For these results, we rely on the notions of safe and proper termination w.r.t. $\mathcal{SP}^{\succ}_C$ (see Definition 3).

4 Modular Termination

We are interested in designing a satisfiability procedure for the union of safely terminating theories sharing T_C. Since T_C is Noetherian, an obvious solution could be to use the combination method provided by Theorem 3. In general, this is not an easy task since it requires to prove T_C-compatibility and effective Noetherianity of the component theories. By Theorem 2, we know that whenever the saturation is safe we can infer effective Noetherianity.

As an alternative, we propose a modular termination result that applies to the union of the considered theories and is based on the analysis of the saturations (similarly to [14]). This result is interesting in that it does not require us to prove the more complex property of T_C-compatibility for the component theories.

Besides, the application of the combination framework is very useful whenever we consider theories that cannot be easily handled through superposition, such as the theories of arithmetic. By Theorem 2, we know that modular termination entails effective Noetherianity for the union of (non-disjoint) theories. This result satisfies the first requirement of the combination framework. To satisfy the second, we show in Section 5 that the strengthening of safe termination into proper termination is enough to prove compatibility.

In the following, an s-*equality* is a ground equality of the form $a \simeq \mathsf{s}^m(b)$, for some constants a and b of sort NUM.

Lemma 1. *For any theory $T \supseteq T_C$ and any finite set of ground flat literals G, if* (1) *a saturation S_ω of $T \cup G$ by $\mathcal{SP}^{\succ}_C$ using a s-good ordering does not contain $\bot$ and* (2) *every s-equality in S_ω is either of the form* (i) *$a \simeq \mathsf{s}^m(b)$, for $m \geq 0$ and $a \succ b$, or of the form* (ii) *$\mathsf{s}^m(a) \simeq b$, for $m \geq 1$ and $a \succ b$, then S_ω contains at most one such equality for each pair of distinct constants a, b.*

Proof: By contradiction, assume there are two s-equalities $a \simeq \mathsf{s}^{m_1}(b)$ and $a \simeq \mathsf{s}^{m_2}(b)$ in S_ω with $m_1 \neq m_2$ of the form (i). By superposition in S_ω there is also the literal $\mathsf{s}^{m_1}(b) \simeq \mathsf{s}^{m_2}(b)$, where $m_1 + m_2 > 0$, and $\bot$, generated from that literal by a finite number of applications of rule R1 and an application of rule C2. Let us now assume there are two s-equalities $a \simeq \mathsf{s}^{m_1}(b)$ and $\mathsf{s}^{m_2}(a) \simeq b$ in S_ω of the form (i) and (ii), respectively. Again, by superposition, we would have also $\mathsf{s}^{m_1+m_2}(b) \simeq b$, where $m_1 + m_2 > 0$, and $\bot$, generated from such literal by an application of rule C2. Finally, let us assume there are two s-equalities $\mathsf{s}^{m_1}(a) \simeq b$ and $\mathsf{s}^{m_2}(a) \simeq b$ in S_ω with $m_1 \neq m_2$ and $m_1, m_2 \geq 1$ of the form (ii). This is impossible by fairness, since R2 could have been applied and one of the two would have been deleted. Therefore, we have at most one equality either of the form (i) or of the form (ii) for each pair of constants. □

Theorem 4. *Assume $\mathcal{SP}^{\succ}_{C}$ is used with a* s*-good ordering $\succ$. The class of theories which are safely terminating (resp. properly terminating) w.r.t. $\mathcal{SP}^{\succ}_{C}$ is closed under union sharing T_C.*

Proof: Let T_i be a safely terminating Σ_i-theory, and G_i be a Σ_i-constraint, for $i = 1, 2$, such that $T_1 \cup T_2$ shares T_C. We first consider the preservation of termination for $T_1 \cup T_2$, and then we consider the preservation of safety (resp. properness) for $T_1 \cup T_2$.

Termination. Let us consider a fair derivation $\delta = S_0, \ldots, S_k, S_{k+1}, \ldots$ starting from $S_0 = T_1 \cup T_2 \cup G_1 \cup G_2$. We will show (by induction on the length of the derivation) that, for each $S_k \in \delta$: (1) S_k is of the form $S^1_k \cup S^2_k$ where S^i_k is the set of Σ_i-literals in S_k, for $i = 1, 2$, and (2) any saturation of S^i_k is finite for $i = 1, 2$. By assumption, these two properties hold for $k = 0$. To prove the inductive case, it is sufficient to show that "across-theories" inferences generate only (finitely many) ground shared literals. To do so, we assume that we use a fair strategy that consists in computing the saturations modulo T_1 and T_2 before applying the "across-theories" inferences. Let S^1_k and S^2_k be saturated. By assumption, we know that S^1_k and S^2_k are safe. Let us analyze superposition and paramodulation inferences (uniformly called paramodulation in the following) from the set S^i_k into the set S^j_k, for $i \neq j$. We have three cases:

(a) Paramodulation from variables. Let $x \simeq t$ be a literal in S^i_k. In order to paramodulate into a literal of S^j_k the variable x must be of sort NUM. By condition (1) of safety, x cannot be maximal in $x \simeq t$ and, therefore, no paramodulation from variables is possible.

(b) Paramodulation from constants. Let $a \simeq t$ be a literal L in S^i_k, where a is a constant. In order to paramodulate from a into a literal in S^j_k, such literal must be of the form $l[a]_p \bowtie r$ and the corresponding m.g.u. is empty. Again, a is of sort NUM. By condition (1) of safety, t can be either a compound term or a constant and, in order to trigger a paramodulation, it must be $a \not\succeq t$. Assume t is compound and of the form $\mathsf{s}^m(u)$ for some $m \geq 0$ and term u which is compound and not s-rooted. Then, by condition (1) of s-goodness, $u \succ a$ and, by the subterm property of $\succ$, $t \succeq u \succ a$, which is a contradiction with the assumption that a is maximal. Thus, we conclude that t is of the form $\mathsf{s}^m(v)$, where v is either a constant or a variable. Now let v be a variable, then also $\mathsf{s}^m(v)$ is maximal and since L is not ground this is not allowed by condition (2) of safety. As a consequence, we have that L is of the form $a \simeq \mathsf{s}^m(b)$, for some constant b, $m \geq 0$, and $a \succ b$.

(c) Paramodulation from compound terms. Let $t \simeq u$ be a literal L in S^i_k and $l \bowtie r$ a literal in S^j_k, where t is a compound term. Since T_1 and T_2 share symbols in a constant expansion of Σ_S, the term t must be s-rooted. Let σ be the m.g.u. of t and $l_{|_p}$, for some position p, then paramodulation requires that $t\sigma \not\preceq u\sigma$. By stability of $\succ$ we have $t \not\preceq u$ as well and t is maximal in $t \simeq u$. By the safety assumption, the literal $t \simeq u$ is ground, σ is the empty substitution, and $t \succ u$. Now, since S^i_k is saturated, u is not s-rooted, otherwise that literal would have been deleted by an application of rule R1. By the safety assumption, L is a

ground literal of the form $\mathsf{s}^m(a) \simeq b$, for $m \geq 1$ and $a \succ b$ (the case where $a = b$ is ruled out by the applicability of rule C2).

The literals needed to perform the paramodulation steps of cases (b) and (c) above are s-equalities that satisfy the assumptions of Lemma 1. Hence, by Lemma 1, the set of literals in the saturation of δ that allow a paramodulation in one of the above cases is finite, because it is bounded by n^2, where n is the finite number of constants occurring in S_0. To formally prove the termination of $\mathcal{SP}^{\succ}_C$, let us consider the following complexity measures for any $S_k \in \delta$: $mse(S_k)$ is n^2 minus the number of s-equalities in S_k, and $mns_i(S_k)$ is the number of remaining steps to reach the saturation of S^i_k, for $i = 1, 2$. We can verify that for any k: (0) $mse(S_k) > mse(S_{k+1})$, or (1) $mse(S_k) \geq mse(S_{k+1})$ and $mns_1(S_k) > mns_1(S_{k+1})$, or (2) $mse(S_k) \geq mse(S_{k+1})$ and $mns_1(S_k) \geq mns_1(S_{k+1})$ and $mns_2(S_k) > mns_2(S_{k+1})$. Therefore, the complexity measure defined as the lexicographic combination of mse, mns_1 and mns_2 is strictly decreased by (each step of) δ, and so δ is finite.

Safe and proper termination. The saturation S_ω of any fair derivation δ is of the form $S^1_\omega \cup S^2_\omega$ such that, for $i = 1, 2$, S^i_ω is a saturation of $T_i \cup G_i \cup E$, where E is the finite set of ground s-equalities generated by δ. By assumption, S^1_ω and S^2_ω are safe (resp. proper), and so $S_\omega = S^1_\omega \cup S^2_\omega$ is safe (resp. proper) too.

The superposition strategy defined for $T_1 \cup T_2$ can be used in an incremental way: given a set G' of new shared literals disjoint from $G_1 \cup G_2$, one can easily check that the saturation of $S_\omega \cup G'$ is still finite since T_1 and T_2 follow Definition 3. □

5 A General Compatibility Result

In this section we show that, by analyzing the saturations, we can infer T_C-compatibility of equational theories extending T_C.

Theorem 5 (T_C-compatibility). *Assume $\mathcal{SP}^{\succ}_C$ is used with a* s-good *ordering $\succ$. If T is properly terminating w.r.t. $\mathcal{SP}^{\succ}_C$, then $T \cup T_C$ is T_C-compatible.*

Proof. By Proposition 2, to show that $T \cup T_C$ is T_C-compatible we have the two cases: (a) for T_I we need to prove that every T-satisfiable constraint S is satisfiable in a model of T where $Pred_I : \forall x(x \not\simeq 0 \Rightarrow \exists y\, x \simeq \mathsf{s}(y))$ holds, and (b) for T_S we need to prove that every T-satisfiable constraint S is satisfiable in a model of T where $Pred_S : \forall x\, \exists y\, x \simeq \mathsf{s}(y)$ holds. We will show that a model $\mathcal{M}$ of $T \cup G$ can be extended to obtain a new model $\mathcal{M}_e$ that satisfies also the above axioms $Pred_I$ and $Pred_S$, respectively (we will refer generically to $Pred_C$).

The construction of the model $\mathcal{M}_e$ from $\mathcal{M}$ is the same for the two theories T_I and T_S, and consists of two steps: (1) we build a sequence $\mathcal{M}_0, \mathcal{M}_1, \ldots$ of models of $T \cup G$, and (2) we define the model $\mathcal{M}_e$ as the *direct limit* [11] of this sequence. Then $\mathcal{M}_e$, by construction, satisfies $T \cup G$ and also the axiom $Pred_C$.

Step 1. We define the sequence $\mathcal{M}_0, \mathcal{M}_1, \ldots$ of models of $T \cup G$ (some of which may be identical) as follows. Each model is constructed starting from a saturated

set S^i_ω of unit clauses over a signature Σ_i obtained by extending Σ with a finite set of new constants. Then, we consider the corresponding set $ground(S^i_\omega)$ of all the ground instances of clauses in S^i_ω w.r.t. Σ_i. From $ground(S^i_\omega)$, by using the so-called *model generation* technique [3], we construct a convergent rewriting system $\mathcal{R}_i$ such that, for any pair of ground terms l and r, $S^i_\omega \models l = r$ iff $l{\downarrow_{\mathcal{R}_i}} = r{\downarrow_{\mathcal{R}_i}}$. Finally, the model $\mathcal{M}_i$ is defined as the pair (D_i, F_i), where the domain (or carrier) D_i is the set $T(\Sigma_i)|_{\mathcal{R}_i}$ of $\mathcal{R}_i$-normal forms, and, for every function symbol $f \in \Sigma_i$, $f_{F_i}(t_1, \ldots, t_n) = f(t_1, \ldots, t_n){\downarrow_{\mathcal{R}_i}}$ for every $t_1, \ldots, t_n \in D_i$. Obviously $\mathcal{M}_i$ is, by construction, a model of S^i_ω.

We can construct the models $\mathcal{M}_0, \ldots, \mathcal{M}_i, \mathcal{M}_{i+1}, \ldots$ as follows. The set S^0_ω is the set S_ω which is a saturation of $T \cup G$ using $\mathcal{SP}^{\succ}_C$ and Σ_0 is the signature Σ. For $i \geq 0$, if there is a constant c in Σ_i of sort NUM such that $c{\downarrow_{\mathcal{R}_i}} \neq \mathsf{s}(t){\downarrow_{\mathcal{R}_i}}$ for any ground term t and ($T_C = T_S$ or $c{\downarrow_{\mathcal{R}_i}} \neq 0$), then we define $\Sigma_{i+1} = \Sigma_i \cup \{c'\}$ and $S^{i+1}_\omega = S^i_\omega \cup \{\mathsf{s}(c') = c\}$, where c' is a constant of sort NUM that does not belong to Σ_i, and $c' \succ c$. Otherwise, we define $\Sigma_{i+1} = \Sigma_i$ and $S^{i+1}_\omega = S^i_\omega$. Now, by assumption of proper termination, there is no occurrence of the term $\mathsf{s}(x)$ in S^i_ω and, since no inference is applicable using the new literal $\mathsf{s}(c') = c$, S^{i+1}_ω is saturated. As a consequence, let φ be a literal in G or a clause in T, for every $i \geq 0$, if $S^i_\omega \models \varphi$ then $S^{i+1}_\omega \models \varphi$. We have also that, for all $0 \leq i < j$, $\mathcal{M}_i \subseteq \mathcal{M}_j$, that is $D_i \subseteq D_j$ and the inclusion map $D_i \to D_j$ is an embedding.

Step 2. We define the limit model $\mathcal{M}_e$ as the pair (D_e, F_e) where the domain is $D_e = \bigcup_{i \geq 0} D_i$ and, for every n-ary function symbol $f \in \Sigma$, and elements $a_1, \ldots, a_n$ of D_e, (by a little abuse of notation) $f_{F_e}(a_1, \ldots, a_n) = f_{F_i}(a_1, \ldots, a_n)$ where $i \geq 0$ is the smallest integer such that $a_1, \ldots, a_n$ are in the domain D_i. By construction, there is an embedding between $\mathcal{M}_i$ and $\mathcal{M}_j$ for $i < j$ (i.e. an injective homomorphism). By (a many-sorted version of) Theorem 2.4.6 in [11] we have that the truth value of a clause φ in $G \cup T$ is *preserved* in $\mathcal{M}_e$, that is, if $\mathcal{M}_0 \models \varphi$ then $\mathcal{M}_e \models \varphi$, and $\mathcal{M}_e$ is a model of $T \cup G$. Furthermore, by construction, there is no constant whose interpretation in $\mathcal{M}_e$ has no predecessor, which entails that also the axiom $Pred_C$ holds.

By these observations we have that the theory T is T_C-compatible. □

The following example shows why we need to strengthen the notion of safe termination into that of proper termination in order to prove T_C-compatibility.

Example 1. Consider the theory $T = \{f(\mathsf{s}(x)) \not\simeq f(c)\}$, which is safely terminating. Let S_ω be a saturation and c be a constant that has no predecessors. If introduce the equality $c \simeq \mathsf{s}(c')$ in S_ω, for some fresh new constant c' not occurring in $T \cup S$. Then, by Paramodulation, we get $f(c) \not\simeq f(c)$ and, thus, the empty clause $\bot$.

6 Applying Modular Termination and Combination

We now consider in more details the function *minmax* introduced in the example of Section 2. We show that the verification task involves theories that are suitable for the application of our modular termination and combination results.

We start by defining the theories of lists T_{LV} and of records T_{RV}. These are enriched with an operator count_D that counts the number of modifications that have been performed on a 'fresh' data structure. Any constant c in the sort of a given data structure D can be declared fresh, i.e. unmodified, by adding the ground literal $\mathsf{count}_D(c) \simeq 0$. Due to the count_D operator, these theories are said to allow 'versioning'. We focus on the constraint satisfiability problem for these theories and their combinations. Through the analysis of their saturations we show that our modularity results enable us to combine these theories together and with the theory of Linear Rational Arithmetic $T_{\mathbb{Q}}$, so to obtain the satisfiability procedure which is necessary to solve our verification problem.

Lists with Versioning. T_{LV} is a theory of lists with extensionality endowed with the counting operator. The many-sorted signature of T_{LV} is given by Σ_S plus the set of function symbols $\{\mathsf{nil} : \textsc{list}, \mathsf{car} : \textsc{list} \to \textsc{elem}, \mathsf{cdr} : \textsc{list} \to \textsc{list}, \mathsf{cons} : \textsc{elem} \times \textsc{list} \to \textsc{list}, \mathsf{count}_L : \textsc{list} \to \textsc{num}\}$ and the predicate symbol $\mathsf{atom} : \textsc{list}$. The axioms of T_{LV} are:

$$\begin{array}{ll}
\mathsf{car}(\mathsf{cons}(x,y)) \simeq x & \neg\mathsf{atom}(x) \to \mathsf{cons}(\mathsf{car}(x),\mathsf{cdr}(x)) \simeq x \\
\mathsf{cdr}(\mathsf{cons}(x,y)) \simeq y & \neg\mathsf{atom}(\mathsf{cons}(x,y)) \\
 & \mathsf{atom}(\mathsf{nil}) \\
\mathsf{count}_L(\mathsf{cons}(x,y)) \simeq \mathsf{s}(\mathsf{count}_L(y)) & \mathsf{count}_L(\mathsf{nil}) \simeq 0
\end{array}$$

Proposition 3. *T_{LV} is properly terminating w.r.t. $\mathcal{SP}^{\succ}_{C}$.*

Proof. We can drop the non-equational axioms by using the reduction described in [2]. Termination is proved in [18]. In the case of lists, the axioms for the count_L operator are identical up to a renaming to those of the list length operator. Therefore, we can use the analysis of the saturation done in [18]. Any inference generates only ground literals (which do not influence safety/properness). Thus, we can conclude that any saturation S_ω is proper and T_{LV} is properly terminating. □

Records with Versioning. T_{RV} is theory of records with extensionality endowed with the counting operator. The many-sorted signature of T_{RV} is given by Σ_S and the function symbols defined as follows. Let RECORD be the sort of records; for every attribute identifier we have two functions $\mathsf{rselect_i} : \textsc{record} \to \textsc{elem}_i$ and $\mathsf{rstore_i} : \textsc{record} \times \textsc{elem}_i \to \textsc{record}$, where $1 \le i \le n$. Moreover, there is the function $\mathsf{count}_R : \textsc{record} \to \textsc{num}$ that counts the number of $\mathsf{rstore_i}$ operations performed on a record. The axioms of T_{RV} are (for every i, j such that $1 \le i, j \le n$ and $i \neq j$):

$$\begin{array}{rl}
\mathsf{rselect_i}(\mathsf{rstore_i}(x,y)) \simeq y & \\
\mathsf{rselect_j}(\mathsf{rstore_i}(x,y)) \simeq \mathsf{rselect_j}(x) & \\
\wedge_{i=1}^{n}(\mathsf{rselect_i}(x) \simeq \mathsf{rselect_i}(y)) \to x \simeq y & \text{(extensionality)} \\
\mathsf{count}_R(\mathsf{rstore_i}(x,y)) \simeq \mathsf{s}(\mathsf{count}_R(x)) &
\end{array}$$

By using the reduction described in [1], which is valid also with the additional axioms for the function count_R, it is possible to drop the extensionality axiom,

so that the theory of records is *equational*. As a consequence, we restrict our attention to (equisatisfiable) sets of literals in which no disequation between records appears and we focus on the saturation of sets of literals of the forms:

i. equational axioms for records:
   a. $\mathsf{rselect_i}(\mathsf{rstore_i}(x,y)) \simeq y$, b. $\mathsf{rselect_j}(\mathsf{rstore_i}(x,y)) \simeq \mathsf{rselect_j}(x)$,
   c. $\mathsf{count}_R(\mathsf{rstore_i}(x,y)) \simeq \mathsf{s}(\mathsf{count}_R(x))$;
ii. ground literals over the sorts RECORD and ELEM_i, for $i \in \{1,\ldots,n\}$:
   a. $r \simeq r'$, b. $e \simeq e'$, c. $e \not\simeq e'$, d. $\mathsf{rselect_i}(r) \simeq e$, e. $\mathsf{rstore_i}(r,e) \simeq r'$;
iii. ground literals over the sort NUM:
   a. $\mathsf{count}_R(r) \simeq \mathsf{s}^n(k)$, b. $\mathsf{s}^n(k) \simeq k'$, c. $\mathsf{s}^m(k) \not\simeq \mathsf{s}^n(k')$;

where e, e' are constants of sort ELEM_i, r, r' are constants of sort RECORD, and k, k' are constants of sort NUM. Note that 0 is one of the constants of sort NUM and, thus, in case (iii.a) is included also the literal $\mathsf{count}_R(r) \simeq 0$.

We consider an LPO ordering $\succ$ over terms such that the underlying precedence over the symbols in the signature satisfies the following requirements: for all i, j in $\{1,\ldots,n\}$, $\mathsf{rstore_i} > \mathsf{rselect_j}$, $\mathsf{rstore_i} > \mathsf{count}_R$, $\mathsf{rselect_i} > c$, and $\mathsf{count}_R > c > 0 > \mathsf{s}$, for every constant c.

Proposition 4. *T_{RV} is properly terminating w.r.t. $\mathcal{SP}^{\succ}_{C}$.*

Proof. Let us consider a set S_0 of literals of the form (i)–(iii). We prove that any saturation S_ω of S_0 constructed using $\mathcal{SP}^{\succ}_{C}$ is finite and proper. Any literal introduced by an inference rule is ground and smaller that the biggest literal in the input set. By well-foundedness of the multiset extension of the (well-founded) ordering $\succ$ we get termination. Since the saturation generates only ground literals (which do not affect safety/properness), the analysis of (i)–(iii) is sufficient to conclude that S_ω is proper and T_{RV} is properly terminating. □

Corollary 1. *$T_{LV} \cup T_{RV}$ is properly terminating w.r.t. $\mathcal{SP}^{\succ}_{C}$, $\mathcal{SP}^{\succ}_{C}$ computes a T_C-basis modulo $T_{LV} \cup T_{RV} \cup T_C$, and $T_{LV} \cup T_{RV} \cup T_C$ is T_C-compatible.*

Proof. By Propositions 3 and 4, Theorem 4, and Theorem 5. □

Theory of Linear Rational Arithmetic. $T_{\mathbb{Q}}$ is the theory of Linear Rational Arithmetic discussed in [18], whose signature over the sort NUM is $\Sigma_{\mathbb{Q}} := \{0, 1, +, -, \mathsf{s}, <\}$, where $0, 1$ are constants, $-$ and s are unary function symbols, $+$ is a binary function symbol and $<$ is a binary predicate symbol. The symbols $0, 1, +, -, \mathsf{s}, <$ are interpreted in their intended meaning. In particular, s is the function that associates to each rational q the rational $q + 1$. Clearly, $T_S \subseteq T_{\mathbb{Q}}$. In [18], it is shown that all the assumptions of the Combination Theorem (Theorem 3) are fulfilled for $T_{\mathbb{Q}}$ when $T_0 = T_S$.

As a consequence of our modular termination result we have that $\mathcal{SP}^{\succ}_{S}$ is a satisfiability procedure for $T_{LV} \cup T_{RV} \cup T_S$ and it can be used to compute a T_S-basis modulo $T_{LV} \cup T_{RV} \cup T_S$. Since $T_{LV} \cup T_{RV} \cup T_S$ is also T_S-compatible, all the assumptions of the Combination Theorem (Theorem 3) are satisfied. Hence, we can construct a combined satisfiability procedure for $(T_{LV} \cup T_{RV} \cup T_S) \cup T_{\mathbb{Q}}$ by combining the $\mathcal{SP}^{\succ}_{S}$ calculus, used as a satisfiability procedure for $(T_{LV} \cup T_{RV} \cup T_S)$, and a satisfiability procedure for $T_{\mathbb{Q}}$.

Example 2. Let us consider the function *minmax* given in Section 1. Assume we want to prove that the record r is not modified more than 'size of l' times. This amounts to prove that the formula $\psi\colon\ \forall l, r\,(r = minmax(l) \Rightarrow \mathsf{count}_R(r) \leq \mathsf{count}_L(l))$ holds. In order to prove ψ we need to prove, for the loop invariant, the formula $\beta\colon\ \forall(\gamma \Rightarrow (\mathsf{count}_R(r) \leq n - \mathsf{count}_L(l) \Rightarrow \mathsf{count}_R(r'') \leq n - \mathsf{count}_L(l')))$ where γ is the conjunction of:

$$\mathsf{rselect}_1(r) \leq \mathsf{rselect}_2(r) \qquad \mathsf{rselect}_2(r') < e \Rightarrow r'' \simeq \mathsf{rstore}_2(r', e)$$
$$e < \mathsf{rselect}_1(r) \Rightarrow r' \simeq \mathsf{rstore}_1(r, e) \qquad e \leq \mathsf{rselect}_2(r') \Rightarrow r'' \simeq r'$$
$$\mathsf{rselect}_1(r) \leq e \Rightarrow r' \simeq r \qquad l' \simeq \mathsf{cdr}(l)$$

Note that, without constraints on the initial values of r the record can be updated more than 'size of l' times. This motivates the assumption $\mathsf{rselect}_1(r) \leq \mathsf{rselect}_2(r)$. The formula β is over the signature $\Sigma_{LV} \cup \Sigma_{RV} \cup \Sigma_{\mathbb{Q}}$ and its validity can be proved by using a SMT solver modulo $T_{LV} \cup T_{RV} \cup T_S \cup T_{\mathbb{Q}}$. Applying our results about the superposition calculus $\mathcal{SP}^{\succ}_S$ together with a combination procedure for the shared theory T_S, we can obtain the necessary satisfiability procedure for $T_{LV} \cup T_{RV} \cup T_S \cup T_{\mathbb{Q}}$.

7 Conclusions

In this paper, we have identified the key notion of safe termination that allows us to prove modular termination and completeness (modulo T_C) of the superposition calculus and the combination framework, respectively. In particular, we have shown that safe termination implies modular termination and proper termination implies compatibility. In the signature-disjoint case, variable-inactivity has been initially introduced to obtain modular termination [1] but it is useful for combinability as well [12] to ensure: (1) that the bases are computable (deduction completeness) and (2) stably infiniteness. In the non-disjoint case, compatibility replaces stably infiniteness. Through these results we show an analogy between variable inactivity and safety. Roughly speaking, safety replaces variable inactivity when considering (unions of) theories sharing T_C.

The property of safe termination of saturation has to be verified for any given set of ground flat literals. Meta-superposition [13] has proved to be useful, in the disjoint case, for checking termination and variable inactivity on a single schematic form of saturation. We plan to develop a meta-superposition calculus modulo counter arithmetics to perform an automatic check of termination and safety.

A further research direction is the extension of our superposition calculus to non-convex theories (that is, non-Horn theories). In that case, the calculus would require significant changes in order to obtain completeness and an effective way to compute the bases, containing entailed *disjunctions* of s-equalities.

More generally, we are interested in extending to other shared theories our proof techniques, and also to investigate complexity issues. A possible candidate could be a different axiomatization of Integer Offsets like the one studied in [4].

References

1. Armando, A., Bonacina, M.P., Ranise, S., Schulz, S.: New results on rewrite-based satisfiability procedures. ACM Trans. Comput. Log. 10(1) (2009)
2. Armando, A., Ranise, S., Rusinowitch, M.: A rewriting approach to satisfiability procedures. Inf. Comput. 183(2), 140–164 (2003)
3. Bachmair, L., Ganzinger, H.: Rewrite-based equational theorem proving with selection and simplification. J. Log. Comput. 4(3), 217–247 (1994)
4. Bonacina, M.P., Echenim, M.: On Variable-inactivity and Polynomial T-Satisfiability Procedures. J. Log. Comput. 18(1), 77–96 (2008)
5. Bonacina, M.P., Echenim, M.: Theory decision by decomposition. J. Symb. Comput. 45(2), 229–260 (2010)
6. Bonacina, M.P., Lynch, C., de Moura, L.M.: On Deciding Satisfiability by $DPLL(\Gamma + \mathcal{T})$ and Unsound Theorem Proving. In: Schmidt [19], pp. 35–50
7. Dershowitz, N., Plaisted, D.: Rewriting. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, vol. I, ch. 9, pp. 535–610. Elsevier Science, Amsterdam (2001)
8. Enderton, H.B.: A Mathematical Introduction to Logic. Academic Press, New York (1972)
9. Ghilardi, S.: Model-theoretic methods in combined constraint satisfiability. J. Autom. Reasoning 33(3-4), 221–249 (2004)
10. Ghilardi, S., Nicolini, E., Zucchelli, D.: A comprehensive combination framework. ACM Trans. Comput. Log. 9(2) (2008)
11. Hodges, W.: Model Theory. Encyclopedia of Mathematics and its Applications, vol. (42). Cambridge University Press, Cambridge (1993)
12. Kirchner, H., Ranise, S., Ringeissen, C., Tran, D.-K.: Automatic combinability of rewriting-based satisfiability procedures. In: Hermann, M., Voronkov, A. (eds.) LPAR 2006. LNCS (LNAI), vol. 4246, pp. 542–556. Springer, Heidelberg (2006)
13. Lynch, C., Ranise, S., Ringeissen, C., Tran, D.-K.: Automatic decidability and combinability. Inf. Comput. 209(7), 1026–1047 (2011)
14. Manna, Z., Sipma, H.B., Zhang, T.: Verifying balanced trees. In: Artemov, S., Nerode, A. (eds.) LFCS 2007. LNCS, vol. 4514, pp. 363–378. Springer, Heidelberg (2007)
15. Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. ACM Trans. Program. Lang. Syst. 1(2), 245–257 (1979)
16. Nicolini, E., Ringeissen, C., Rusinowitch, M.: Combinable extensions of abelian groups. In: Schmidt [19], pp. 51–66
17. Nicolini, E., Ringeissen, C., Rusinowitch, M.: Data structures with arithmetic constraints: A non-disjoint combination. In: Ghilardi, S., Sebastiani, R. (eds.) FroCoS 2009. LNCS, vol. 5749, pp. 319–334. Springer, Heidelberg (2009)
18. Nicolini, E., Ringeissen, C., Rusinowitch, M.: Combining satisfiability procedures for unions of theories with a shared counting operator. Fundam. Inform. 105(1-2), 163–187 (2010)
19. Schmidt, R.A. (ed.): Automated Deduction – CADE-22. LNCS, vol. 5663. Springer, Heidelberg (2009)
20. Sofronie-Stokkermans, V.: Locality results for certain extensions of theories with bridging functions. In: Schmidt [19], pp. 67–83
21. Suter, P., Dotta, M., Kuncak, V.: Decision procedures for algebraic data types with abstractions. In: Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, pp. 199–210. ACM, New York (2010)

Congruence Closure of Compressed Terms in Polynomial Time

Manfred Schmidt-Schauss, David Sabel, and Altug Anis

Dept. Informatik und Mathematik, Inst. Informatik, Goethe-University,
PoBox 11 19 32, D-60054 Frankfurt, Germany,
{schauss,sabel,altug}@ki.informatik.uni-frankfurt.de

Abstract. The word-problem for a finite set of equational axioms between ground terms is the question whether for terms s, t the equation $s = t$ is a consequence. We consider this problem under grammar based compression of terms, in particular compression with singleton tree grammars (STGs) and with directed acyclic graphs (DAGs) as a special case. We show that given a DAG-compressed ground and reduced term rewriting system T, the T-normal form of an STG-compressed term s can be computed in polynomial time, and hence the T-word problem can be solved in polynomial time. This implies that the word problem of STG-compressed terms w.r.t. a set of DAG-compressed ground equations can be decided in polynomial time. If the ground term rewriting system (gTRS) T is STG-compressed, we show NP-hardness of T-normal-form computation. For compressed, reduced gTRSs we show a PSPACE upper bound on the complexity of the normal form computation of STG-compressed terms. Also special cases are considered and a prototypical implementation is presented.

Keywords: Term rewriting, grammar based compression, singleton tree grammars, congruence closure.

1 Introduction

This paper is dedicated to combining equational reasoning with grammar compression for terms. Automated deduction systems, formalizations of logical systems, systems for checking propositional logic and term rewriting systems [3,6] either are based on equational reasoning or may employ equational reasoning. The general form of equational reasoning using unrestricted sets of equational axioms is known to be very expressive, but to the price of undecidability of simple questions about derivability. A special case that leads to a decidable word problem occurs when all equational axioms are ground and thus quantifiers do not play any role. Congruence closure algorithms can solve this special kind of word problem in time $O(n \log n)$ ([20,26,13]). Extending SAT-solvers by theories leads to so-called SMT (SAT modulo theories) [21], which among other theories can also deal with equational theories defined by a set of ground equations.

Since terms in automated deduction systems may grow large during reasoning and search for a proof, compact or compressed representations of large terms

C. Tinelli and V. Sofronie-Stokkermans (Eds.): FroCoS 2011, LNAI 6989, pp. 227–242, 2011.

can be exploited to optimize the space usage, which must go hand-in-hand with specific algorithms that access the compact representations and process the compressed representation without too much decompression.

Instead of using specialized compression formats (e.g. Lempel-Ziv [29]), there are also investigations into a general mechanism: straight-line programs (SLP) for strings [23] and corresponding algorithms. An SLP here means an acyclic context free grammar, where every nonterminal has one rule, and thus can generate exactly one string. The generalization to terms is for example in [4,5] to compress XML-trees. The grammars for compressing terms were called *singleton tree grammars* (STG) [14]. STGs generalize directed acyclic graphs, since they are not limited to sharing subterms but they can also share contexts – terms with a single hole – and thus they allow to share parts of terms. For instance the grammar $A ::= C_3[B], B ::= b, C_0 ::= f([\cdot]), C_1 ::= C_0[C_0], C_2 ::= C_1[C_1], C_3 ::= C_2[C_2]$ is an STG where the nonterminal A generates the term $f(f(f(f(f(f(f(f(b)))))))))$.

Grammar-compression was also used for analyzing the complexity of unification algorithms: SLPs in [14,15] and STGs in [16,10,11]. A more general form of compression, employing terms with several holes, was shown to be polynomially equivalent to STGs [19].

A key algorithm is the equality check of two compressed strings (trees), which can be done in cubic time [17,22]. Almost all efficient algorithms on STGs use variants of the equality check of compressed words/terms. Implementations of related algorithms are described in [12].

Complexity results w.r.t. the compressed word and membership problem of confluent semi-Thue systems can be found in [18]. Note that ground TRSs on strings are like restricted semi-Thue systems, where the reduction relation is only applicable to the suffix of words.

The main result of this paper is an efficient normalization and thus an efficient solution of the word problem of STG-compressed terms w.r.t. DAG-compressed ground equations (Theorem 14). If the axioms are STG-compressed, then only partial results are obtained: For a non-confluent compressed gTRS, the question whether a term s reduces to a constant a is NP-hard (Proposition 17), and for reduced, confluent and STG-compressed gTRSs the normal form computation of STG-compressed terms is in PSPACE (Proposition 15). Also several special cases like one-rule gTRSs and monadic gTRSs are considered in the remainder of Section 4. Finally, in Section 5 a prototypical implementation, some examples, and experimental results are presented.

2 Preliminaries

In this section we briefly recall required notions and results on term rewriting systems and singleton tree grammars.

2.1 Term Rewriting Systems

A *signature* Σ is a set of function symbols, where every $f \in \Sigma$ has a fixed arity $\mathrm{ar}(f) \in \mathbb{N}_0$. Let $\mathcal{V}$ be a countably infinite set of variables. The set of *terms*

$\mathcal{T}(\Sigma,\mathcal{V})$ over the signature Σ and variables $\mathcal{V}$ is inductively defined as follows: for all $x \in \mathcal{V} : x \in \mathcal{T}(\Sigma,\mathcal{V})$; if $f \in \Sigma$ and $\mathrm{ar}(f) = 0$ then $f \in \mathcal{T}(\Sigma,\mathcal{V})$; and if $t_i \in \mathcal{T}(\Sigma,\mathcal{V})$, for $i = 1,\ldots,n$, $f \in \Sigma$, and $\mathrm{ar}(f) = n \geq 1$, then $f(t_1,\ldots,t_n) \in \mathcal{T}(\Sigma,\mathcal{V})$. With $Var(t)$ we denote the *set of variables* occurring in term t. A term $t \in \mathcal{T}(\Sigma,\mathcal{V})$ is called *ground* if $Var(t) = \emptyset$. A *substitution* σ is a mapping of variables to terms. Let $\mathrm{dom}(\sigma) = \{x \in \mathcal{V} \mid \sigma(x) \neq x\}$. The extension σ_E of σ to terms is inductively defined as $\sigma_E(x) = x$ if $x \notin \mathrm{dom}(\sigma)$, $\sigma_E(x) = \sigma(x)$ if $x \in \mathrm{dom}(\sigma)$, $\sigma_E(f(t_1,\ldots,t_n)) = f(\sigma_E(t_1),\ldots,\sigma_E(t_n))$. In the following we do not distinguish between a substitution and its extension to terms. A *context* C is a term where the special constant $[\cdot]$ (the "hole") occurs exactly once (as a subterm). The term $C[t]$ is constructed by replacing the hole in C by term t. A context C_1 is a *prefix* of context C_2 if there exists a context C_3 such that $C_1[C_3] = C_2$.

A *term rewriting system* (TRS) T is a finite set of pairs of terms $\{(l_i, r_i) \mid i = 1,\ldots,m\}$, usually written $l_i \to r_i$, where we assume that for all i: l_i is not a variable and $Var(r_i) \subseteq Var(l_i)$ (see e.g. [3]). The term rewriting relation $\xrightarrow{T}$ is defined as: $t \xrightarrow{T} t'$, if $t = C[\sigma(l_i)]$ and $t' = C[\sigma(r_i)]$ for some i, some substitution σ, and some C. The transitive and reflexive-transitive closures of $\xrightarrow{T}$ are written as $\xrightarrow{T,+}$ and $\xrightarrow{T,*}$, respectively. A term t is *T-irreducible* or a *T-normal form*, iff it cannot be further reduced using the rules of T. If the TRS is interpreted as a set of equations $E := \{l_i = r_i \mid l_i \to r_i \in T\}$, then the equality $=_E$ is the equational theory on the terms w.r.t. a signature Σ. Operationally one can define $s =_E t$ iff $s \xrightarrow{E,*} t$ by permitting the equational axioms as rewrite rules in both directions. Alternatively, $=_E$ can be defined as the smallest congruence relation with $\sigma(l_i) =_E \sigma(r_i)$ for all substitutions σ and all $i = 1,\ldots,n$. The *word problem* is to decide for given terms s,t, whether $s =_E t$. A TRS is called *terminating*, if there are no infinite reduction sequences of $\xrightarrow{T}$, and it is called *confluent* iff whenever $t_1 \xleftarrow{T,*} t \xrightarrow{T,*} t_2$, there exists a term t_3 with $t_1 \xrightarrow{T,*} t_3 \xleftarrow{T,*} t_2$. A TRS that is confluent and terminating is also called *canonical*. Canonical TRSs permit to compute unique *normal forms* of terms by rewriting them exhaustively. For a canonical TRS T the word problem is decidable by rewriting the terms s,t to their normal form and then comparing the normal forms for syntactic equality.

In this paper we are interested in ground equations and ground term rewriting systems (gTRS), i.e. when the equations in E (the rules in T, respectively) consist of ground terms. A term rewriting system T is *reduced* if every right hand side r_i is a T-normal form, and every left hand side l_i is irreducible for the system $T \setminus \{l_i \to r_i\}$. It is well-known that every reduced gTRS is canonical. It is also well-known that the word problem for ground equations E is decidable [20,26,13]. The algorithms are usually variants of the so-called congruence closure computation on term graphs. This can be computed in time $O(n \log n)$, where it is essential that DAGs are used.

In [27,28,8] it is shown that the computation of a DAG-representation of a canonical gTRS can be done in time $O(n \log n)$. The computed canonical gTRS

$T = \{l_i \to r_i \mid i = 1, \ldots, n\}$ has the additional property that it is reduced and thus T is canonical.

2.2 Grammar Compressed Terms and Term Rewriting Systems

For compression of (ground) terms we use singleton tree grammars:

Definition 1 ([14]). *A* singleton tree grammar (STG) *is a 4-tuple* $G = (\mathcal{TN}, \mathcal{CN}, \Sigma, R)$*, where* $\mathcal{TN}$ *are tree/term nonterminals, or nonterminals of arity* 0*,* $\mathcal{CN}$ *are context nonterminals, or nonterminals of arity* 1*, and* Σ *is a signature of function symbols (the terminals), such that the sets* $\mathcal{TN}$*,* $\mathcal{CN}$*, and* Σ *are pairwise disjoint. The set of nonterminals* $\mathcal{N}$ *is defined as* $\mathcal{N} = \mathcal{TN} \cup \mathcal{CN}$*. The rules in* R *may be of the form:*

- $A ::= f(A_1, \ldots, A_m)$*, where* $A, A_i \in \mathcal{TN}$ *for* $i = 1, \ldots, m$*, and* $f \in \Sigma$ *with* $\mathrm{ar}(f) = m$*.*
- $A ::= C_1[A_2]$ *where* $A, A_2 \in \mathcal{TN}$*, and* $C_1 \in \mathcal{CN}$*.*
- $C ::= [\cdot]$ *where* $C \in \mathcal{CN}$*.*
- $C ::= C_1[C_2]$*, where* $C, C_1, C_2 \in \mathcal{CN}$*.*
- $C ::= f(A_1, \ldots, A_{i-1}, [\cdot], A_{i+1}, \ldots, A_m)$*, where* $C \in \mathcal{CN}$*,* $A_j \in \mathcal{TN}$ *for* $j = 1, \ldots, i-1, i+1, \ldots m$*, and* $f \in \Sigma$ *with* $\mathrm{ar}(f) = m$*.*
- $A ::= A'$*, where* A *and* A' *are term nonterminals*

Each nonterminal X *appears as a left-hand side of exactly one rule of* R*. The transitive closure* $\xrightarrow{+}_G$ *of the relation* $\to_G$ *over* $\mathcal{N}$ *is terminating, where* $X \to_G Y$*, iff* $X ::= r$ *is a rule in* G*, and* $Y \in \mathcal{N}$ *occurs in* r*. The term (or context) generated by a nonterminal* N *of* G*, denoted by* $\mathrm{val}_G(N)$ *or* $\mathrm{val}(N)$ *when* G *is clear from the context, is the term (or context) over* Σ *reached from* N *by successive and exhaustive applications of the rules of* G*. More rigorously:*

$$
\begin{array}{ll}
\mathrm{val}_G(A) = f(\mathrm{val}_G(A_1), \ldots, \mathrm{val}_G(A_m)), & \text{if } A ::= f(A_1, \ldots, A_m) \\
\mathrm{val}_G(A) = \mathrm{val}_G(C_1)[\mathrm{val}_G(A_2)] & \text{if } A ::= C_1[A_2] \\
\mathrm{val}_G(A) = \mathrm{val}_G(A') & \text{if } A ::= A' \\
\mathrm{val}_G(C) = [\cdot] & \text{if } C ::= [\cdot] \\
\mathrm{val}_G(C) = \mathrm{val}_G(C_1)[\mathrm{val}_G(C_2)] & \text{if } C ::= C_1[C_2] \\
\mathrm{val}_G(C) = f(t_1, \ldots, t_{i-1}, [\cdot], t_{i+1}, \ldots, t_m) & \\
\qquad \text{if } C ::= f(A_1, \ldots, A_{i-1}, [\cdot], A_{i+1}, \ldots, A_m) \text{ and} & \\
\qquad t_j = \mathrm{val}_G(A_j) \text{ for } j = 1, \ldots, i-1, i+1, \ldots, m &
\end{array}
$$

The cdepth *of a context nonterminal* D *is the maximal* n *of all sequences* $D \to_G D_1 \to_G D_2 \ldots \to_G D_n$*, where only rules of the form* $C ::= C_1[C_2]$ *are taken into account, and the cdepth of the STG* G *is defined as the maximum of all cdepths of nonterminals. The* size $|G|$ *of a grammar* G *is the sum of the sizes of all right hand sides, where the hole* $[\cdot]$ *counts as 1. If the arity of function symbols is* $O(1)$*, then we could have also used the number of the rules of* G*.* □

Note that for every nonterminal N of G, the term or context $\mathrm{val}_G(N)$ is defined.

Example 2. For $n \in \mathbb{N}$ let G_n be the STG $(\{A, B\}, \{C_0, \ldots, C_n\}, \{a, f\}, R)$ where R is the following set of productions:
$A ::= a$; $B ::= C_n[A]$; $C_0 ::= f([\cdot])$; $C_{i+1} ::= C_i[C_i]$ for $i = 0, \ldots, n-1$.
Then $\mathrm{val}_{G_n}(B) = f^{(2^n)}(a)$ and $|G_n| = 2n + 5$. The cdepth of C_i is i.

Proposition 3 ([4,24,5,17,22]). *For an STG G and two term nonterminals A_1, A_2 it can be decided in $O(|G|^3)$ whether $\mathrm{val}_G(A_1) = \mathrm{val}_G(A_2)$ holds.*

Directed acyclic graphs (DAGs) can be represented by STGs. Since DAGs only share subterms, context nonterminals must not be used to represent DAGs.

Definition 4. *A* DAG *G is an STG where $\mathcal{CN} = \emptyset$. A DAG is called* optimally compressed*, if for nonterminals $A, B : A \neq B \implies \mathrm{val}(A) \neq \mathrm{val}(B)$.*

Note that for every non-optimal DAG there exists either a production $A_1 ::= A_2$ or at least two productions $A_1 := r$ and $A_2 ::= r$. For the following complexity analyses and lemma we assume that the signature is fixed, and that the arity of function symbols is $O(1)$.

Lemma 5 ([20,26,13]). *A DAG G can be transformed into an optimally compressed DAG in time $O(|G| \cdot \log(|G|))$. The size is not increased by this operation.*

Proof. Though this appears to be well-known, we give a sketch: First we use topological sorting to produce in time $O(|G|)$ a list L of nonterminals of G where $A > A'$, if $A \xrightarrow{+}_G A'$. We operate on the list in reverse order. Assume that we construct the optimally compressed grammar from G by scanning the list. During the reconstruction, two data structures are used: (i) a data base with keys $f(A_1, \ldots, A_n)$, nonterminals as entries, and $O(\log m)$ access time if the data base has m entries; (ii) a function *on* mapping nonterminals to their optimal node. Let A be the current nonterminal, G' be the constructed new DAG and G'' be the remaining rules of the grammar G. For the current given nonterminal A with rule $A ::= f(A_1, \ldots, A_k)$, there are two cases: (i) If there is an entry A' under key $f(on(A_1), \ldots, on(A_k))$, then define $on(A) := A'$, and remove the rule for A. (ii) If there is no entry under key $f(on(A_1), \ldots, on(A_k))$, then define $on(A) := A$, and let the new rule be $A ::= f(on(A_1), \ldots, on(A_k))$ and insert A in the data base with key $f(on(A_1), \ldots, on(A_k))$.

Finally, the start symbol can be replaced using function *on*. Since the sum of the number of nonterminals of G' and G'' is at most $|G|$, the time per nonterminal is $O(\log(|G|))$, and the list is also of length $O(|G|)$, hence this can be done in time $O(|G| \cdot \log(|G|))$. The size of the DAG is not increased. □

We say a term t is *STG-compressed* (or *DAG-compressed*, respectively), if there is an STG (or a DAG, respectively) G and a term-nonterminal A of G such that $\mathrm{val}_G(A) = t$. A TRS T is called *STG-compressed* (or *DAG-compressed*, respectively), if all terms l_i, r_i of the rewriting relation are represented by term-nonterminals L_i, R_i of an STG (or a DAG, respectively) G such that $\mathrm{val}_G(L_i) = l_i$ and $\mathrm{val}_G(R_i) = r_i$. We use the analogous notions also for sets of equations.

3 The Word-Problem for STG-Compressed Terms with DAG-Compressed Ground Equations

In this section we show that the word problem for ground equations and STG-compressed terms s, t can be solved in polynomial time w.r.t. the size of the compressed representation.

3.1 A Normalizing Algorithm for Compressed Terms

First we describe an algorithm for T-normalizing all terms represented by term nonterminals in an STG, where T is a given reduced, canonical gTRS. The idea is to modify the STG such that there is no reducible subterm of any represented term in the STG, for any nonterminal.

We assume that a reduced, canonical TRS T is given and that T is optimally DAG-compressed, i.e. the terms l_i and r_i for $i = 1, \ldots, m$ are represented in the (optimally compressed) DAG $G_T = (\mathcal{TN}_T, \emptyset, \Sigma_T, R_T)$, such that there are nonterminals $L_i, R_i \in \mathcal{TN}_T$ with $\mathrm{val}_{G_T}(L_i) = l_i$ and $\mathrm{val}_{G_T}(R_i) = r_i$ for $i = 1, \ldots, n$. For convenience we sometimes write $T = \{L_1 \to R_1, \ldots, L_m \to R_m\}$ for the TRS. Note that the nonterminals $R_1, \ldots, R_m$ are not necessarily distinct. We assume that the to-be-normalized terms are STG-compressed, i.e. there is an input STG $G_{inp} = (\mathcal{TN}_{inp}, \mathcal{CN}_{inp}, \Sigma_{inp}, R_{inp})$, such that $\mathcal{TN}_{inp} \cap \mathcal{TN}_T = \emptyset$. Let G be the union of G_T and G_{inp}.

For the TRS T and its corresponding DAG G_T we define the sets $\mathrm{subterms}_{\mathrm{NT}}(T)$ and $\mathrm{subterms}(T)$ as follows, where $\top$ is an extra symbol:

$$\begin{aligned}\mathrm{subterms}_{\mathrm{NT}}(T) &:= \{A \mid L_i \xrightarrow{+}_{G_T} A,\ i = 1, \ldots, m\} \cup \{R_i \mid i = 1, \ldots, m\} \cup \{\top\}\\ \mathrm{subterms}(T) &:= \{\mathrm{val}(A) \mid A \in \mathrm{subterms}_{\mathrm{NT}}(T) \setminus \{\top\}\}\end{aligned}$$

The set $\mathrm{subterms}_{\mathrm{NT}}(T)$ comprises all nonterminals that are referenced by left hand sides L_i of T, the nonterminals R_i, and a distinguished constant $\top$. Every proper subterm of a left-hand side l_i is represented by one nonterminal in $\mathrm{subterms}_{\mathrm{NT}}(T)$, a nonterminal for every right-hand side r_i is in $\mathrm{subterms}_{\mathrm{NT}}(T)$, and $\top$ represents the other terms.

Note that $\{L_i \mid i = 1, \ldots, m\} \cap \mathrm{subterms}_{\mathrm{NT}}(T) = \emptyset$, and that for every $A \in \mathrm{subterms}_{\mathrm{NT}}(T) \setminus \{\top\}$ the term $\mathrm{val}(A)$ is T-irreducible, since T is reduced.

The algorithm below will modify the grammar G bottom-up, where only the rules of G_{inp} as a sub-STG of G are modified by replacing the right hand sides of the grammar rules, and also some rules for constructing context-nonterminals will be added. For the term nonterminals of G_{inp} the resulting STG G' will only represent normalized terms, i.e. for every nonterminal A of G_{inp}: $\mathrm{val}_{G'}(A)$ is the T-normal form of $\mathrm{val}_G(A)$. The following algorithms are designed to avoid the Plandowski-equality check (Proposition 3) and perform all checks for equality either on the name of the symbols or checking in the DAG, which requires that the DAG for T is optimally compressed.

First we define an algorithm that computes two functions ϕ_0, ϕ_1 such that:

- For every $A \in \mathcal{TN}_{inp}$, we have $\phi_0(A) \in \mathrm{subterms}_{\mathrm{NT}}(T) \subseteq \mathcal{TN}_{inp}$. If $\mathrm{val}(A) \in \mathrm{subterms}(T)$ then $\phi_0(A) \in \mathrm{subterms}_{\mathrm{NT}}(T) \setminus \{\top\}$, and the term $\mathrm{val}(\phi_0(A))$ will be the normal form of $\mathrm{val}(A)$. Otherwise, $\phi_0(A)$ will be $\top$.
- For every $C \in \mathcal{CN}_{inp}$, we have $\phi_1(C) : \mathrm{subterms}_{\mathrm{NT}}(T) \to \mathrm{subterms}_{\mathrm{NT}}(T)$. This function computes the mapping behavior of the context $\mathrm{val}(C)$ on $\mathrm{subterms}(T)$ after normalization: For every nonterminal A in $\mathrm{subterms}_{\mathrm{NT}}(T)$, if the T-normal form of $\mathrm{val}(C)[\mathrm{val}(A)]$ is in $\mathrm{subterms}(T)$, then $\phi_1(C)(A) \in \mathrm{subterms}_{\mathrm{NT}}(T) \setminus \{\top\}$ and $\mathrm{val}(\phi_1(C)(A))$ is the T-normal form of $\mathrm{val}(C)[\mathrm{val}(A)]$, otherwise $\phi_1(C)(A)$ will be $\top$.

The following subalgorithm dagNode is required, which computes for nonterminals A_i from G_T the node in G_T for $f(A_1, \ldots, A_n)$ if it exists. Formally, for $A_i \in \mathcal{TN}_T \cup \{\top\}$ and a function symbol $f \in \Sigma_T \cup \Sigma_{inp}$ of arity n let $\mathrm{dagNode}(f, A_1, \ldots, A_n)$ be defined as follows:

$$\mathrm{dagNode}(f, A_1, \ldots, A_n) := \begin{cases} N, \text{ if } N ::= f(A_1, \ldots, A_n) \in R_T \\ \bot, \text{ otherwise} \end{cases}$$

Algorithm 6 (The ϕ-Computation Algorithm). The algorithm incrementally computes ϕ_0 and ϕ_1 by inspecting the production rules of G_{inp} in bottom-up order, i.e. in the order reverse to $\to_{G_{inp}}$.

The treatment of production rules is defined by a case analysis:

1. $A ::= f(A_1, \ldots, A_n)$, with $n \geq 0$. If $\mathrm{dagNode}(f, \phi_0(A_1), \ldots, \phi_0(A_n)) = \bot$ then define $\phi_0(A) := \top$.
   Otherwise, let $\mathrm{dagNode}(f, \phi_0(A_1), \ldots, \phi_0(A_n)) = N$ and define
   $$\phi_0(A) := \begin{cases} R_i, \text{ if } N = L_i \text{ for some left hand side of a rule } L_i \to R_i \\ N, \text{ if } N \neq L_i \text{ and } N \in \mathrm{subterms}_{\mathrm{NT}}(T) \\ \top, \text{ otherwise} \end{cases}$$
2. $C ::= f(A_1, \ldots, A_{i-1}, [\cdot], A_{i+1}, \ldots, A_n)$. We compute the function $\phi_1(C)$ for all arguments $B \in \mathrm{subterms}_{\mathrm{NT}}(T)$:
   The first case is $\phi_0(A_j) = \top$ for some $j \neq i$. Then $\phi_1(C)(B) := \top$ for all $B \in \mathrm{subterms}_{\mathrm{NT}}(T)$.
   The other case is $\phi_0(A_j) \neq \top$ for all $j \neq i$. Let $\phi_1(C)(B) := \top$, if $B = \top$ or $\mathrm{dagNode}(f, \phi_0(A_1), \ldots, \phi_0(A_{i-1}), B, \phi_0(A_{i+1}), \ldots, \phi_0(A_n)) = \bot$. Otherwise, let $N := \mathrm{dagNode}(f, \phi_0(A_1), \ldots, \phi_0(A_{i-1}), B, \phi_0(A_{i+1}), \ldots, \phi_0(A_n))$ and define:
   $$\phi_1(C)(B) := \begin{cases} R_i, \text{ if } N = L_i \text{ for some left hand side of a rule } L_i \to R_i \\ N, \text{ if } N \neq L_i \text{ and } N \in \mathrm{subterms}_{\mathrm{NT}}(T) \\ \top, \text{ otherwise} \end{cases}$$
3. $A_1 ::= A_2$. Then define $\phi_0(A_1) := \phi_0(A_2)$.
4. $C ::= [\cdot]$. Then $\phi_1(C)$ is the identity function on $\mathrm{subterms}_{\mathrm{NT}}(T)$.
5. $C ::= C_1[C_2]$. Since the algorithm proceeds bottom-up, we already have computed the functions $\phi_1(C_1), \phi_1(C_2)$, and so we can compute the function $\phi_1(C)$ as the composition $\phi_1(C_1) \circ \phi_2(C_2)$.
6. $A ::= C[B]$. Then $\phi_0(A) := \phi_1(C)(\phi_0(B))$.

Lemma 7. *Let G be an STG and T be a reduced and confluent gTRS. Then for every $A \in \mathcal{TN}_{inp}$ the following holds: If $\phi_0(A) = \top$, then* $\mathrm{val}(A)$ *is not a subterm of any left hand side of T. Consequently, there are no superterms of* $\mathrm{val}(A)$ *that are redexes. This also holds during the whole reduction process, hence also the T-normal form s of* $\mathrm{val}(A)$ *is not a subterm of any left hand side of T.*

Example 8. We consider the TRS T with one rule $f(f(a)) \to a$ represented by $\{L_1 \to R_1\}$ where the corresponding DAG G_T has the production rules $L_1 ::= f(F),\ F ::= f(R_1),\ R_1 ::= a$. Then $\mathrm{subterms}_{\mathrm{NT}}(T) = \{F, R_1, \top\}$. We consider the STG of Example 2 and compute ϕ_0, ϕ_1 as follows:

$$\phi_0(A) = R_1 \quad \left| \begin{array}{l} \phi_1(C_0)(\top) = \top \\ \phi_1(C_0)(F) = R_1 \\ \phi_1(C_0)(R_1) = F \end{array} \right. \quad \left| \begin{array}{l} \phi_1(C_1)(\top) = \top \\ \phi_1(C_1)(F) = \phi_1(C_0)(\phi_1(C_0)(F)) = F \\ \phi_1(C_1)(R_1) = \phi_1(C_0)(\phi_1(C_0)(R_1)) = R_1 \end{array} \right.$$

For $i = 1, \ldots, n-1$ we have $\phi_1(C_{i+1}) = \phi_1(C_i) \circ \phi_1(C_i)$ and thus $\phi_1(C_i)$ is the identity on $\mathrm{subterms}_{\mathrm{NT}}(T)$ for $i \geq 1$. Finally, we can compute $\phi_0(B) = \phi_1(C_n)(\phi_0(A)) = \phi_1(C_n)(R_1) = R_1$.

The normalization algorithm uses the functions ϕ_0 and ϕ_1 to compute an STG that represents all T-normal forms of terms represented by G_{inp}. Usually, it only changes productions for term nonterminals of the input grammar. The difficult case is a production of the form $A ::= C[B]$ where a subterm of $\mathrm{val}(C)[\mathrm{val}(B)]$ which is a proper superterm of $\mathrm{val}(B)$, is indicated by ϕ_0 as having a normal form representable by some nonterminal A' in $\mathrm{subterms}_{\mathrm{NT}}(T)$, and which is maximal. Then a new context nonterminal C' used to generate the normal form of $\mathrm{val}(C)[\mathrm{val}(B)]$ as $\mathrm{val}(C')[\mathrm{val}(A')]$ must be found and added to the grammar.

Algorithm 9 (Normalization Algorithm). The algorithm has $G = G_{inp} \cup G_T$ as input and uses the (algorithmically defined) functions ϕ_0, ϕ_1. It iterates over all productions of G_{inp} and modifies them according to the following cases.

1. The rules for context nonterminals are unchanged.
2. If the rule is $A_1 ::= A_2$, then the rule is unchanged.
3. If the rule is $A ::= f(A_1, \ldots, A_n)$, and $\phi_0(A) \neq \top$, then replace this rule by $A ::= \phi_0(A)$. If $\phi_0(A) = \top$, do not change the rule.
4. Let the rule be $A ::= C[B]$. Then
   (a) If $\phi_0(B) = \top$, then do not change the rule.
   (b) If $\phi_0(A) \neq \top$, then replace the rule for A by $A ::= \phi_0(A)$.
   (c) If $\phi_0(B) \neq \top$, but $\phi_0(A) = \top$, then the normalization stops somewhere between $\mathrm{val}(C[B])$ and $\mathrm{val}(B)$. The compressed normal form of $\mathrm{val}_G(C[B])$ is constructed as follows: we construct a context nonterminal C' such that $\mathrm{val}_G(C')$ is a prefix of $\mathrm{val}_G(C)$, and have to find a term nonterminal $B' \in \mathrm{subterms}_{\mathrm{NT}}(T) \setminus \{\top\}$ such that $C'[B']$ represents the normal form of $\mathrm{val}(C[B])$ and $\mathrm{val}(C'[B'])$ is irreducible. This is done top-down starting from C by the algorithm *Prefix* defined below. We perform $\mathit{Prefix}(C, \phi_0(B))$ (that may add rules for context nonterminals as a side-effect) and obtain a result (C', D), which we use to replace the rule for A by the rule $A ::= C'[D]$.

Now we describe *Prefix*(C, D), where we assume that $D \in \text{subterms}_{\text{NT}}(T) \setminus \{\top\}$. The cases are:

1. $C ::= [\cdot]$ is the rule for C, then return (C, D).
2. $C ::= f(A_1, \ldots, A_{i-1}, [\cdot], A_{i+1}, \ldots, A_n)$ is the rule for C. If $\phi_1(C)(D) \neq \top$, then return $([\cdot], \phi_1(C)(D))$. Otherwise, return (C, D).
3. $C ::= C_1[C_2]$ is the rule for C. Then there are two cases.
   (i) If $\phi_1(C_2)(D) = \top$, then let (C_2', D_2') be the result of *Prefix*(C_2, D). Construct $C' ::= C_1[C_2']$ in the grammar and return (C', D_2').
   (ii) If $\phi_1(C_2)(D) = D' \neq \top$ then return the result of *Prefix*(C_1, D').

Example 10. We again consider Example 2 and the gTRS of Example 8. The normalization algorithm produces the grammar G_{out} with the productions:

$$\begin{array}{llll} L_1 ::= f(F) & F ::= f(R_1) & R_1 ::= a & C_{i+1} ::= C_i[C_i] \\ A ::= R_1 & B ::= R_1 & C_0 ::= f([\cdot]) & \text{for } i = 0, \ldots, n-1 \end{array}$$

As expected we have $\text{val}_{G_{out}}(B) = a$.

Lemma 11. *Algorithm 9 is correct: It computes a new STG* $G' := G'_{inp} \cup G_T$*, where for every term nonterminal* A *of* G_{inp}*:* $\text{val}_G(A) =_T \text{val}_{G'}(A)$*, and* $\text{val}_{G'}(A)$ *is* T*-irreducible.*

Proof. (sketch) It is easy to verify that every modification in the grammar retains equality w.r.t. the equational theory $=_T$. It is also straightforward to check that the only irreducible expressions are represented by the STG after the normalization process has finished. □

Now we estimate the complexity of the normalization algorithm where we use $|G|$, $|G_{inp}|$, and $|G_T|$ as parameters and where we assume that $|\Sigma|$, as well as the arity of function symbols is $O(1)$.

Lemma 12. *The normalization increases the grammar* G *by at most* $O(|G|^2)$ *and requires time* $O(|G|^2 + |G| \cdot |G_T| \cdot \log(|G_T|))$.

Proof. The cardinality of $\text{subterms}_{\text{NT}}(T)$ is at most $|G_T|$ since every subterm of left hand sides and every right hand side is represented by a node in the DAG. The grammar is increased during the construction as follows. There is no cdepth-increase of context nonterminals during constructing the prefix of context nonterminals. *Prefix* returns a context and a term nonterminal. The only possibility for the returned term nonterminal for *Prefix* is the input term nonterminal, or a term nonterminal from $\text{subterms}_{\text{NT}}(T) \setminus \{\top\}$. The size increase by one normalization step is at most $\text{cdepth}(G_{inp})$, since the size increase by *Prefix* depends only on the cdepth of context nonterminals. Thus the size increase is $|G_{inp}| \cdot \text{cdepth}(G_{inp})$, which is $O(|G|^2)$.

Concerning the running time, note that all equality comparisons only require to compare nonterminals from $\text{subterms}_{\text{NT}}(T)$, since other comparisons are prevented by $\top$. Hence a single comparison can be done in constant time.

Assuming that ϕ_0, ϕ_1 are stored in an efficient data structure, the computation of the function ϕ_1 corresponding to the context nonterminals requires time $|G_T| \cdot \log(|G_T|)$, which has to be done for every context nonterminal of the initial grammar, thus it requires time $O(|G_{inp}| \cdot |G_T| \cdot \log(|G_T|))$. □

3.2 Deciding the Word Problem

Using the construction in the last subsection, we show how to decide the equality of two STG-compressed terms s_1, s_2 given a set of DAG-compressed ground equations, which is more general than considering plain equations. The following steps provide such a decision algorithm:

The input is a DAG G_E and equations $L_1 = R_1, \ldots, L_n = R_n$, where $\mathrm{val}(L_i)$, $\mathrm{val}(R_i)$ are ground (and all the symbols L_i, R_i are different) and an STG G that represents the terms s_1, s_2 by the nonterminals S_1, S_2.

1. Compute a DAG G_T that represents a reduced gTRS T which is equivalent to G_E using Snyder's algorithm ([27,28,8]). Note that Snyder's algorithm can also be used for DAG-compressed ground equations.
2. Optimally compress the DAG G_T using Lemma 5.
3. Use the algorithm in the previous subsection to construct an STG G' that represents the STG-compressed normal forms of all term nonterminals, in particular the normal forms of s_1, s_2 by the nonterminals S_1, S_2.
4. Use the Plandowski-Lifshits algorithm (Proposition 3) to decide whether S_1, S_2 represent the same terms.

Lemma 13. *[27,28] For a set of ground equations E represented by a DAG G_E (with different symbols for the terms in the equations) one can compute a reduced gTRS T, with $=_T \; = =_E$, represented by a DAG G_T in time $O(|G_E| \cdot \log^2 |G_E|)$ where $|G_T| = O(|G_E|)$.*

Proof. We analyze the steps of the algorithm in [28]: The first step is to generate a DAG for a given set of ground equations E. This step is not necessary for our claim, since E is already represented by G_E. All other steps in the algorithm of [28] are performed on the DAG, and the dominating cost is computing the congruence closure, which can either be done in time and space $O(|G_E| \cdot \log |G_E|)$ or in time $O(|G_E| \cdot \log^2 |G_E|)$ and $O(|G_E|)$ space [7]. □

Using this result and the previous results on normalization we obtain:

Theorem 14. *Given a set of ground equations E, represented by a DAG G_E (with different symbols for the terms in the equations), and two terms s_1, s_2 represented by nonterminals S_1, S_2, respectively, of an STG G_{inp}, a reduced, canonical gTRS T, equivalent to E, and the STG G' representing the T-normal forms of S_1, S_2 can be computed in time $O(|G|^2 + |G| \cdot |G_E| \cdot \log |G_E| + |G_E| \cdot \log^2 |G_E|)$, and $\mathrm{val}(S_1) =_E \mathrm{val}(S_2)$ can be decided in time $O(|G|^6 \cdot \log^3 |G|)$, where $G = G_{inp} \cup G_E$.*

Proof. The construction of the gTRS can be done in time $O(|G_E| \cdot \log^2 |G_E|)$ and space $O(|G_E|)$. The STG G' can be computed in time $O(|G|^2 + |G| \cdot |G_E| \cdot \log |G_E|)$, which results in time $O(|G|^2 + |G| \cdot |G_E| \cdot \log |G_E| + |G_E| \cdot \log^2 |G_E|)$.

Since $G_E \subset G$, the estimation is $O(|G|^2 \cdot \log |G|)$ for the time of the construction, and $O(|G|^6 \cdot \log^3 |G|)$ to perform the equality decision using the Plandowski-Lifshits-algorithm (Proposition 3). □

4 STG-Compressed Ground Term Rewriting Systems

If the ground TRS is STG-compressed, then the normalization algorithms become more involved if we want efficient ones. It is obvious that there is an exponential upper bound on the running time for normalization and the word problem, since after decompression, which increases the size at most exponentially to $2^{|G|}$, we can use the well-known algorithms with $O(n \cdot \log(n))$ running time. In the following we look for improved bounds in special cases.

4.1 Complexity Bounds

Proposition 15. *Given a reduced, confluent gTRS T, represented as $T = \{L_1 \to R_1, \ldots, L_n \to R_n\}$ where L_i, R_i are from an STG G_T. Let s be a term with $\mathrm{val}(S) = s$ where S is a term nonterminal from the STG G. Then the T-normal form of s is computable in polynomial space depending on $|G| + |G_T|$.*

Proof. We show that there is a reduction sequence $s \to s_1 \to \ldots \to s_k \xrightarrow{*} s_n$ where s_n is the T-normal form of s, and where for every k the STG G_k representing s_k requires polynomial space. The claim is that for every k: G_k can be directly derived from G as follows:

- G_k contains the rules of G_T as well as (perhaps modified) rules of G, plus perhaps some additional rules.
- Some term nonterminals in right hand sides of the G-rules may be replaced by R_i for some i.
- Some right hand sides of G-rules of the form $C[A]$ are replaced by $C'[R_i]$, for some i, where $\mathrm{val}(C')$ is a prefix of $\mathrm{val}(C)$. G is extended by the rules generating C'.

Since prefixes of contexts can be generated by an at most polynomial enlargement of the grammar, and the prefixes can be added independently, the size of the STG G_k is at most polynomial in the size of G.

Note that this construction cannot be turned into an efficient algorithm, since the justifications where to replace would require the whole rewrite sequence $s \to s_1 \to \ldots \to s_k$.

It remains to show that G_{k+1} can be derived from G_k by a parallel rewriting step: If the rewriting replaces a term nonterminal in G_k, then the replacement constructs a grammar that can be immediately derived from G.

If the rewriting replaces a subterm of some right hand side $C'[R]$, then the only possibility is that a context nonterminal C'' has to be constructed such

that val(C'') is a prefix of val(C') and the right hand side $C'[R]$ is replaced by $C''[R']$, where R' is a right hand side of a rule in T. Hence the algorithm runs in polynomial space. □

Corollary 16. *Given a reduced, confluent gTRSs T, STG-compressed by G_T and two terms s, t also STG-compressed by G, and let $=_T$ be the equality relation derived from T. Then the word problem, i.e. whether $s =_T t$, is in PSPACE.*

We do not know more efficient algorithms for simplifying a compressed term by a reduced gTRS, or for making a compressed gTRS reduced. Determining the exact complexity of the word problem of STG-compressed terms w.r.t. a set of STG-compressed ground equational axioms is left for future research.

Proposition 17. *Let $\Sigma = \{f, b_1, \ldots, b_n, b_{n+1}\}$ be a signature (n a positive integer) where f is unary and b_i are constants. Given a gTRSs T over Σ, compressed by an STG G_T, such that the right hand sides of rewriting rules are constants from Σ, and a term s also compressed by an STG G, then the problem whether s has b_{n+1} as a normal form under T is NP-hard.*

Proof. We adapt the proofs in [23] for our specific problem. We use positive SUBSETSUM as an NP-hard problem ([9]). Given n (positive) integers $S := \{a_1, \ldots, a_n\}$, and another integer m. Then the question is whether there there is a subset $S' \subseteq S$, such that $\sum_{a \in S'} a = m$.

The uncompressed TRS T is constructed as follows: It has rules of the forms $f^{a_i}(b_i) \to b_{i+1}$ and $b_i \to b_{i+1}$ for $i = 1, \ldots, n$, and the term s is of the form $f^m(b_1)$. These terms can easily be compressed in polynomial space. The question is whether s can be reduced to b_n using the rules of T. Such a reduction corresponds to a sum as in SUBSETSUM. Hence the problem is NP-hard. □

Remark 18. Proposition 17 does not show that the compressed word-problem w.r.t. a ground equational theory is NP-hard. For example, the equational theory of the encoding in Proposition 17 can be decided in polynomial time: It can be reduced to the axiom $f^k(b_{n+1}) =_T b_{n+1}$, where k is the greatest common divisor of all the numbers a_i, and the axioms $b_i = b_{n+1}$. Then the word problem can be decided in polynomial time by a computation modulo k.

4.2 STG-Compressed gTRS with One Rule

We consider the problem of normalizing an STG-compressed term t using a single STG-compressed ground rule $L \to R$, where L does not occur in R, and hence for deciding the word problem w.r.t. $L \to R$.

A naive method is to perform step-by-step normalization. One step is to find all positions of val(L) in val(t), constructing the corresponding nonterminals and replacing them by a reference to R. Such a single step can be done in polynomial time. In general, this will lead to an exponential number of normalization steps: Let the term be $f^n(a)$, and the rewrite rule be $f^{m+1}(a) \to f^m(a)$, where $n > m$ are large numbers. Since $f^n(a)$ can be represented in an STG of size $\log(n)$, and

since every rewrite step only reduces the exponent by 1, there will be $n - m$ rewrite steps during normalization, which may be exponentially large in $|G|$. For the following special cases normalization can be performed in polynomial time:

- If $\mathrm{val}(L)$ has no occurrences of $\mathrm{val}(R)$, then replacing all occurrences of L by R is sufficient. These may be explicit occurrences, when $\mathrm{val}(L) = \mathrm{val}(A)$ for a term nonterminal, or implicit occurrences, when $\mathrm{val}(L)$ occurs in $\mathrm{val}(C[A])$ as a subterm between A and $C[A]$. In this case there is only one such possibility, which can easily be constructed.
- For the (nontrivial) case that $\mathrm{val}(R)$ occurs exactly once in $\mathrm{val}(L)$ the occurrence can be found by ground submatching, also the representation $L'[R]$ for L such that $\mathrm{val}(L'[R]) = \mathrm{val}(L)$ with a context nonterminal L' can be easily constructed. The computation of a representation of all occurrences of the context L' in some C is in [25]. More exactly, the occurrences of the form $L'^n[R]$ with a maximal n have to be determined. Using the context-in-context table of [25], and using binary search for the maximal n, a polynomial algorithm can be constructed for this task. If the occurrences are found, (at most one per term nonterminal), then we can replace these occurrences by R, and obtain a normalized term t' for t.

4.3 Monadic Ground Term Rewriting Systems

We investigate the special case of monadic signatures $\Sigma := \{f_1, \ldots, f_m, a\}$ consisting only of unary function symbols f_i and a single constant. Assume a compressed and reduced (i.e., a confluent and terminating) gTRS $\{L_i \to a \mid i = 1, \ldots, n\}$ over monadic Σ. We compute the nonterminals $A_{i,j}$, and the corresponding rules, such that whenever $\mathrm{val}(R_i)$ occurs in $\mathrm{val}(L_j)$, then $\mathrm{val}(A_{i,j}(R_i)) = \mathrm{val}(L_j)$. Given a term s, the rewriting process identifies a left hand side $\mathrm{val}(L_i)$ occurring in s, and rewrites this to $\mathrm{val}(R_i)$. Since the signature is monadic, this can be interpreted as rewriting a string where every rewrite must replace a suffix. Since the gTRS is reduced, the rewriting process can also be seen as a computation that acts like a deterministic finite automaton: For instance, let $s = \mathrm{val}(A_{2,5}(A_{3,2}(A_{1,3}(R_1))))$. Then $\mathrm{val}(A_{1,3}(R_1)) = \mathrm{val}(L_3)$, hence it proceeds as $\mathrm{val}(A_{2,5}(A_{3,2}(R_3)))$. The next reduction steps are $\mathrm{val}(A_{2,5}(A_{3,2}(R_3))) = \mathrm{val}(A_{2,5}(L_2)) \to \mathrm{val}(A_{2,5}(R_2) = \mathrm{val}(L_5) \to \mathrm{val}(R_5)$. Since the TRS is confluent and reduced, the computation is deterministic. Translating this into a DFA-computation: the starting state is i, where L_i is the left hand side occurring in s, and the next state depends on the symbol $A_{i,j}$. We can also add an initial step $\varepsilon \to R_i$, where we can omit ambiguous steps. i.e. if an $\mathrm{val}(R_i)$ is a proper suffix of $\mathrm{val}(R_j)$, then we can omit this step in the automata. Every state of the DFA is accepting.

On the other hand, every DFA where all states are accepting can be interpreted as such a TRS: Let all $\mathrm{val}(R_i)$ be trivial, and let the left hand sides be 1, 10, 100, 1000, ... Then the TRS is reduced, and the question whether a compressed string can be reduced to ε can be solved looking at the DFA.

This implies:

Table 1. Test series 1

n	time (sec)
50 000	2.15
100 000	5.77
250 000	20.79
500 000	61.91
1 000 000	221.66

Table 2. Test series 2

n	m	time (sec)
5 000	1 000	6.03
5 000	2 000	11.89
5 000	5 000	38.75
5 000	10 000	93.85
100 000	1 000	649.13

Table 3. Test series 3

n	k	time (sec)
5 000	1 000	6.06
5 000	2 000	13.20
5 000	5 000	36.65
5 000	10 000	81.19
100 000	1 000	666.34

Lemma 19. *The question whether a term over a monadic signature can be reduced to a is polynomially equivalent to the question: given a DFA and a compressed word s, is there is a word w over context nonterminals accepted by the DFA such that* $\mathrm{val}(w) = \mathrm{val}(s)$.

However, note that for small $\mathrm{val}(A_{i,j})$, i.e. if $\mathrm{val}(A_{i,j})$ can be viewed as part of the input, there is a polynomial algorithm to solve this problem by using dynamic programming over the compressions of s. Thus the open question is the complexity of these problems for arbitrary $\mathrm{val}(A_{i,j})$.

5 Implementation and Tests

We implemented the normalization algorithm and the Plandowski-Lifshits equality check in the lazy functional programming language Haskell [1]. STGs are implemented as maps (available by the Haskell library `Data.Map`) where the right hand side of a production is mapped to its left hand side. Haskell's maps are based on size balanced binary trees [2] and provide selection and construction operations, like lookup, insertion, or deletion, in logarithmic time which makes our prototypical implementation reasonably fast. Including some example grammars and term rewriting systems our prototypical implementation consists of about 2000 lines of Haskell source code. A cabal[1] package is available under http://www.ki.informatik.uni-frankfurt.de/research/gbc/.

We performed several tests[2] on the grammar of Example 2 as G_{inp}.

For the first series of tests we used the gTRS of Example 8. Table 1 shows the runtime of the normalization algorithm for different values of n. We observe that the runtime grows by increasing n. Nevertheless our algorithm performs fast, since the gTRS has only one rule and the corresponding DAG is small.

The second series of tests uses TRSs with a single rule of the form $f^m(a) \to a$. The corresponding DAG is $G_T = \{\{L_1, A_1, \ldots, A_{m-1}, R_1\}, \emptyset, \{f, a\}, R\}$ where $R = \{L_1 ::= f(A_1), A_1 ::= f(A_2), \ldots, A_{m-1} ::= f(R_1), R_1 ::= a\}$. Table 2 shows runtimes for different m, n. Note that $|\mathrm{subterms}_{\mathrm{NT}}(T)|$ is much larger than in the first series of tests.

The third series of tests concerns a growing number of rules of the TRS. We used TRSs with k rules $f(a) \to b_1, f(b_1) \to b_2, \ldots f(b_{k-1}) \to a$ (represented by

[1] http://www.haskell.org/cabal/

[2] All tests have been compiled with the Glasgow Haskell compiler with optimization turned on, on a Linux machine with an Intel(R) Core(TM) i5 CPU 680 @ 3.60GHz with 4 MB cache processor and 8 GB main memory.

a DAG of size $3k$). The runtimes for normalization are given in table 3. This tables validates the expectation that the size of the DAGs is important, since the DAG-sizes in the first and second series are similar.

6 Conclusion and Further Work

We showed that STG-compression can advantageously be applied to the word problem for STG-compressed large terms w.r.t. DAG-compressed ground equational theories, which may have a potential use in deduction systems.

Further work is to attack some of the open questions: look for an efficient algorithm for solving the word problem for a set of ground equations or gTRSs under STG-compression or prove hardness results.

Acknowledgement. We thank Markus Lohrey for hints on the complexity of related problems. We also thank the anonymous referees for their helpful comments.

References

1. The Haskell Programming Language (2011), http://www.haskell.org
2. Adams, S.: Efficient sets - a balancing act. J. Funct. Program. 3(4), 553–561 (1993)
3. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press, New York (1998)
4. Busatto, G., Lohrey, M., Maneth, S.: Efficient memory representation of XML documents. In: Bierman, G., Koch, C. (eds.) DBPL 2005. LNCS, vol. 3774, pp. 199–216. Springer, Heidelberg (2005)
5. Busatto, G., Lohrey, M., Maneth, S.: Efficient memory representation of XML document trees. Inf. Syst. 33(4-5), 456–474 (2008)
6. Comon, H., Dauchet, M., Gilleron, R., Jacquemard, F., Lugiez, D., Tison, S., Tommasi, M.: Tree automata techniques and applications (1997), http://www.grappa.univ-lille3.fr/tata (release October 2002)
7. Downey, P.J., Sethi, R., Tarjan, R.E.: Variations on the common subexpression problem. J. ACM 27, 758–771 (1980)
8. Gallier, J.H., Narendran, P., Plaisted, D.A., Raatz, S., Snyder, W.: An algorithm for finding canonical sets of ground rewrite rules in polynomial time. J. ACM 40(1), 1–16 (1993)
9. Garey, M.R., Johnson, D.S.: Computers and Intractability: A guide to the theory of NP-completeness. W.H. Freeman and Co., San Francisco (1979)
10. Gascón, A., Godoy, G., Schmidt-Schauß, M.: Context matching for compressed terms. In: 23rd LICS, pp. 93–102. IEEE Computer Society, Los Alamitos (2008)
11. Gascón, A., Godoy, G., Schmidt-Schauß, M.: Unification and matching on compressed terms (2010), http://arxiv.org/abs/1003.1632v1
12. Gascón, A., Maneth, S., Ramos, L.: First-Order Unification on Compressed Terms. In: Schmidt-Schauß, M. (ed.) 22nd RTA. LIPIcs, vol. 10, pp. 51–60 (2011)
13. Kozen, D.: Complexity of finitely presented algebras. In: 9th STOC, pp. 164–177. ACM, New York (1977)

14. Levy, J., Schmidt-Schauß, M., Villaret, M.: Bounded second-order unification is NP-complete. In: Pfenning, F. (ed.) RTA 2006. LNCS, vol. 4098, pp. 400–414. Springer, Heidelberg (2006)
15. Levy, J., Schmidt-Schauß, M., Villaret, M.: Stratified context unification is NP-complete. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 82–96. Springer, Heidelberg (2006)
16. Levy, J., Schmidt-Schauß, M., Villaret, M.: On the complexity of bounded second-order unification and stratified context unification (2011) (to appear in Logic J. of the IGPL), http://www.ki.informatik.uni-frankfurt.de/papers/schauss/
17. Lifshits, Y.: Processing compressed texts: A tractability border. In: Ma, B., Zhang, K. (eds.) CPM 2007. LNCS, vol. 4580, pp. 228–240. Springer, Heidelberg (2007)
18. Lohrey, M.: Word problems and membership problems on compressed words. SIAM J. Comput. 35(5), 1210–1240 (2006)
19. Lohrey, M., Maneth, S., Schmidt-Schauß, M.: Parameter reduction in grammar-compressed trees. In: de Alfaro, L. (ed.) FOSSACS 2009. LNCS, vol. 5504, pp. 212–226. Springer, Heidelberg (2009)
20. Nelson, G., Oppen, D.C.: Fast decision procedures based on congruence closure. J. ACM 27(2), 356–364 (1980)
21. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(t). J. ACM 53(6), 937–977 (2006)
22. Plandowski, W.: Testing equivalence of morphisms in context-free languages. In: van Leeuwen, J. (ed.) ESA 1994. LNCS, vol. 855, pp. 460–470. Springer, Heidelberg (1994)
23. Plandowski, W., Rytter, W.: Complexity of language recognition problems for compressed words. In: Karhumäki, J., Maurer, H.A., Paun, G., Rozenberg, G. (eds.) Jewels are Forever, pp. 262–272. Springer, Heidelberg (1999)
24. Schmidt-Schauß, M.: Polynomial equality testing for terms with shared substructures. Frank report 21, Institut für Informatik. FB Informatik und Mathematik. J. W. Goethe-Universität Frankfurt am Main (2005)
25. Schmidt-Schauß, M.: Pattern matching of compressed terms and contexts and polynomial rewriting. Frank report 43, Institut für Informatik. Goethe-Universität Frankfurt am Main (2011)
26. Shostak, R.E.: An algorithm for reasoning about equality. Commun. ACM 21(7), 583–585 (1978)
27. Snyder, W.: Efficient ground completion: An $o(nlogn)$ algorithm for generating reduced sets of ground rewrite rules equivalent to a set of ground equations e. In: Dershowitz, N. (ed.) RTA 1989. LNCS, vol. 355, pp. 419–433. Springer, Heidelberg (1989)
28. Snyder, W.: A fast algorithm for generating reduced ground rewriting systems from a set of ground equations. J. Symb. Comput. 15(4), 415–450 (1993)
29. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Trans. Inform. Theory 23(3), 337–343 (1977)

Generalized and Formalized Uncurrying*

Christian Sternagel and René Thiemann

Institute of Computer Science, University of Innsbruck, Austria
{christian.sternagel,rene.thiemann}@uibk.ac.at

Abstract. Uncurrying is a termination technique for applicative term rewrite systems. During our formalization of uncurrying in the theorem prover Isabelle, we detected a gap in the original pen-and-paper proof which cannot directly be filled without further preconditions. Our final formalization does not demand additional preconditions, and generalizes the existing techniques since it allows to uncurry non-applicative term rewrite systems. Furthermore, we provide new results on uncurrying for relative termination and for dependency pairs.

Keywords: uncurrying, termination, formalization, interactive theorem proving, dependency pairs, term rewriting.

1 Introduction

In recent years, the way to prove termination of term rewrite systems (TRSs) has changed. Current termination tools no longer search for a single reduction order containing the rewrite relation. Instead, they combine various termination techniques in a modular way, resulting in large and tree-like termination proofs, where at each node a specific technique is applied.

On the one hand, this combination makes termination tools more powerful. On the other hand, it makes them more complex and error-prone. It is regularly demonstrated that we cannot blindly trust the output of termination provers. Every now and then, some prover delivers a faulty proof. Often, this is only detected if there is another prover giving a contradictory answer for the same input, as a manual inspection of proofs is infeasible due to their size.

The problem is solved by combining two systems. For a given TRS, we first use a termination tool to automatically detect a termination proof (which may contain errors). Then, we use a highly trusted certifier which checks whether the detected proof is indeed correct. In total, the combination yields a powerful and trustable workflow to prove termination.

To obtain a highly trustable certifier, a common approach is to first formalize the desired termination techniques once and for all (thereby ensuring their soundness) and then, for a given proof, check that the used techniques are applied correctly [2,3,14]. We formalized the dependency pair framework (DP framework) [5] and many termination techniques in our Isabelle/HOL [11] library IsaFoR [14]

* This research is supported by FWF (Austrian Science Fund) project P22767-N13.

C. Tinelli and V. Sofronie-Stokkermans (Eds.): FroCoS 2011, LNAI 6989, pp. 243–258, 2011.

(in the remainder we just write *Isabelle*, instead of *Isabelle/HOL*). From IsaFoR, we code-extract CeTA, an automatic certifier for termination proofs.

In this paper, we present one of the latest additions to IsaFoR: the formalization of uncurrying, as described in [8]. However, we did not only formalize uncurrying, but also generalized it. Furthermore, we found a gap in one of the original proofs, which we could fortunately close.

Note that all the proofs that are presented (or omitted) in the following, have been formalized as part of IsaFoR. Hence, in this paper, we merely give sketches of our "real" proofs. Our goal is to show the general proof outlines and help to understand the full proofs. The library IsaFoR with all formalized proofs, the executable certifier CeTA, and all details about our experiments are available at CeTA's website: `http://cl-informatik.uibk.ac.at/software/ceta`.

The paper is structured as follows. In Sect. 2, we shortly recapitulate some required notions of term rewriting. Afterwards, in Sect. 3, we describe applicative rewriting, give an overview of approaches using *uncurrying* for proving termination, and present our generalization of uncurrying for TRSs. Then, in Sect. 4, we show how to lift uncurrying to the DP framework. We present heuristics and our experiments in Sect. 5, before we conclude in Sect. 6.

2 Preliminaries

We assume familiarity with term rewriting [1]. Still, we recall the most important notions that are used later on. A *(first-order) term* t over a set of *variables* $\mathcal{V}$ and a set of *(function) symbols* $\mathcal{F}$ is either a variable $x \in \mathcal{V}$ or a function symbol $f \in \mathcal{F}$ applied to argument terms $f(t_1, \ldots, t_n)$ where the *arity* of f is $\mathsf{ar}(f) = n$. A *context* C is a term containing exactly one hole $\Box$. Replacing $\Box$ in a context C by a term t is denoted by $C[t]$.

A *rewrite rule* is a pair of terms $\ell \to r$ and a TRS $\mathcal{R}$ is a set of rewrite rules. The set of *defined symbols* (of $\mathcal{R}$) is $\mathcal{D}_\mathcal{R} = \{f \mid f(\ldots) \to r \in \mathcal{R}\}$. The *rewrite relation (induced by $\mathcal{R}$)* $\to_\mathcal{R}$ is the closure under substitutions and contexts of $\mathcal{R}$, i.e., $s \to_\mathcal{R} t$ iff there is a context C, a rewrite rule $\ell \to r \in \mathcal{R}$, and a substitution σ such that $s = C[\ell\sigma]$ and $t = C[r\sigma]$. A term t is *root-stable w.r.t.* $\mathcal{R}$ iff there is no derivation $t \to^*_\mathcal{R} \ell\sigma$ for some $\ell \to r \in \mathcal{R}$ and substitution σ.

We say that a term t is *terminating w.r.t.* $\mathcal{R}$ ($\mathsf{SN}_\mathcal{R}(t)$) if it cannot start an infinite derivation $t = t_1 \to_\mathcal{R} t_2 \to_\mathcal{R} t_3 \to_\mathcal{R} \cdots$. A TRS is terminating ($\mathsf{SN}(\mathcal{R})$) iff all terms are terminating w.r.t. $\mathcal{R}$. A TRS $\mathcal{R}$ is *terminating relative to* a TRS $\mathcal{S}$ iff there is no infinite $\mathcal{R} \cup \mathcal{S}$-derivation with infinitely many $\mathcal{R}$-steps.

3 Applicative Rewriting and Uncurrying

An applicative term rewrite system (ATRS) is a TRS over an *applicative signature* $\mathcal{F} = \{\circ\} \cup \mathcal{C}$, where $\circ$ is a unique binary symbol (the application symbol) and all symbols in $\mathcal{C}$ are constants. ATRSs can be used to encode many higher-order functions without explicit abstraction as first-order TRSs. In the remainder we use $\circ$ as an infix-symbol which associates to the left ($s \circ t \circ u = (s \circ t) \circ u$). In examples we omit $\circ$ whenever this increases readability.

Example 1. The following ATRS $\mathcal{R}$ (a variant of [8, Example 7], replacing addition by subtraction) contains the map function (which applies a function to all arguments of a list) and subtraction on Peano numbers in applicative form.

$$\begin{array}{lll}
1: & \mathsf{sub}\ 0 \to \mathsf{K}\ 0 & 5: \quad \mathsf{K}\ x\ y \to x \\
2: & \mathsf{sub}\ x\ 0 \to x & 6: \quad \mathsf{map}\ z\ \mathsf{nil} \to \mathsf{nil} \\
3: & \mathsf{sub}\ x\ x \to 0 & 7: \quad \mathsf{map}\ z\ (\mathsf{cons}\ x\ xs) \to \mathsf{cons}\ (z\ x)\ (\mathsf{map}\ z\ xs) \\
4: & \mathsf{sub}\ (\mathsf{s}\ x)\ (\mathsf{s}\ y) \to \mathsf{sub}\ x\ y &
\end{array}$$

Proving termination of ATRSs is challenging without dedicated termination techniques (e.g., for reduction orders, we cannot interpret $\mathsf{sub} \circ x \circ y$ as x, since sub is a constant and not binary).

Until now, there have at least been three approaches to tackle this problem. All of them try to uncurry a TRS such that, for example, Rule 4 from above becomes $\mathsf{sub}(\mathsf{s}(x), \mathsf{s}(y)) \to \mathsf{sub}(x, y)$.

To distinguish the three approaches, we need the following definitions:

Definition 2. *A term t is* head variable free *iff t does not contain a subterm of the form $x \circ s$ where x is a variable. The* applicative arity *of a constant f in an ATRS $\mathcal{R}$ ($\mathsf{aa}_{\mathcal{R}}(f)$) is the maximal number n, such that $f \circ t_1 \circ \cdots \circ t_n$ occurs as a subterm in $\mathcal{R}$.* Uncurrying *an application $f \circ t_1 \circ \cdots \circ t_n$ with $\mathsf{aa}_{\mathcal{R}}(f) = n$ yields the term $f(t_1, \ldots, t_n)$. A term t is* proper w.r.t. $\mathsf{aa}_{\mathcal{R}}$ *iff t is a variable or $t = f \circ t_1 \circ \cdots \circ t_n$ where $\mathsf{aa}_{\mathcal{R}}(f) = n$ and each t_i is proper.*

The oldest of the three approaches is from [9]. It requires that all terms in a TRS are proper w.r.t. $\mathsf{aa}_{\mathcal{R}}$, and shows that then termination of $\mathcal{R}$ is equivalent to termination of the TRS obtained by uncurrying all terms of $\mathcal{R}$. Since proper terms do not contain any partial applications, the application symbol is completely eliminated by uncurrying. However, requiring proper terms is rather restrictive: Essentially, it is demanded that the TRS under consideration is a standard first-order TRS which is just written in applicative form. For example, the approach is not applicable to Example 1, since there is a head variable in the right-hand side of Rule 7 ($z \circ x$) and sub as well as K are applied to a single argument in Rule 1, even though $\mathsf{aa}_{\mathcal{R}}(\mathsf{sub}) = 2$ and $\mathsf{aa}_{\mathcal{R}}(\mathsf{K}) = 2$.

The second approach was given in [6,13]. Here, the same preconditions as in [9] apply, but the results are extended to innermost rewriting and to the DP framework. The latter has the advantage, that only the current subproblem has to satisfy the preconditions. For example, when treating the dependency pair

$$\mathsf{map} \circ z \circ^{\sharp} (\mathsf{cons} \circ x \circ xs) \to \mathsf{map} \circ z \circ^{\sharp} xs \tag{1}$$

for the recursive call of map, we can perform uncurrying (since there are no usable rules and (1) satisfies the preconditions). Moreover, in [13] uncurrying is combined with the reduction pair processor to further relax the preconditions.

The third approach is given in [8]. Here, the preconditions for uncurrying have been reduced drastically as only the left-hand sides of the TRS $\mathcal{R}$ must be head variable free. In return, we have to η-saturate $\mathcal{R}$ and add uncurrying rules. Moreover, for each constant f with $\mathsf{aa}_{\mathcal{R}}(f) = n$ we obtain n new function symbols $f_1, \ldots, f_n$ of arities $1, 2, \ldots, n$ which handle partial applications.

Example 3. When η-saturating the TRS $\mathcal{R}$ of Example 1, we have to add the rule $\mathsf{sub} \circ \mathsf{0} \circ y \to \mathsf{K} \circ \mathsf{0} \circ y$. The uncurried TRS consists of the following rules:

$$
\begin{array}{rlrl}
8: & \mathsf{sub}_1(\mathsf{0}) \to \mathsf{K}_1(\mathsf{0}) & 12: & \mathsf{sub}_2(\mathsf{s}_1(x), \mathsf{s}_1(y)) \to \mathsf{sub}_2(x, y) \\
9: & \mathsf{sub}_2(\mathsf{0}, y) \to \mathsf{K}_2(\mathsf{0}, y) & 13: & \mathsf{K}_2(x, y) \to x \\
10: & \mathsf{sub}_2(x, \mathsf{0}) \to x & 14: & \mathsf{map}_2(z, \mathsf{nil}) \to \mathsf{nil} \\
11: & \mathsf{sub}_2(x, x) \to \mathsf{0} & 15: & \mathsf{map}_2(z, \mathsf{cons}_2(x, xs)) \to \mathsf{cons}_2(z \circ x, \mathsf{map}_2(z, xs))
\end{array}
$$

Moreover, we have to add the following uncurrying rules:

$$
\begin{array}{rlrl}
16: & \mathsf{s} \circ x \to \mathsf{s}_1(x) & 21: & \mathsf{cons} \circ x \to \mathsf{cons}_1(x) \\
17: & \mathsf{K} \circ x \to \mathsf{K}_1(x) & 22: & \mathsf{cons}_1(x) \circ y \to \mathsf{cons}_2(x, y) \\
18: & \mathsf{K}_1(x) \circ y \to \mathsf{K}_2(x, y) & 23: & \mathsf{map} \circ x \to \mathsf{map}_1(x) \\
19: & \mathsf{sub} \circ x \to \mathsf{sub}_1(x) & 24: & \mathsf{map}_1(x) \circ y \to \mathsf{map}_2(x, y) \\
20: & \mathsf{sub}_1(x) \circ y \to \mathsf{sub}_2(x, y) &&
\end{array}
$$

Also [8] gives an extension to the DP framework.

To summarize, the traditional technique of uncurrying of [9] is completely subsumed by [6,13], but [6,13] and [8] are incomparable. The advantage of [6,13] is that the generated TRSs and DP problems are smaller, and that uncurrying is also available in a combination with the reduction pair processor, whereas [8] supports head variables (see [13, Chap. 6] for a more detailed comparison).

Since [8] is used in more termination tools (it is used in at least Jambox [4] and TTT2 [10] whereas we only know of AProVE [7] that implements all uncurrying techniques of [6,13]), we incorporated the techniques of [8] in our certifier CeTA.

During our formalization we have

- detected a gap in a proof of [8] which could not directly be closed without adding further preconditions to one of the main results,
- generalized the technique of uncurrying which now entails the result of [8] even without adding any additional precondition, and
- generalized the technique of *freezing* from [8].

The structure in [8] is as follows. First, uncurrying is developed for TRSs over applicative signatures $\{\circ\} \cup \mathcal{C}$. Then, it is extended to DP problems, introducing a second application symbol $\circ^\sharp$ that may only occur at root-positions of $\mathcal{P}$ and is not uncurried at all. Finally, *freezing* is applied to uncurry applications of $\circ^\sharp$.

Following this structure, we first fully formalized uncurrying on TRSs. However, in the extension to DP problems there is a missing step which is illustrated in more detail in Example 14 on page 251. The main problem is that signature restrictions on DP problems are in general unsound.

To fill the gap, one option is to use the results of [12] about signature restrictions, which can however only be applied if $\mathcal{R}$ is left-linear. This clearly weakens the applicability of uncurrying, e.g., Example 1 is not left-linear.

Alternatively, one can try to perform uncurrying without restricting to applicative signatures. This is what we did. All uncurrying techniques that we formalized work on terms and TRSs over arbitrary signatures.

The major complication is the increase of complexity in the cases that have to be considered. For example, using an applicative signature, we can assume

that every term is of the form $x \circ t_1 \circ \cdots \circ t_n$ or $f \circ t_1 \circ \cdots \circ t_n$ where $n \in \mathbb{N}$, x is a variable, and f is a constant. Generalizing this to arbitrary signatures we have to consider the two cases $x \circ t_1 \circ \cdots \circ t_n$ and $f(s_1, \ldots, s_m) \circ t_1 \circ \cdots \circ t_n$ instead, where f is an m-ary symbol. Hence, when considering a possible rewrite step, we also have to consider the new case that the step is performed in some s_i.

We not only generalized uncurrying to work for arbitrary signatures and relative rewriting, but also to a free choice of the applicative arity $\mathsf{aa}(f)$. This is in contrast to [8], where the applicative arity is fixed by Definition 2. We will elaborate on this difference after presenting our main theorem.

Definition 4 (Symbol maps and applicative arity). *Let $\mathcal{F}$ be a signature. A* symbol map *is a mapping $\pi : \mathcal{F} \to [\mathcal{F}]$ from symbols to non-empty lists of symbols. It is* injective *if for all f and g, $\pi(f)$ contains no duplicates, $\pi(f)$ does not contain $\circ$, and whenever $f \neq g$ then $\pi(f)$ and $\pi(g)$ do not share symbols. If $\pi(f) = [f_0, \ldots, f_n]$, then the* applicative arity of f w.r.t. π *is $\mathsf{aa}_\pi(f) = n$. The* applicative arity of a term *is defined by $\mathsf{aa}_\pi(t) = \mathsf{aa}_\pi(f) \dot{-} n$, where $x \dot{-} y = \max(x - y, 0)$, for $t = f(s_1, \ldots, s_m) \circ t_1 \circ \cdots \circ t_n$ and is undefined otherwise.*

Intuitively, if $\pi(f) = [f_0, \ldots, f_n]$ then every application of f on $i \leqslant n$ arguments $t_1, \ldots, t_i$ will be fully uncurried to $f_i(t_1, \ldots, t_i)$. If more than n arguments are applied, then we obtain $f_n(t_1, \ldots, t_n) \circ t_{n+1} \circ \ldots \circ t_i$. A symbol map containing an entry for f, uniquely determines the applicative arity n as well as the names of the (partial) applications $f_0, \ldots, f_n$ of f.

In the following we assume a fixed symbol map π and just write $\mathsf{aa}(f)$ and $\mathsf{aa}(t)$ instead of $\mathsf{aa}_\pi(f)$ and $\mathsf{aa}_\pi(t)$, respectively. Additionally, we assume that $\pi(f) = [f_0, \ldots, f_{\mathsf{aa}(f)}]$ where in examples we write f instead of f_0. Now we can define the uncurrying TRS w.r.t. π.

Definition 5. *The* uncurrying TRS $\mathcal{U}$ *contains the rule*

$$f_k(x_1, \ldots, x_m, y_1, \ldots, y_k) \circ y_{k+1} \to f_{k+1}(x_1, \ldots, x_m, y_1, \ldots, y_{k+1})$$

for every $f \in \mathcal{F}$ with $\mathsf{ar}(f) = m$ and $\mathsf{aa}(f) = n$, and every $k < n$. The variables $x_1, \ldots, x_m, y_1, \ldots, y_{k+1}$ are pairwise distinct.

In [8], terms are uncurried by computing the unique normal form w.r.t. $\mathcal{U}$. For our formalization we instead used the upcoming uncurrying function for the following two reasons: First, we do not have any results about confluence of TRSs. Hence, to even define *the* normal form w.r.t. $\mathcal{U}$ would require to formalize several additional lemmas which show that every term has exactly one normal form. This would be quite some effort which we prefer to avoid. The second reason is efficiency. When certifying the application of uncurrying in large termination proofs, we have to compute the uncurried version of a term. It is just more efficient to use a function which performs uncurrying directly, than to compute a normal form w.r.t. a TRS where possible redexes have to be searched, etc.

Definition 6. *The uncurrying function $\llcorner\cdot\lrcorner$ on terms is defined as*

- $\llcorner x \circ t_1 \circ \cdots \circ t_n \lrcorner = x \circ \llcorner t_1 \lrcorner \circ \cdots \circ \llcorner t_n \lrcorner$

– $\llcorner f(s_1, \ldots, s_m) \circ t_1 \circ \cdots \circ t_n \lrcorner = f_k(\llcorner s_1 \lrcorner, \ldots, \llcorner s_m \lrcorner, \llcorner t_1 \lrcorner, \ldots, \llcorner t_k \lrcorner) \circ \llcorner t_{k+1} \lrcorner \circ \cdots \circ \llcorner t_n \lrcorner$ *where* $k = \min(n, \mathsf{aa}(f))$

It is homomorphically extended to operate on rules, TRSs, and substitutions.

We establish the link between $\mathcal{U}$ and $\llcorner \cdot \lrcorner$ in the following lemma which generalizes the corresponding results in [8].

Lemma 7

– *if* $k < \mathsf{aa}(f)$ *and* $\mathsf{ar}(f) = m$ *then* $f_k(s_1, \ldots, s_{m+k}) \circ t \to_{\mathcal{U}} f_{k+1}(s_1, \ldots, s_{m+k}, t)$
– *if* $k + n \leqslant \mathsf{aa}(f)$ *and* $\mathsf{ar}(f) = m$ *then* $f_k(s_1, \ldots, s_{m+k}) \circ t_1 \circ \ldots \circ t_n \to^*_{\mathcal{U}} f_{k+n}(s_1, \ldots, s_{m+k}, t_1, \ldots, t_n)$
– $\llcorner s \lrcorner \circ \llcorner t_1 \lrcorner \circ \cdots \circ \llcorner t_n \lrcorner \to^*_{\mathcal{U}} \llcorner s \circ t_1 \circ \cdots \circ t_n \lrcorner$
– *if* $\mathsf{aa}(s) = 0$ *or* $\mathsf{aa}(s)$ *is undefined then* $\llcorner s \circ t_1 \circ \cdots \circ t_n \lrcorner = \llcorner s \lrcorner \circ \llcorner t_1 \lrcorner \circ \cdots \circ \llcorner t_n \lrcorner$
– $\llcorner s \lrcorner \cdot \llcorner \sigma \lrcorner \to^*_{\mathcal{U}} \llcorner s \cdot \sigma \lrcorner$
– *if* t *is head variable free then* $\llcorner s \cdot \sigma \lrcorner = \llcorner s \lrcorner \cdot \llcorner \sigma \lrcorner$

The last two results show how uncurrying can be exchanged with applying substitutions. As we will often need the equality $\llcorner \ell \cdot \sigma \lrcorner = \llcorner \ell \lrcorner \cdot \llcorner \sigma \lrcorner$ for left-hand sides ℓ, it is naturally to demand that left-hand sides are head variable free.

Definition 8. *A TRS is* left head variable free *if all left-hand sides are head variable free.*

Termination of $\llcorner \mathcal{R} \lrcorner \cup \mathcal{U}$ does not suffice to conclude termination of $\mathcal{R}$, cf. [8, Example 13]. The reason is that first we have to η-saturate $\mathcal{R}$.

Definition 9. *A TRS* $\mathcal{R}$ *is* η-closed *iff for every rule* $\ell \to r$ *with* $\mathsf{aa}(\ell) > 0$ *there is a rule* $\ell \circ x \to r \circ x \in \mathcal{R}$ *where* x *is fresh w.r.t.* $\ell \to r$*. The* η-saturation $\mathcal{R}_\eta$ *of* $\mathcal{R}$ *is obtained by adding new rules* $\ell \circ x \to r \circ x$ *until the result is* η*-closed.*

The upcoming theorem is the key to use uncurrying for termination proofs. It allows to simulate one $\mathcal{R}$-step by many steps in the uncurried system.

Theorem 10. *Let* $\mathcal{R}$ *be* η*-closed and left head variable free. Let there be no left-hand side of* $\mathcal{R}$ *which is a variable. If* $s \to_{\mathcal{R}} t$ *then* $\llcorner s \lrcorner \to^+_{\llcorner \mathcal{R} \lrcorner \cup \mathcal{U}} \llcorner t \lrcorner$.

Proof. Let $s = C[\ell\sigma] \to_{\mathcal{R}} C[r\sigma] = t$ where $\ell \to r \in \mathcal{R}$. We show $\llcorner s \lrcorner \to^+_{\llcorner \mathcal{R} \lrcorner \cup \mathcal{U}} \llcorner t \lrcorner$ by induction on the size of C.

– If $C = f(s_1, \ldots, D, \ldots, s_m) \circ t_1 \circ \cdots \circ t_n$ for some $f \neq \circ$ then by the induction hypothesis we know that $\llcorner D[\ell\sigma] \lrcorner \to^+_{\llcorner \mathcal{R} \lrcorner \cup \mathcal{U}} \llcorner D[r\sigma] \lrcorner$. Moreover,

$$\begin{aligned}
\llcorner s \lrcorner &= \llcorner f(s_1, \ldots, D[\ell\sigma], \ldots, s_m) \circ t_1 \circ \cdots \circ t_n \lrcorner \\
&= f_k(\llcorner s_1 \lrcorner, \ldots, \llcorner D[\ell\sigma] \lrcorner, \ldots, \llcorner s_m \lrcorner, \llcorner t_1 \lrcorner, \ldots, \llcorner t_k \lrcorner) \circ \llcorner t_{k+1} \lrcorner \circ \cdots \circ \llcorner t_n \lrcorner \\
&\to^+_{\llcorner \mathcal{R} \lrcorner \cup \mathcal{U}} f_k(\ldots, \llcorner D[r\sigma] \lrcorner, \ldots, \llcorner s_m \lrcorner, \llcorner t_1 \lrcorner, \ldots, \llcorner t_k \lrcorner) \circ \llcorner t_{k+1} \lrcorner \circ \cdots \circ \llcorner t_n \lrcorner \\
&= \llcorner f(s_1, \ldots, D[r\sigma], \ldots, s_m) \circ t_1 \circ \cdots \circ t_n \lrcorner \\
&= \llcorner t \lrcorner
\end{aligned}$$

where $k = \min(n, \mathsf{aa}(f))$.

– If $C = t_0 \circ D \circ t_1 \circ \cdots \circ t_n$ then by the induction hypothesis we know that $\llcorner D[\ell\sigma] \lrcorner \to^+_{\llcorner \mathcal{R} \lrcorner \cup \mathcal{U}} \llcorner D[r\sigma] \lrcorner$. If $\mathsf{aa}(t_0) = 0$ or $\mathsf{aa}(t_0)$ is undefined then

$$\begin{aligned}
\llcorner s \lrcorner &= \llcorner t_0 \circ D[\ell\sigma] \circ t_1 \circ \cdots \circ t_n \lrcorner \\
&= \llcorner t_0 \lrcorner \circ \llcorner D[\ell\sigma] \lrcorner \circ \llcorner t_1 \lrcorner \circ \cdots \circ \llcorner t_n \lrcorner \\
&\to^+_{\llcorner \mathcal{R} \lrcorner \cup \mathcal{U}} \llcorner t_0 \lrcorner \circ \llcorner D[r\sigma] \lrcorner \circ \llcorner t_1 \lrcorner \circ \cdots \circ \llcorner t_n \lrcorner \\
&= \llcorner t_0 \circ D[r\sigma] \circ t_1 \circ \cdots \circ t_n \lrcorner \\
&= \llcorner t \lrcorner
\end{aligned}$$

Otherwise, $\mathsf{aa}(t_0) > 0$ and hence, $t_0 = f(s_1, \ldots, s_m) \circ s_{m+1} \circ \cdots \circ s_{m+k}$ where $k < \mathsf{aa}(f)$. It follows that

$$\begin{aligned}
\llcorner s \lrcorner &= \llcorner f(s_1, \ldots, s_m) \circ s_{m+1} \circ \cdots \circ s_{m+k} \circ D[\ell\sigma] \circ t_1 \circ \cdots \circ t_n \lrcorner \\
&= f_{k+1+n'}(\ldots, \llcorner s_{m+k} \lrcorner, \llcorner D[\ell\sigma] \lrcorner, \llcorner t_1 \lrcorner, \ldots, \llcorner t_{n'} \lrcorner) \circ \llcorner t_{n'+1} \lrcorner \circ \cdots \circ \llcorner t_n \lrcorner \\
&\to^+_{\llcorner \mathcal{R} \lrcorner \cup \mathcal{U}} f_{k+1+n'}(\ldots, \llcorner s_{m+k} \lrcorner, \llcorner D[r\sigma] \lrcorner, \llcorner t_1 \lrcorner, \ldots, \llcorner t_{n'} \lrcorner) \circ \llcorner t_{n'+1} \lrcorner \circ \cdots \\
&= \llcorner f(s_1, \ldots, s_m) \circ s_{m+1} \circ \cdots \circ s_{m+k} \circ D[r\sigma] \circ t_1 \circ \cdots \circ t_n \lrcorner \\
&= \llcorner t \lrcorner
\end{aligned}$$

where $n' = \min(\mathsf{aa}(f) - k - 1, n)$.

– If $C = \Box \circ t_1 \circ \cdots \circ t_n$ and $n = 0 \vee \mathsf{aa}(\ell) = 0$ then

$$\begin{aligned}
\llcorner s \lrcorner &= \llcorner \ell \cdot \sigma \circ t_1 \circ \cdots \circ t_n \lrcorner \\
&= \llcorner \ell \cdot \sigma \lrcorner \circ \llcorner t_1 \lrcorner \circ \cdots \circ \llcorner t_n \lrcorner \\
&= \llcorner \ell \lrcorner \cdot \llcorner \sigma \lrcorner \circ \llcorner t_1 \lrcorner \circ \cdots \circ \llcorner t_n \lrcorner \\
&\to_{\llcorner \mathcal{R} \lrcorner} \llcorner r \lrcorner \cdot \llcorner \sigma \lrcorner \circ \llcorner t_1 \lrcorner \circ \cdots \circ \llcorner t_n \lrcorner \\
&\to^*_{\mathcal{U}} \llcorner r \cdot \sigma \lrcorner \circ \llcorner t_1 \lrcorner \circ \cdots \circ \llcorner t_n \lrcorner \\
&\to^*_{\mathcal{U}} \llcorner r \cdot \sigma \circ t_1 \circ \cdots \circ t_n \lrcorner \\
&= \llcorner t \lrcorner
\end{aligned}$$

since ℓ is head variable free and if $n \neq 0$ then $\mathsf{aa}(\ell\sigma) = \mathsf{aa}(\ell) = 0$.

– If $C = \Box \circ t_1 \circ \cdots \circ t_n$ with $n > 0$ and $\mathsf{aa}(\ell) > 0$ then $\ell' \to r' = \ell \circ x \to r \circ x \in \mathcal{R}$ since $\mathcal{R}$ is η-closed. Moreover, by changing σ to $\delta = \sigma \uplus \{x/t_1\}$ we achieve $s = \ell \cdot \sigma \circ t_1 \circ \cdots \circ t_n = \ell'\delta \circ t_2 \circ \cdots \circ t_n = D[\ell'\delta]$ and $r = r \cdot \sigma \circ t_1 \circ \cdots \circ t_n = r'\delta \circ t_2 \circ \cdots \circ t_n = D[r'\delta]$ for the context $D = \Box \circ t_2 \circ \cdots t_n$ which is strictly smaller than C. Hence, the result follows by the induction hypothesis.

– If $C = \Box \circ t_1 \circ \cdots \circ t_n$ with $n > 0$ and $\mathsf{aa}(\ell)$ is undefined then $\ell = x \circ \ell_1 \circ \cdots \circ \ell_k$ with $k \geq 0$. But if $k > 0$ then ℓ is not head variable free and if $k = 0$ then $\mathcal{R}$ contains a variable as left-hand side. In both cases we get a contradiction to the preconditions in the theorem. □

Note that the condition that the left-hand sides of $\mathcal{R}$ are not variables is new in comparison to [8]. Nevertheless, in the following corollary we can drop this condition, since otherwise $\llcorner \mathcal{R} \lrcorner$ is not terminating anyway.

Corollary 11. *If $\mathcal{R}_\eta$ is left head variable free then termination of $\llcorner\mathcal{R}_\eta\lrcorner \cup \mathcal{U}$ implies termination of $\mathcal{R}$.*

When using uncurrying for relative termination of $\mathcal{R}/\mathcal{S}$, it turns out that the new condition on the left-hand sides can only be ignored for $\mathcal{R}$ – since otherwise relative termination of $\llcorner\mathcal{R}\lrcorner/\llcorner\mathcal{S}\lrcorner \cup \mathcal{U}$ does not hold – but not for $\mathcal{S}$.

Corollary 12. *If $\mathcal{R}_\eta \cup \mathcal{S}_\eta$ is left head variable free and the left-hand sides of $\mathcal{S}$ are not variables, then relative termination of $\llcorner\mathcal{R}_\eta\lrcorner/\llcorner\mathcal{S}_\eta\lrcorner \cup \mathcal{U}$ implies relative termination of $\mathcal{R}/\mathcal{S}$.*

Example 13. Let $\mathcal{R} = \{\mathsf{f} \circ \mathsf{f} \circ x \to \mathsf{f} \circ x\}$ and $\mathcal{S} = \{x \to \mathsf{f} \circ x\}$. Then $\mathcal{R}/\mathcal{S}$ is not relative terminating since $\mathsf{f} \circ \mathsf{f} \circ x \to_\mathcal{R} \mathsf{f} \circ x \to_\mathcal{S} \mathsf{f} \circ \mathsf{f} \circ x \to_\mathcal{R} \ldots$ is an infinite $\mathcal{R} \cup \mathcal{S}$-derivation with infinitely many $\mathcal{R}$-steps.

For $\pi(\mathsf{f}) = [\mathsf{f}, \mathsf{f}_1, \mathsf{f}_2]$ we have $\mathcal{R}_\eta = \mathcal{R}$, $\mathcal{S}_\eta = \mathcal{S}$, $\llcorner\mathcal{R}_\eta\lrcorner = \{\mathsf{f}_2(\mathsf{f}, x) \to \mathsf{f}_1(x)\}$, $\llcorner\mathcal{S}_\eta\lrcorner = \{x \to \mathsf{f}_1(x)\}$, and $\mathcal{U} = \{\mathsf{f} \circ x \to \mathsf{f}_1(x), \mathsf{f}_1(x) \circ y \to \mathsf{f}_2(x, y)\}$. It is easy to see that $\llcorner\mathcal{R}_\eta\lrcorner/\llcorner\mathcal{S}_\eta\lrcorner \cup \mathcal{U}$ is relative terminating by counting the number of f-symbols. Since both $\mathcal{R}_\eta$ and $\mathcal{S}_\eta$ are head variable free, we have shown that Corollary 12 does not hold if one would drop the new variable condition on $\mathcal{S}$.

As already mentioned, Corollary 11 generalizes the similar result of [8, Theorem 16] in two ways: first, there is no restriction to applicative signatures, and second, one can freely choose the applicative arities. Since in principle the choice of π does not matter (uncurrying preserves termination for every choice of π), we can only heuristically determine whether the additional freedom increases termination proving power and therefore refer to our experiments in Sect. 5.

4 Uncurrying in the Dependency Pair Framework

The DP framework [5] facilitates modular termination proofs. Instead of single TRSs, we consider *DP problems* $(\mathcal{P}, \mathcal{R})$, consisting of two TRSs $\mathcal{P}$ and $\mathcal{R}$ where elements of $\mathcal{P}$ are often called *pairs* to distinguish them from the *rules* of $\mathcal{R}$. The *initial DP problem* for a TRS $\mathcal{R}$ is $(\mathsf{DP}(\mathcal{R}), \mathcal{R})$, where $\mathsf{DP}(\mathcal{R}) = \{f^\sharp(s_1, \ldots, s_n) \to g^\sharp(t_1, \ldots, t_m) \mid f(s_1, \ldots, s_n) \to C[g(t_1, \ldots, t_m)] \in \mathcal{R}, g \in \mathcal{D}_\mathcal{R}\}$ is the set of *dependency pairs* of $\mathcal{R}$, incorporating a fresh *tuple symbol* $f^\sharp$ for each defined symbol f. The initial DP problem is also a *standard* DP problem, i.e., root symbols of pairs do not occur elsewhere in $\mathcal{P}$ or $\mathcal{R}$.[1] A $(\mathcal{P}, \mathcal{R})$-*chain* is a possibly infinite derivation of the form:

$$s_1\sigma_1 \to_\mathcal{P} t_1\sigma_1 \to^*_\mathcal{R} s_2\sigma_2 \to_\mathcal{P} t_2\sigma_2 \to^*_\mathcal{R} s_3\sigma_3 \to_\mathcal{P} \cdots \qquad (\star)$$

where $s_i \to t_i \in \mathcal{P}$ for all $i > 0$. If additionally every $t_i\sigma_i$ is terminating w.r.t. $\mathcal{R}$, then the chain is *minimal*. A DP problem $(\mathcal{P}, \mathcal{R})$ is called *finite* [5], if there is no minimal infinite $(\mathcal{P}, \mathcal{R})$-chain. Proving finiteness of a DP problem is done by simplifying $(\mathcal{P}, \mathcal{R})$ using so called *processors* recursively. A processor transforms a DP problem into a new DP problem. The aim is to reach a DP problem with empty $\mathcal{P}$-component (such DP problems are trivially finite). To conclude

[1] Several termination provers only work on standard DP problems.

finiteness of the initial DP problem, the applied processors need to be *sound*. A processor *Proc* is sound whenever for all DP problems $(\mathcal{P},\mathcal{R})$ we have that finiteness of $Proc(\mathcal{P},\mathcal{R})$ implies finiteness of $(\mathcal{P},\mathcal{R})$.

In the following we explain how uncurrying is used as processor in the DP framework. In Sect. 3 it was already mentioned that in [8] the notion of applicative TRS was lifted to applicative DP problem by allowing a new binary application symbol $\circ^\sharp$ (where we sometimes just write $^\sharp$ in examples). The symbol $\circ^\sharp$ naturally occurs as tuple symbol of $\circ$.

To prove soundness of the uncurrying processor, in [8] it is assumed that there is a minimal $(\mathcal{P},\mathcal{R})$-chain $s_1\sigma_1 \to_{\mathcal{P}} t_1\sigma_1 \to^*_{\mathcal{R}} s_2\sigma_2 \to_{\mathcal{P}} \cdots$, which is converted into a minimal $(\llcorner\mathcal{P}\lrcorner,\llcorner\mathcal{R}_\eta\lrcorner\cup\mathcal{U})$-chain by reusing the results for TRSs. However, there is a gap in this reasoning. Right in the beginning it is silently assumed that all terms $s_i\sigma_i$ and $t_i\sigma_i$ have tuple symbols as roots and that their arguments are applicative terms, i.e., terms over an applicative signature $\{\circ\}\cup\mathcal{C}$. Without this assumption it is not possible to apply the results of uncurrying for TRSs, since those are only available for applicative terms in [8].

The following variant of [12, Example 14] shows that in general restricting substitutions in chains to an applicative signature $\{\circ\}\cup\mathcal{C}$ is unsound.

Example 14. Consider the applicative and standard DP problem $(\mathcal{P},\mathcal{R})$ where $\mathcal{P} = \{\mathsf{g}^\sharp\ (\mathsf{f}\ x\ y\ z\ z\ u\ v) \to \mathsf{g}^\sharp\ (\mathsf{f}\ x\ y\ x\ y\ x\ (\mathsf{h}\ y\ u))\}$ and $\mathcal{R}$ contains the rules:

$$\begin{aligned}
\mathsf{a} &\to \mathsf{b} & \mathsf{f}\ \mathsf{a}\ x_2\ x_3\ x_4\ x_5 &\to \mathsf{f}\ \mathsf{a}\ x_2\ x_3\ x_4\ x_5\\
\mathsf{a} &\to \mathsf{c} & \mathsf{f}\ x_1\ \mathsf{a}\ x_3\ x_4\ x_5 &\to \mathsf{f}\ x_1\ \mathsf{a}\ x_3\ x_4\ x_5\\
\mathsf{h}\ x\ x &\to \mathsf{h}\ x\ x & \mathsf{f}\ (y\ z)\ x_2\ x_3\ x_4\ x_5 &\to \mathsf{f}\ (y\ z)\ x_2\ x_3\ x_4\ x_5\\
& & \mathsf{f}\ x_1\ (y\ z)\ x_3\ x_4\ x_5 &\to \mathsf{f}\ x_1\ (y\ z)\ x_3\ x_4\ x_5
\end{aligned}$$

There is a minimal $(\mathcal{P},\mathcal{R})$-chain taking $s_i = \mathsf{g}^\sharp\ (\mathsf{f}\ x\ y\ z\ z\ u\ v)$, $t_i = \mathsf{g}^\sharp\ (\mathsf{f}\ x\ y\ x\ y\ x\ (\mathsf{h}\ y\ u))$, and $\sigma_i = \{x/\mathsf{k}(\mathsf{a}), y/\mathsf{k}(\mathsf{b}), z/\mathsf{k}(\mathsf{b}), u/\mathsf{k}(\mathsf{c}), v/\mathsf{h}\ (\mathsf{k}(\mathsf{b}))\ (\mathsf{k}(\mathsf{c}))\}$ where k is a unary symbol. However, there is no minimal $(\mathcal{P},\mathcal{R})$-chain using substitutions over the signature $\{\circ\}\cup\mathcal{C}$, regardless of the choice of constants $\mathcal{C}$.

Since our generalizations in Sect. 3 do not have any restrictions on the signature, we were able to correct the corresponding proofs in [8] such that the major theorems are still valid.[2] It follows the generalization of [8, Theorem 33].

Theorem 15. *The uncurrying processor* $\mathcal{U}'_1$ *is sound where* $\mathcal{U}'_1(\mathcal{P},\mathcal{R}) =$

$$\begin{cases} (\llcorner\mathcal{P}\lrcorner,\llcorner\mathcal{R}_\eta\lrcorner\cup\mathcal{U}) & \text{if } \mathcal{P}\cup\mathcal{R}_\eta \text{ is left head variable free and } \pi \text{ is injective,}\\ (\mathcal{P},\mathcal{R}) & \text{otherwise.}\end{cases}$$

Proof. The proof mainly uses the results from the previous section. We assume an infinite minimal $(\mathcal{P},\mathcal{R})$-chain $s_1\sigma_1 \to_{\mathcal{P}} t_1\sigma_1 \to^*_{\mathcal{R}} s_2\sigma_2 \to_{\mathcal{P}} t_2\sigma_2 \to^*_{\mathcal{R}} \cdots$ and construct an infinite minimal $(\llcorner\mathcal{P}\lrcorner,\llcorner\mathcal{R}_\eta\lrcorner\cup\mathcal{U})$-chain as follows.

[2] After the authors of [8] where informed of the gap, they independently developed an alternative fix, which is part of an extended, but not yet published version of [8].

We achieve $\llcorner s_i\sigma_i \lrcorner = \llcorner s_i \lrcorner \cdot \llcorner \sigma_i \lrcorner$ and $\llcorner t_i \lrcorner \cdot \llcorner \sigma_i \lrcorner \to^*_{\mathcal{U}} \llcorner t_i\sigma_i \lrcorner$ since $\mathcal{P}$ is left head variable free. Moreover, using $t_i\sigma_i \to^*_{\mathcal{R}} s_{i+1}\sigma_{i+1}$ and Theorem 10 we conclude $\llcorner t_i\sigma_i \lrcorner \to^*_{\llcorner \mathcal{R}_\eta \lrcorner \cup \mathcal{U}} \llcorner s_{i+1}\sigma_{i+1} \lrcorner$. Here, the condition that $\mathcal{R}_\eta$ must not contain variables as left-hand sides is ensured by the minimality of the chain: if $x \to r \in \mathcal{R}_\eta$ then $\mathsf{SN}_{\mathcal{R}}(t_i\sigma)$ does not hold. Hence, we constructed a $(\llcorner \mathcal{P} \lrcorner, \llcorner \mathcal{R}_\eta \lrcorner \cup \mathcal{U})$-chain as

$$\llcorner s_i\sigma_i \lrcorner = \llcorner s_i \lrcorner \cdot \llcorner \sigma_i \lrcorner \to_{\llcorner \mathcal{P} \lrcorner} \llcorner t_i \lrcorner \cdot \llcorner \sigma_i \lrcorner \to^*_{\mathcal{U}} \llcorner t_i\sigma_i \lrcorner \to^*_{\llcorner \mathcal{R}_\eta \lrcorner \cup \mathcal{U}} \llcorner s_{i+1}\sigma_{i+1} \lrcorner$$

for all i. To ensure that the chain is minimal it is demanded that π is injective. Otherwise, two different symbols can be mapped to the same new symbol which clearly can introduce nontermination. The structure of the proof that minimality is preserved is similar to the one in [8] and we just refer to IsaFoR for details.

The uncurrying processor of Theorem 15 generalizes [8, Theorem 33] in three ways: the signature does not have to be applicative, we can freely choose the applicative arity via π, and we can freely choose the application symbol. The last generalization lets Theorem 15 almost subsume the technique of freezing [8, Corollary 40] which is used to uncurry $\circ^\sharp$.

Definition 16 (Freezing [8]). *A* simple freeze $❄$ *is a subset of* $\mathcal{F}$.[3] Freezing *is applied on non-variable terms as follows*

$$❄(f(t_1, \ldots, t_n)) = \begin{cases} f(t_1, \ldots, t_n) & \text{if } n = 0 \text{ or } f \notin ❄ \\ f^g(s_1, \ldots, s_m, t_2, \ldots, t_m) & \text{if } t_1 = g(s_1, \ldots, s_m) \text{ and } f \in ❄ \end{cases}$$

where each f^g *is a new symbol. It is homomorphically extended to rules and TRSs. The freezing DP processor is defined as* $❄(\mathcal{P}, \mathcal{R}) =$

$$\begin{cases} (❄(\mathcal{P}), \mathcal{R}) & \text{if } (\mathcal{P}, \mathcal{R}) \text{ is a standard DP problem where for all } s \to f(t_1, \ldots, t_n) \\ & \in \mathcal{P} \text{ with } f \in ❄, \text{ both } t_1 \notin \mathcal{V} \text{ and all instances of } t_1 \text{ are root-stable,} \\ (\mathcal{P}, \mathcal{R}) & \text{otherwise.} \end{cases}$$

In [8, Theorem 39], it is shown that freezing is sound.

Example 17. In the following we use numbers to refer to rules from previous examples. We consider the DP problem $(\mathcal{P}, \mathcal{R})$ where $\mathcal{P} = \{\mathsf{sub}\ (\mathsf{s}\ x)^\sharp (\mathsf{s}\ y) \to \mathsf{sub}\ x^\sharp\ y\}$ and $\mathcal{R} = \{2\text{–}4\}$. Uncurrying $\circ$ with $\pi(\mathsf{s}) = [\mathsf{s}, \mathsf{s}_1]$, $\pi(\mathsf{sub}) = [\mathsf{sub}, \mathsf{sub}_1, \mathsf{sub}_2]$, $\pi(\mathsf{0}) = [\mathsf{0}]$, and $\pi(^\sharp) = [^\sharp]$ yields the DP problem $(\llcorner \mathcal{P} \lrcorner, \llcorner \mathcal{R}_\eta \lrcorner \cup \mathcal{U})$ where $\llcorner \mathcal{P} \lrcorner = \{\mathsf{sub}_1(\mathsf{s}_1(x))\ ^\sharp\ \mathsf{s}_1(y) \to \mathsf{sub}_1(x)\ ^\sharp\ y\}$ and $\llcorner \mathcal{R}_\eta \lrcorner \cup \mathcal{U}$ consists of {10–12, 16, 19, 20}.

Afterwards we uncurry the resulting DP problem using $\circ^\sharp$ as application symbol and π where $\pi(\mathsf{sub}_1) = [\mathsf{sub}_1, -^\sharp]$ and $\pi(f) = [f]$ for all other symbols. We obtain $(\mathcal{P}', \mathcal{R}')$ where $\mathcal{P}' = \{\mathsf{s}_1(x)\ -^\sharp\ \mathsf{s}_1(x)) \to x\ -^\sharp\ y\}$ and $\mathcal{R}' = \llcorner \mathcal{R}_\eta \lrcorner \cup \mathcal{U} \cup \{\mathsf{sub}_1(x)\ ^\sharp\ y \to -^\sharp(x, y)\}$. Note that freezing returns nearly the same DP problem. The only difference is that uncurrying produces the additional rule $\mathsf{sub}_1(x)\ ^\sharp\ y \to x\ -^\sharp\ y$ which we do not obtain via freezing. However, since this rule is not usable it also does not harm that much.

[3] In [8] one can also specify an argument position for each symbol. This can be simulated by permuting the arguments accordingly.

Moreover, uncurrying sometimes is applicable where freezing is not. If we would have started with the DP problem $(\mathcal{P}, \mathcal{R}'')$ where $\mathcal{R}'' = \{1\text{–}5\}$ then uncurrying would result in $(\llcorner\mathcal{P}\lrcorner, \llcorner\mathcal{R}''_\eta\lrcorner \cup \mathcal{U}')$ where $\llcorner\mathcal{R}''_\eta\lrcorner \cup \mathcal{U}' = \{8\text{–}13, 16\text{–}20\}$. On this DP problem freezing is not applicable (the instances of $\mathsf{sub}_1(x)$ in the right-hand side of the only pair in $\llcorner\mathcal{P}\lrcorner$ are not root-stable due to Rule 8). Nevertheless, one can uncurry $\circ^\sharp$, resulting in $(\mathcal{P}', \llcorner\mathcal{R}''_\eta\lrcorner \cup \mathcal{U}' \cup \mathcal{R}_{\mathsf{new}})$ where $\mathcal{R}_{\mathsf{new}} = \{\mathsf{sub}_1(x) \,\sharp\, y \to x -^\sharp y, \mathsf{0} -^\sharp y \to \mathsf{K}_1(\mathsf{0}) \,\sharp\, y\}$. Note that the uncurrying of $\circ^\sharp$ transformed a standard DP problem into a non-standard one, as $-^\sharp$ occurs as root of a term in $\mathcal{P}'$, but also within $\mathcal{R}_{\mathsf{new}}$.

Whenever freezing with $❄ = \{\circ^\sharp\}$ is applicable, then also uncurrying of $\circ^\sharp$ is possible: the condition $t_1 \notin \mathcal{V}$ in Definition 16 implies that $\mathcal{P} \cup \mathcal{R}_\eta$ is left head variable free. The only difference is that uncurrying produces more rules than freezing, namely the uncurrying rules and the uncurried rules of those rules which have to be added for the η-saturation. However, if freezing is applicable then none of these additional rules are usable.[4] Hence, all techniques which only consider the usable rules (like the reduction pair processor) perform equally well, no matter whether one has applied freezing or uncurrying. Still, one wants to get rid of the additional rules, especially since they are also the reason why standard DP problems are transformed into non-standard ones.

In Example 17 we have seen that sometimes uncurrying of tuple symbols is applicable where freezing is not. Thus, to have the best of both techniques we devised a special uncurrying technique for tuple symbols which fully subsumes freezing without the disadvantage of $\mathcal{U}'_1$: if freezing is applicable then standard DP problems are transformed into standard DP problems by the new technique.

Before we describe the new uncurrying processor formally, we shortly list the differences to the uncurrying processor of Theorem 15:

- Since the task is to uncurry tuple symbols, we restrict the applicative arities to be at most one. Moreover, uncurrying is performed only on the top-level. Finally, the application symbol may be of arbitrary non-zero arity.
- Rules that have to be added for the η-saturation and the uncurrying rules are added as pairs (to the $\mathcal{P}$-component), and not as rules (to the $\mathcal{R}$-component).
- If freezing is applicable, we do neither add the uncurrying rules nor do we apply η-saturation.

Example 18. We continue with the DP problems of Example 17.

If one applies the special uncurrying processor on $(\llcorner\mathcal{P}\lrcorner, \llcorner\mathcal{R}_\eta\lrcorner \cup \mathcal{U})$ then one obtains $(\mathcal{P}', \llcorner\mathcal{R}_\eta\lrcorner \cup \mathcal{U})$ which is the same as $❄(\llcorner\mathcal{P}\lrcorner, \llcorner\mathcal{R}_\eta\lrcorner \cup \mathcal{U})$ for $❄ = \{\circ^\sharp\}$.

And if one applies the special uncurrying processor on $(\llcorner\mathcal{P}\lrcorner, \llcorner\mathcal{R}''_\eta\lrcorner \cup \mathcal{U}')$ then one obtains the standard DP problem $(\mathcal{P}' \cup \mathcal{R}_{\mathsf{new}}, \llcorner\mathcal{R}''_\eta\lrcorner \cup \mathcal{U}')$.

Definition 19. *Let $\circ^\sharp$ be an n-ary application symbol where $n > 0$. Let π be an injective symbol map where $\pi(f) \in \{[f], [f, f^\sharp]\}$ for all f. The top-uncurrying function $\ulcorner\cdot\urcorner$ maps terms to terms. It is defined as $\ulcorner t \urcorner =$*

[4] In detail: a technique that can detect that instances of a subterm of a right-hand side of $\mathcal{P}$ are root-stable can also detect that the additional rules are not usable.

$$\begin{cases} f^\sharp(s_1,\ldots,s_m,t_2,\ldots,t_n) & \text{if } t = \circ^\sharp(f(s_1,\ldots,s_m),t_2,\ldots,t_n) \text{ and } \pi(f) = [f,f^\sharp] \\ t & \text{otherwise} \end{cases}$$

and is homomorphically extended to pairs, rules, and substitutions. The top-uncurrying rules *are defined as*

$$\mathcal{U}^t = \{\circ^\sharp(f(x_1,\ldots,x_m),y_2,\ldots,y_n) \to f^\sharp(x_1,\ldots,x_m,y_2,\ldots,y_n) \mid \pi(f) = [f,f^\sharp]\}$$

Then the top-uncurrying processor *is defined as* $\mathsf{top}(\mathcal{P},\mathcal{R}) =$

$$\begin{cases} (\ulcorner\mathcal{P}_\eta\urcorner \cup \mathcal{U}^t_?, \mathcal{R}) & \text{if } \circ^\sharp \text{ is not defined w.r.t. } \mathcal{R} \text{ and for all } s \to t \in \mathcal{P}_\eta : s,t \notin \mathcal{V}, \\ & \quad s \neq \circ^\sharp(x,s_2,\ldots,s_n), \text{ and the root of } t \text{ is not defined w.r.t. } \mathcal{R} \\ (\mathcal{P},\mathcal{R}) & \text{otherwise} \end{cases}$$

where $\mathcal{U}^t_? = \emptyset$ *and* $\mathcal{P}_\eta = \mathcal{P}$ *if for all* $s \to \circ^\sharp(t_1,\ldots,t_n) \in \mathcal{P}$ *and* σ *the term* $t_1\sigma$ *is root-stable, and* $\mathcal{U}^t_? = \mathcal{U}^t$ *and* $\mathcal{P}_\eta = \mathcal{P} \cup \{\circ^\sharp(\ell,x_2,\ldots,x_n) \to \circ^\sharp(r,x_2,\ldots,x_n) \mid \ell \to r \in \mathcal{R}, \mathrm{root}(\ell) = g, \pi(g) = [g,g^\sharp]\}$*, otherwise. Here,* $x_2,\ldots,x_n$ *are distinct fresh variables that do not occur in* $\ell \to r$*.*

Theorem 20. *The top-uncurrying processor* top *is sound.*

Proof. The crucial part is to prove that whenever $t = f(t_1,\ldots,t_m) \to^*_{\mathcal{R}} s$, $f \notin \mathcal{D}_{\mathcal{R}}$, t is an instance of a right-hand side of $\mathcal{P}$, and $\mathsf{SN}_{\mathcal{R}}(t)$, then $\ulcorner t\urcorner \to^*_{\ulcorner\mathcal{P}_\eta\urcorner \cup \mathcal{U}^t_? \cup \mathcal{R}} \ulcorner s\urcorner$ where $\ulcorner\mathcal{P}_\eta\urcorner \cup \mathcal{U}^t_?$-steps are root steps and all terms in this derivation are terminating w.r.t. $\mathcal{R}$.

Using this result, the main result is established as follows. Assume there is an infinite minimal $(\mathcal{P},\mathcal{R})$-chain. Then every step $s\sigma \to_{\mathcal{P}} t\sigma \to^*_{\mathcal{R}} s'\sigma'$ in the chain is transformed as follows. Since $s \to t \in \mathcal{P}$, we conclude that $t\sigma = f(t_1\sigma,\ldots,t_m\sigma)$ where $f \notin \mathcal{D}_{\mathcal{R}}$ and $\mathsf{SN}_{\mathcal{R}}(t\sigma)$. Hence, using the crucial step we know that $\ulcorner t\sigma\urcorner \to^*_{\ulcorner\mathcal{P}_\eta\urcorner \cup \mathcal{U}^t_? \cup \mathcal{R}} \ulcorner s'\sigma'\urcorner$. Moreover, by case analysis on t one can show that $\ulcorner t\urcorner\sigma \to^*_{\mathcal{U}^t_?} \ulcorner t\sigma\urcorner$ via root reductions, and similarly, by additionally using the restrictions on s one derives $\ulcorner s\sigma\urcorner = \ulcorner s\urcorner\sigma$. Hence,

$$\ulcorner s\sigma\urcorner = \ulcorner s\urcorner\sigma \to_{\ulcorner\mathcal{P}\urcorner} \ulcorner t\urcorner\sigma \to^*_{\mathcal{U}^t_?} \ulcorner t\sigma\urcorner \to^*_{\ulcorner\mathcal{P}_\eta\urcorner \cup \mathcal{U}^t_? \cup \mathcal{R}} \ulcorner s'\sigma'\urcorner$$

where all terms in this derivation right of $\to_{\ulcorner\mathcal{P}\urcorner}$ are terminating w.r.t. $\mathcal{R}$ and where all $\ulcorner\mathcal{P}_\eta\urcorner \cup \mathcal{U}^t_?$-steps are root reductions. Thus, we can turn the root reductions into pairs, resulting in an infinite minimal $(\ulcorner\mathcal{P}_\eta\urcorner \cup \mathcal{U}^t_?, \mathcal{R})$-chain.

To prove the crucial part we perform induction on the number of steps where the base case – no reductions – is trivial. Otherwise, $t = f(t_1,\ldots,t_m) \to^*_{\mathcal{R}} u \to_{\mathcal{R}} s$. Using $\mathsf{SN}_{\mathcal{R}}(t)$ we also know $\mathsf{SN}_{\mathcal{R}}(u)$ and since $f \notin \mathcal{D}_{\mathcal{R}}$ we know that $u = f(u_1,\ldots,u_m)$ and $t_i \to^*_{\mathcal{R}} u_i$ for all $1 \leqslant i \leqslant m$. Moreover, $s = f(s_1,\ldots,s_m)$ and s is obtained from u by a reduction $u_i \to_{\mathcal{R}} s_i$ for some $1 \leqslant i \leqslant m$. Hence, we may apply the induction hypothesis and conclude $\ulcorner t\urcorner \to^*_{\ulcorner\mathcal{P}_\eta\urcorner \cup \mathcal{U}^t_? \cup \mathcal{R}} \ulcorner u\urcorner$.

It remains to simulate the reduction $u \to_{\mathcal{R}} s$. The simulation is easy if $f \neq \circ^\sharp$, since then $\ulcorner u\urcorner = u \to_{\mathcal{R}} s = \ulcorner s\urcorner$. Otherwise, $f = \circ^\sharp$ and $m = n$. We again first

consider the easy case where $u_i \to_{\mathcal{R}} s_i$ for some $i > 1$. Then an easy case analysis on u_1 yields $\ulcorner u \urcorner \to_{\mathcal{R}} \ulcorner s \urcorner$ since u and s are uncurried in the same way (since $u_1 = s_1$). Otherwise, $u = \circ^\sharp(u_1, \ldots, u_m)$, $s = \circ^\sharp(s_1, u_2, \ldots, u_m)$ and $u_1 \to_{\mathcal{R}} s_1$. If $u_1 \to_{\mathcal{R}} s_1$ is a reduction below the root then both s and t are uncurried in the same way and again $\ulcorner u \urcorner \to_{\mathcal{R}} \ulcorner s \urcorner$ is easily established. If however $u_1 = \ell\sigma \to r\sigma = s_1$ for some rule $\ell \to r \in \mathcal{R}$ then we know that u_1 is not root-stable and hence also t_1 is not root-stable. As $t = \circ^\sharp(t_1, \ldots, t_n)$ is an instance of a right-hand side of $\mathcal{P}$ we further know that there is a pair $s' \to \circ^\sharp(t'_1, \ldots, t'_n) \in \mathcal{P}$ where $t_1 = t'_1\sigma$. Since $t'_1\sigma$ is not root-stable $\mathcal{U}^t_? = \mathcal{U}^t$ and $\mathcal{P}_\eta \supseteq \{\circ^\sharp(\ell, x_2, \ldots, x_n) \to \circ^\sharp(r, x_2, \ldots, x_n) \mid \ell \to r \in \mathcal{R}, \mathrm{root}(\ell) = g, \pi(g) = [g, g^\sharp]\}$. Let $\ell = g(\ell_1, \ldots, \ell_k)$. If $\pi(g) = [g]$ then

$$\begin{aligned}
\ulcorner u \urcorner &= \ulcorner \circ^\sharp(g(\ell_1, \ldots, \ell_k)\sigma, u_2, \ldots, u_n) \urcorner \\
&= \circ^\sharp(g(\ell_1, \ldots, \ell_k)\sigma, u_2, \ldots, u_n) \\
&\to_{\mathcal{R}} \circ^\sharp(r\sigma, u_2, \ldots, u_n) \\
&\to^*_{\mathcal{U}^t_?} \ulcorner \circ^\sharp(r\sigma, u_2, \ldots, u_n) \urcorner \\
&= \ulcorner s \urcorner.
\end{aligned}$$

And otherwise, $\pi(g) = [g, g^\sharp]$. Hence, $\circ^\sharp(\ell, x_2, \ldots, x_n) \to \circ^\sharp(r, x_2, \ldots, x_n) \in \mathcal{P}_\eta$. We define $\delta = \sigma \uplus \{x_2/u_2, \ldots, x_n/u_n\}$ and achieve

$$\begin{aligned}
\ulcorner u \urcorner &= \ulcorner \circ^\sharp(g(\ell_1, \ldots, \ell_k)\sigma, u_2, \ldots, u_n) \urcorner \\
&= g^\sharp(\ell_1\sigma, \ldots, \ell_k\sigma, u_2, \ldots, u_n) \\
&= g^\sharp(\ell_1, \ldots, \ell_k, x_2, \ldots, x_n)\delta \\
&= \ulcorner \circ^\sharp(g(\ell_1, \ldots, \ell_k), x_2, \ldots, x_n) \urcorner\delta \\
&= \ulcorner \circ^\sharp(\ell, x_2, \ldots, x_n) \urcorner\delta \\
&\to_{\ulcorner \mathcal{P}_\eta \urcorner} \ulcorner \circ^\sharp(r, x_2, \ldots, x_n) \urcorner\delta \\
&\to^*_{\mathcal{U}^t_?} \ulcorner \circ^\sharp(r, x_2, \ldots, x_n)\delta \urcorner \\
&= \ulcorner \circ^\sharp(r\sigma, u_2, \ldots, u_n) \urcorner \\
&= \ulcorner s \urcorner.
\end{aligned}$$

Using that π is injective one can also show that termination of all terms in the derivation is guaranteed where we refer to our library IsaFoR for details.

Note that the top-uncurrying processor fully subsumes freezing since the step from $(\mathcal{P}, \mathcal{R})$ to $(\mathfrak{B}(\mathcal{P}), \mathcal{R})$ using $\mathfrak{B} = \{f_1, \ldots, f_n\}$ can be simulated by n applications of top where in each iteration one chooses f_i as application symbol and defines $\pi(g) = [g, f_i^g]$ for all $g \neq f_i$. The following example shows that top is also useful where freezing is not applicable.

Example 21. Consider the TRS $\mathcal{R}$ where $x \div y$ computes $\lceil \frac{x}{2y} \rceil$.

$$\begin{aligned}
\mathsf{s}(x) - \mathsf{s}(y) &\to x - y & \mathsf{double}(x) &\to x + x\\
0 - y &\to 0 & \mathsf{double}(0) &\to 0\\
x - 0 &\to x & \mathsf{double}(\mathsf{s}(x)) &\to \mathsf{s}(\mathsf{s}(\mathsf{double}(x)))\\
0 + y &\to y & 0 \div \mathsf{s}(y) &\to 0\\
\mathsf{s}(x) + y &\to \mathsf{s}(x + y) & \mathsf{s}(x) \div \mathsf{s}(y) &\to \mathsf{s}((\mathsf{s}(x) - \mathsf{double}(\mathsf{s}(y))) \div \mathsf{s}(y))
\end{aligned}$$

Proving termination is hard for current termination provers. Let us consider the interesting DP problem $(\mathcal{P}, \mathcal{R})$ where $\mathcal{P} = \{\mathsf{s}(x) \div^\sharp \mathsf{s}(y) \to (\mathsf{s}(x) - \mathsf{double}(\mathsf{s}(y))) \div^\sharp \mathsf{s}(y)\}$. The problem is that one cannot use standard reduction pairs with argument filters since one has to keep the first argument of $-$, and then the filtered term of $\mathsf{s}(x)$ is embedded in the filtered term of $\mathsf{s}(x) - \mathsf{double}(\mathsf{s}(y))$. Consequently, powerful termination provers such as AProVE and T$_\text{T}$T$_2$ fail on this TRS.

However, one can uncurry the tuple symbol $\div^\sharp$ where $\pi(-) = [-, -^\sharp]$, $\pi(\mathsf{s}) = [\mathsf{s}, \mathsf{s}^\sharp]$, and $\pi(f) = [f]$, otherwise. Then the new DP problem $(\mathcal{P}', \mathcal{R})$ is created where $\mathcal{P}'$ consists of the following pairs

$$\begin{aligned}
(x - y) \div^\sharp z &\to -^\sharp(x, y, z) & -^\sharp(\mathsf{s}(x), \mathsf{s}(y), z) &\to -^\sharp(x, y, z)\\
\mathsf{s}(x) \div^\sharp y &\to \mathsf{s}^\sharp(x, y) & -^\sharp(0, y, z) &\to 0 \div^\sharp z\\
\mathsf{s}^\sharp(x, \mathsf{s}(y)) &\to -^\sharp(\mathsf{s}(x), \mathsf{double}(\mathsf{s}(y)), \mathsf{s}(y)) & -^\sharp(x, 0, z) &\to x \div^\sharp z
\end{aligned}$$

where the subtraction is computed via the new pairs, and not via the rules anymore. The right column consists of the uncurried and η-saturated $-$-rules, and the left column contains the two uncurrying rules followed by the uncurried pair of $\mathcal{P}$. Proving finiteness of this DP problem is possible using standard techniques: linear 0/1-polynomial interpretations and the dependency graph suffice. Therefore, termination of the whole example can be proven fully automatically by using a new version of T$_\text{T}$T$_2$ where top-uncurrying is integrated.

5 Heuristics and Experiments

The generalizations for uncurrying described in this paper are implemented in T$_\text{T}$T$_2$ [10]. To fix the symbol map we used the following three heuristics:

- π_+ corresponds to the definition of applicative arity of [8]. More formally, $\pi_+(f) = [f_0, \ldots, f_n]$ where n is maximal w.r.t. all $f(\ldots) \circ t_1 \circ \cdots \circ t_n$ occurring in $\mathcal{R}$. The advantage of π_+ is that all uncurryings are performed.
- $\pi_\pm$ is like π_+, except that the applicative arity is reduced whenever we would have to add a rule during η-saturation. Formally, $\pi_\pm(f) = [f_0, \ldots, f_n]$ where $n = \min(\mathsf{aa}_{\pi_+}(f), \min\{k \mid f(\ldots) \circ t_1 \circ \cdots \circ t_k \to r \in \mathcal{R}\})$.
- π_- is almost dual to π_+. Formally, $\pi_-(f) = [f_0, \ldots, f_n]$ where n is minimal w.r.t. all maximal subterms of the shape $f(\ldots) \circ t_1 \circ \cdots \circ t_n$ occurring in $\mathcal{R}$. The idea is to reduce the number of uncurrying rules.

We conducted two sets of experiments to evaluate our work. Note that all proofs generated during our experiments are certified by CeTA (version 1.18). Our experiments were performed on a server with eight dual-core AMD Opteron® processors 885, running at a clock rate of 2.6 GHz and on 64 GB of main memory. The time limit for the first set of experiments was 10 s (as in [8]), whereas

Table 1. Experiments as in [8]

	direct		processor	
	none	trs	$\mathcal{U}'_1$	$\mathcal{U}'_2$
subterm criterion	41	53	41	66
matrix (dimension 1)	66	98	95	114
matrix (dimension 2)	108	137	133	138

Table 2. Newly certified proofs

	direct	processor		total
	trs	$\mathcal{U}'_1$	$\mathcal{U}'_2$	
π_+	26	16	22	35
$\pi_\pm$	28	15	17	29
π_-	24	14	14	24
total	28	16	24	36

the time limit for the second set was 5 s (T$_\text{T}$T$_2$'s time limit in the termination competition).

The first set of experiments was run with a setup similar to [8]. Accordingly, as input we took the same 195 ATRSs from the termination problem database (TPDB). For proving termination, we switch from the input TRS to the initial DP problem and then repeat the following as often as possible: compute the estimated dependency graph, split it into its strongly connected components and apply the "main processor." Here, as "main processor" we incorporated the subterm criterion and matrix interpretations (of dimensions one and two). Concerning uncurrying, the following approaches were tested: no uncurrying (none), uncurry the given TRS before computing the initial DP problem (trs), apply $\mathcal{U}'_1/\mathcal{U}'_2$ as soon as all other processors fail (where $\mathcal{U}'_2$ is the composition of $\mathcal{U}'_1$ and top). The results can be found in Table 1. Since on ATRSs, our generalization of uncurrying corresponds to standard uncurrying, it is not surprising that the numbers of the first three columns coincide with those of [8] (modulo *mirroring* and a slight difference in the used strategy for trs). They are merely included to see the relative gain when using uncurrying on ATRSs.

With the second set of experiments, we tried to evaluate the total gain in certified termination proofs. Therefore, we took a restricted version of T$_\text{T}$T$_2$'s competition strategy that was used in the July 2010 issue of the international termination competition[5] (called *base strategy* in the following). The restriction was to use only those termination techniques that where certifiable by CeTA before our formalization of uncurrying. Then, we used this base strategy to filter the TRSs (we did ignore all SRSs) of the TPDB (version 8.0). The result were 511 TRSs for which T$_\text{T}$T$_2$ did neither generate a termination proof nor a non-termination proof using the base strategy. For our experiments we extended the base strategy by the generalized uncurrying techniques using different heuristics for the applicative arity. The results can be found in Table 2. It turned out, that the π_- heuristic is rather weak. Concerning $\pi_\pm$, there is at least one TRS that could not be proven using π_+, but with $\pi_\pm$. The total of 35 in the first row of Table 2 is already reached without taking $\mathcal{U}'_1$ into account. This indicates that in practice a combination of uncurrying as initial step (trs) and the processor $\mathcal{U}'_2$, gives the best results. Finally, note that in comparison to the July 2010

[5] http://termcomp.uibk.ac.at

termination competition (where TTT2 could generate 262 certifiable proofs), the number of certifiable proofs of TTT2 is increased by over 10 % using the new techniques. In these experiments, termination has been proven for 10 *non-applicative* TRSs where our generalizations of uncurrying have been the key to success.

6 Conclusions

This paper describes the first formalization of uncurrying, an important technique to prove termination of higher-order functions which are encoded as first-order TRSs. The formalization revealed a gap in the original proof which is now fixed. Adding the newly developed generalization of uncurrying to our certifier CeTA, increased the number of certifiable proofs on the TPDB by 10 %.

References

1. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press, Cambridge (1999)
2. Blanqui, F., Delobel, W., Coupet-Grimal, S., Hinderer, S., Koprowski, A.: CoLoR, a Coq library on rewriting and termination. In: WST 2006, pp. 69–73 (2006)
3. Contejean, É., Paskevich, A., Urbain, X., Courtieu, P., Pons, O., Forest, J.: A3PAT, an approach for certified automated termination proofs. In: Gallagher, J.P., Voigtländer, J. (eds.) PEPM 2010, pp. 63–72. ACM Press, New York (2010)
4. Endrullis, J.: Jambox, http://joerg.endrullis.de
5. Giesl, J., Thiemann, R., Schneider-Kamp, P., Falke, S.: Mechanizing and improving dependency pairs. Journal of Automated Reasoning 37(3), 155–203 (2006)
6. 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)
7. Giesl, J., Schneider-Kamp, P., Thiemann, R.: AProVE 1.2: automatic termination proofs in the dependency pair framework. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 281–286. Springer, Heidelberg (2006)
8. Hirokawa, N., Middeldorp, A., Zankl, H.: Uncurrying for termination. In: Cervesato, I., Veith, H., Voronkov, A. (eds.) LPAR 2008. LNCS (LNAI), vol. 5330, pp. 667–681. Springer, Heidelberg (2008)
9. Kennaway, R., Klop, J.W., Sleep, R., de Vries, F.J.: Comparing curried and uncurried rewriting. Journal of Symbolic Computation 21(1), 15–39 (1996)
10. 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)
11. Nipkow, T., Paulson, L.C., Wenzel, M.T.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002)
12. Sternagel, C., Thiemann, R.: Signature extensions preserve termination. In: Dawar, A., Veith, H. (eds.) CSL 2010. LNCS, vol. 6247, pp. 514–528. Springer, Heidelberg (2010)
13. Thiemann, R.: The DP Framework for Proving Termination of Term Rewriting. Ph.D. thesis, RWTH Aachen University, Technical Report AIB-2007-17 (2007)
14. Thiemann, R., Sternagel, C.: Certification of termination proofs using CeTA. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 452–468. Springer, Heidelberg (2009)

A Semantic Account for Modularity in Multi-language Modelling of Search Problems

Shahab Tasharrofi and Eugenia Ternovska

Simon Fraser University, Canada
{sta44,ter}@cs.sfu.ca

Abstract. Motivated by the need to combine systems and logics, we develop a modular approach to the model expansion (MX) problem, a task which is common in applications such as planning, scheduling, computational biology, formal verification. We develop a modular framework where parts of a modular system can be written in different languages. We start our development from a previous work, [14], but modify and extend that framework significantly. In particular, we use a model-theoretic setting and introduce a feedback (loop) operator on modules. We study the expressive power of our framework and demonstrate that adding the feedback operator increases the expressive power considerably. We prove that, even with individual modules being polytime solvable, the framework is expressive enough to capture all of NP, a property which does not hold without loop. Moreover, we demonstrate that, using monotonicity and anti-monotonicity of modules, one can significantly reduce the search space of a solution to a modular system.

1 Introduction

Formulating AI tasks as model finding has recently become very promising due to the overwhelming success of SAT solvers and related technology such as SMT. In our research direction we focus on a particular kind of model finding which we call *model expansion*. The task of model expansion underlies all search problems where for an instance of a problem, which we represent as a logical structure, one needs to find a certificate (solution) satisfying certain specification. For example, given a graph, we are looking for its 3-colouring in a classic NP-search problem. Such search problems occur broadly in applications; they include planning, scheduling, problems in formal verification (where we are looking for a path to a bug), computational biology, and so on. In addition to being quite common, the task of model expansion is generally simpler than satisfiability from the computational point of view. Indeed, for a given logic $\mathcal{L}$, we have, in terms of computational complexity,

$$\text{MC}(\mathcal{L}) \leq \text{MX}(\mathcal{L}) \leq \text{Satisfiability}(\mathcal{L}),$$

where $\text{MC}(\mathcal{L})$ stands for model checking (structure for the entire vocabulary of the formula in logic $\mathcal{L}$ is given), $\text{MX}(\mathcal{L})$ stands for model expansion (structure interpreting a part of the vocabulary is given) and $\text{Satisfiability}(\mathcal{L})$ stands for satisfiability task (where we are looking for a structure satisfying the formula). A comparison of the complexity

C. Tinelli and V. Sofronie-Stokkermans (Eds.): FroCoS 2011, LNAI 6989, pp. 259–274, 2011.

of the three tasks for several logics of practical interest is given in [15]. Satisfiability problem has been studied for many logics of practical interest, however model expansion problem has not been studied. In particular, issues related to combining specialized formalisms have been investigated, to a large degree, for satisfiability problem but not at all for model expansion. As we develop our framework, we aim at understanding the expressive power of the operations we add. Knowing the expressiveness of a framework is essential in particular to understanding the complexity of solution finding.

Our contributions are as follows:

- We develop a semantics-based formalism which abstractly represents combinations of modules. Our model-theoretic view allows us to study modular systems independently from the logical languages in which each module is axiomatized.
- In [14], the authors define a modular constraint-based framework where different modelling languages such as ASP, CP and SAT can be combined. We considerably extend their work mostly due to to the introduction of loops and results that follow.
- Unlike [14], we represent modules as sets of structures, each such set corresponding to a model expansion task solved by a module. This model-theoretic view is essential (1) to study the expressiveness of the framework itself, and its expressiveness as a function of the expressiveness of the languages of individual modules; (2) to connect to descriptive complexity (capturing complexity classes). In both cases, the constraint-based approach [14] is not suitable – one needs to talk about formulas being true in a structure, thus the model-theoretic view.
- We formulate an algebra on our modules (module expansion tasks). Several algebraic operations have already been used in [14], although in a constraint setting. An essential contribution here is the addition of a loop (or feedback) operator. Loops are present in all non-trivial computer programs and systems, including those consisting of multiple modules. In all the results in this paper, the loop operator is essential.
- We then investigate the expressive power of modular systems. We set apart and study the expressive power which is added purely by the algebraic operations. Among the operations, the loop operator is the most interesting. Adding loops gives a jump from P to NP: we prove that NP is captured even with all modules being polytime, due to the loop operator. In fact, adding it gives a jump in the polynomial time hierarchy. The operators introduced in [14] do not add additional expressive power, while the loop operator does.
- A crucial question is how to compute solution to the modular system under the assumption that we can compute solutions of individual modules. We begin our investigation of this question. We study some cases where solution to a modular system can be approximated in polynomial time by relying on the construction used in the well-founded semantics of logic programs.
- In many cases, we can view modules as operators. We consider monotonicity and antimonotonocity properties of modules viewed as operators. These are important properties because they allow us to derive some knowledge about solutions to the entire modular system.

2 Background: Model Expansion Task

In [17], the authors formalize combinatorial search problems as the task of *model expansion (MX)*, the logical task of expanding a given (mathematical) structure with new relations. Formally, the user axiomatizes their problem in some logic $\mathcal{L}$. This axiomatization relates an instance of the problem (a *finite structure*, i.e., a universe together with some relations and functions), and its solutions (certain *expansions* of that structure with new relations or functions). Logic $\mathcal{L}$ corresponds to a specification/modelling language. It could be an extension of first-order logic such as FO(ID), or an ASP language, or a modelling language from the CP community such as ESSENCE [12]. MX task underlies many practical approaches to declarative problem solving, which motivates us to investigate modularity in the context of the MX task.

Recall that a vocabulary is a set of non-logical (predicate and function) symbols. An interpretation for a vocabulary is provided by a *structure*, which consists of a set, called the domain or universe and denoted by $dom(.)$, together with a collection of relations and (total) functions over the universe. A structure can be viewed as an *assignment* to the elements of the vocabulary. An expansion of a structure $\mathcal{A}$ is a structure $\mathcal{B}$ with the same universe, and which has all the relations and functions of $\mathcal{A}$, plus some additional relations or functions. The task of model expansion for an arbitrary logic $\mathcal{L}$ (abbreviated $\mathcal{L}$-MX), is:

Model Expansion for logic $\mathcal{L}$
Given: *(1)* An $\mathcal{L}$-formula ϕ with vocabulary $\sigma \cup \varepsilon$ and *(2)* A structure $\mathcal{A}$ for σ
Find: an expansion of $\mathcal{A}$, to $\sigma \cup \varepsilon$, that satisfies ϕ.

We call σ, the vocabulary of $\mathcal{A}$, the *instance* vocabulary, and $\varepsilon := vocab(\phi) \setminus \sigma$ the *expansion* vocabulary[1].

Example 1. The following formula ϕ of first order logic constitutes an MX specification for Graph 3-colouring:

$$\begin{array}{c} \forall x\,[(R(x) \vee B(x) \vee G(x)) \\ \wedge\neg((R(x) \wedge B(x)) \vee (R(x) \wedge G(x)) \vee (B(x) \wedge G(x)))] \\ \wedge\, \forall x \forall y\,[E(x,y) \supset (\neg(R(x) \wedge R(y)) \\ \wedge\neg(B(x) \wedge B(y)) \wedge \neg(G(x) \wedge G(y)))]. \end{array}$$

An instance is a structure for vocabulary $\sigma = \{E\}$, i.e., a graph $\mathcal{A} = \mathcal{G} = (V; E)$. The task is to find an interpretation for the symbols of the expansion vocabulary $\varepsilon = \{R, B, G\}$ such that the expansion of $\mathcal{A}$ with these is a model of ϕ:

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

The interpretations of ε, for structures $\mathcal{B}$ that satisfy ϕ, are exactly the proper 3-colourings of $\mathcal{G}$.

Given a specification, we can talk about a set (class) of $\sigma \cup \varepsilon$-structures which satisfy the specification. Alternatively, we can simply talk about a set (class) of $\sigma \cup \varepsilon$-structures as an MX-task, without mentioning a particular specification the structures satisfy.

[1] By ":=" we mean "is by definition" or "denotes".

3 Modular Systems

Definition 1 (Module). *A* module M *is an MX task, i.e., a set (class) of* $\sigma \cup \varepsilon$*-structures.*

Characterizing a module using a set of structures does not assume anything about how it is specified, which makes our study language-independent. A modular system is formally described as a set of primitive modules (individual MX tasks) combined using the operations of: (1) Projection($\pi_\tau(M)$) which restricts the vocabulary of a module, (2) Composition($M_1 \rhd M_2$) which connects outputs of M_1 to inputs of M_2, (3) Union($M_1 \cup M_2$) and (4) Feedback($M[R = S]$). Operations (1)-(3) were introduced in [14], in a constraint setting. Here, we use a model-theoretic setting. The feedback operation is new here, and it is essential since all non-trivial systems use loops. Moreover, adding this operation increases the expressive power of modular systems.

Definition 2 (Composable, Independent [14]). *Modules* M_1 *and* M_2 *are* composable *if* $\varepsilon_{M_1} \cap \varepsilon_{M_2} = \emptyset$ *(no output interference). Module* M_1 *is* independent *from* M_2 *if* $\sigma_{M_1} \cap \varepsilon_{M_2} = \emptyset$ *(no cyclic module dependencies).*

Definition 3 (Modular Systems). *Modular systems are built inductively from constraint modules using projection, composition, union and feedback operators:*

1. A module is a modular system.
2. For modular system M *and* $\tau \subseteq \sigma_M \cup \varepsilon_M$*, modular system* $\pi_\tau(M)$ *is defined such that (a)* $\sigma_{\pi_\tau(M)} = \sigma_M \cap \tau$*, (b)* $\varepsilon_{\pi_\tau(M)} = \varepsilon_M \cap \tau$*, and (c)* $\mathcal{B} \in \pi_\tau(M)$ *iff there is a structure* $\mathcal{B}' \in M$ *with* $\mathcal{B}'|_\tau = \mathcal{B}$*.*
3. For composable modular systems M *and* M' *(no output interference) with* M *independent from* M' *(no cyclic module dependencies),* $M \rhd M'$ *is a modular system such that (a)* $\sigma_{M \rhd M'} = \sigma_M \cup (\sigma_{M'} \setminus \varepsilon_M)$*, (b)* $\varepsilon_{M \rhd M'} = \varepsilon_M \cup \varepsilon_{M'}$*, and (c)* $\mathcal{B} \in (M \rhd M')$ *iff* $\mathcal{B}|_{vocab(M)} \in M$ *and* $\mathcal{B}|_{vocab(M')} \in M'$*.*
4. For modular systems M_1 *and* M_2 *with* $\sigma_{M_1} \cap \sigma_{M_2} = \sigma_{M_1} \cap \varepsilon_{M_2} = \varepsilon_{M_1} \cap \sigma_{M_2} = \emptyset$*, the expression* $M_1 \cup M_2$ *defines a modular system such that (a)* $\sigma_{M_1 \cup M_2} = \sigma_{M_1} \cup \sigma_{M_2}$*, (b)* $\varepsilon_{M_1 \cup M_2} = \varepsilon_{M_1} \cup \varepsilon_{M_2}$*, and (c)* $\mathcal{B} \in (M_1 \cup M_2)$ *iff* $\mathcal{B}|_{vocab(M_1)} \in M_1$ *or* $\mathcal{B}|_{vocab(M_2)} \in M_2$*.*
5. For modular system M *and* $R \in \sigma_M$ *and* $S \in \varepsilon_M$ *being two symbols of similar type (i.e., either both function symbols or both predicate symbols) and of the same arities; expression* $M[R = S]$ *is a modular system such that (a)* $\sigma_{M[R=S]} = \sigma_M \setminus \{R\}$*, (b)* $\varepsilon_{M[R=S]} = \varepsilon_M \cup \{R\}$*, and (c)* $\mathcal{B} \in M[R = S]$ *iff* $\mathcal{B} \in M$ *and* $R^{\mathcal{B}} = S^{\mathcal{B}}$*.*

Further operators for combining modules can be defined as combinations of basic operators above. For instance, [14] introduced $M_1 \blacktriangleright M_2$ (composition with projection operator) as $\pi_{\sigma_{M_1} \cup \varepsilon_{M_2}}(M_1 \rhd M_2)$. Also, $M_1 \cap M_2$ is defined to be equivalent to $M_1 \rhd M_2$ (or $M_2 \rhd M_1$) when $\sigma_{M_1} \cap \varepsilon_{M_2} = \sigma_{M_2} \cap \varepsilon_{M_1} = \varepsilon_{M_1} \cap \varepsilon_{M_2} = \emptyset$. Here is an example of a modular system M combined from modules M_1, M_2, M_3, M_4 and M_5:

$$M := \pi_{E,H_1}\left[([M_1 \rhd (M_2 \cap M_3)] \rhd M_4)\,[H_1 = H_5] \rhd M_5\right].$$

One can look at M as an algebraic formula where, for example, sub-formulas $M_1 \rhd (M_2 \cap M_3)$ and M_2 both represent modules that appear in M. We call modules M_1, M_2, M_3, M_4 and M_5 *primitive in* M because they do not contain any operations.

Proposition 1. *A modular system constructed using composition, projection, feedback and union is a module.*

Proposition 2 (Law of Substitution). *Let M_1 be a modular system, M' an arbitrary (not necessarily primitive) module that appears in M_1, and let M'' be a modular system such that $M' = M''$ (equality of two sets (classes) of structures). If we replace M' in M_1 by M'', then for the resulting compound system M_2, we have $M_1 = M_2$.*

Definition 4 (Models, Solutions). *For a modular system M, a σ_M-structure $\mathcal{A}$ and a $(\sigma_M \cup \varepsilon_M)$-structure $\mathcal{B}$, we say $\mathcal{B}$* is a model of *M if $\mathcal{B} \in M$. We also say $\mathcal{B}$* is a solution to *$\mathcal{A}$* in *M if $\mathcal{B} \in M$ and $\mathcal{B}$ expands $\mathcal{A}$.*

Comparison with [14]. The framework [14] is based on a set of variables $\mathcal{X}$ each $x \in \mathcal{X}$ having a domain $D(x)$. An *assignment* over a subset of variables $X \subseteq \mathcal{X}$ is a function $\mathcal{B} : X \to \cup_{x \in X} D(x)$, which maps variables in X to values in their domains. A constraint C over a set of variables X is characterized by a set C of assignments over X, called the *satisfying assignments*. The variables of [14] are represented here by the elements of the combined $\sigma \cup \varepsilon$ vocabulary. Assignments are structures here, and we use symbols of various vocabularies instead of variables. It is essential to reformulate this notion using structures since we want to go back and forth between modules and logics in which modules are represented. In addition, we streamline most definitions. We eliminated input and output vocabularies, which can be defined, if needed, as subsets of instance and expansion vocabularies using projections. The main difference is the addition of loops, which are essential in all the results here.

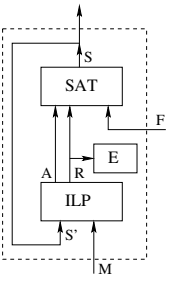


Fig. 1. Modular System Representing an SMT Solver for the Theory of Integer Linear Arithmetic

Example 2 (SMT Solvers). Consider Figure 3: It shows two boxes with solid lines which correspond to primitive MX modules and a box with dotted borders which depicts our module of interest. The vocabulary here consists of all symbols A, R, L, L', M and F where symbols A, R and L' are internal to the module, while others form the module's interface. Also, there is a line connecting L to L' which depicts a feedback.

Overall, this modular system describes a simple SMT solver for the theory of Integer Linear Arithmetic (T_{ILA}). Our two MX modules are SAT and ILP. They work on different parts of a specification. The ILP module takes a set L' of literals and a mapping M from atoms to linear arithmetic formulas. It returns two sets R and A. Semantically, R represents a set of subsets of L' so that $T_{ILA} \cup M|_r$ is *unsatisfiable* for all subsets

$r \in R$. Set A represents a set of propagated literals together with their justifications, i.e., a set of pairs (l, Q) where l is an unassigned literal (i.e., neither $l \in L'$ nor $\neg l \in L'$) and Q is a set of assigned literals asserting $l \in L'$, i.e., $Q \subseteq L'$ and $T_{ILA} \cup M|_Q \models M|_l$ (the ILA formula $M|_l$ is a logical consequence of ILA formulas $M|_Q$). The SAT module takes R and A and a propositional formula F and returns set L of literals such that: (1) L makes F true, (2) L is not a superset of any $r \in R$ and, (3) L respects all propagations (l, Q) in A, i.e., if $Q \subseteq L$ then $l \in L$. Using these modules and our operators, module SMT is defined as below to represent our simple SMT solver:

$$SMT := \pi_{\{F,M,L\}}((ILP \rhd SAT)[L = L']). \tag{1}$$

The combined module SMT is correct because, semantically, L satisfies F and all models in it should have $R = \emptyset$, i.e., $T_{ILA} \cup M|_L$ is satisfiable. This is because ILP contains structures for which if $r \in R$, then $r \subseteq L' = L$. Also, for structures in SAT, if $r \in R$ then $r \not\subseteq L$. Thus, to satisfy both these conditions, R has to be empty. Also, one can easily see that all sets L which satisfy F and make $T_{ILA} \cup M|_L$ satisfiable are solutions to this modular system (set $A = R = \emptyset$ and $L' = L$). So, there is a one-to-one correspondence between models of the modular system above and SMT's solutions to the propositional part.

Example 3 (Hamiltonian Path). In this example, we describe the Hamiltonian path problem through a combination of basic modules M_1, M_2, M_3, M_4 and M_5. We start by an informal description of each of these modules.

Module M_1 takes binary relations E and H_1 and outputs their intersection H_2. Modules M_2 and M_3 take H_2 as their input and, respectively, output binary relations H_3 and H_4 so that (1) they both are subsets of H_2, (2) all tuples in H_3 are unique with respect to the element in their first positions, and (3) all tuples in H_4 are unique with respect to the element in their second positions. Next, relations H_3 and H_4 are passed to module M_4 which outputs their intersection as the binary relation H_5. Now, H_5 is fed back into module M_1 to create a loop. Finally, M_5 takes H_5 and accepts it iff undirected transitive closure of H^5 is V^2 (V being the domain). More formally, modular system M for Hamiltonian path problem is defined as:

$$M := \pi_{E,H_1}\left[([M_1 \rhd (M_2 \cap M_3)] \rhd M_4)\,[H_1 = H_5] \rhd M_5\right].$$

Using our definitions for operations on modules, we have that $\sigma_M = \{E\}$ and $\varepsilon_M = \{H_1\}$. We claim that model expansion task for M finds a Hamiltonian path in a graph: let $\mathcal{A} = (V; E^{\mathcal{A}})$ be a graph and $\mathcal{B} = (V; E^{\mathcal{A}}, H_1^{\mathcal{B}})$ be any expansion of $\mathcal{A}$ to $\sigma_M \cup \varepsilon_M$. We know that $\mathcal{B} \in M$ iff there is expansion $\mathcal{B}'$ of $\mathcal{B}$ to $\{E, H_1, H_2, H_3, H_4, H_5\}$ such that $\mathcal{B}'|_{\{E,H_1,H_2\}} \in M_1$, $\mathcal{B}'|_{\{H_2,H_3\}} \in M_2$, $\mathcal{B}'|_{\{H_2,H_4\}} \in M_3$, $\mathcal{B}'|_{\{H_3,H_4,H_5\}} \in M_4$, $\mathcal{B}'|_{\{H_5\}} \in M_5$ and $H_1^{\mathcal{B}'} = H_5^{\mathcal{B}'}$.

Module M describes the Hamiltonian path problem because, first, any model $\mathcal{B}'$ as above has to give common interpretations to all relations H_1 to H_5. This is because $H_5 \subseteq H_3 \cap H_4 \subseteq H_2 \subseteq H_1 \subseteq H_5$. So, the common interpretation $\mathcal{R}$ to these symbols should be (1) the graph of a partial function (by definition of M_2), (2) one-to-one (by definition of M_3), and, (3) a subset of the edges (by definition of M_1). So, $\mathcal{R}$ is a collection of vertex disjoint paths and cycles in the input graph $\mathcal{A}$. Thus, as M_5 asserts

that all vertices should be reachable to each other via $\mathcal{R}$, then $\mathcal{R}$ is either a cycle or a simple path passing all vertices, i.e., either a Hamiltonian cycle or a Hamiltonian path.

4 Expressive Power

The authors of [17] emphasized the importance of *capturing NP* and other complexity classes. The capturing property, say for NP, is of fundamental importance as it shows that, for a given language:

(a) *we can express all of NP* – which gives the user an assurance of universality of the language for the given complexity class,

(b) *no more than NP can be expressed* – thus solving can be achieved by means of constructing a universal polytime reduction (called *grounding*) to an NP-complete problem such as SAT or CSP.

In the context of modular systems, we also want to investigate the expressive power of the defined language. This section defines model-theoretic properties that a module may satisfy such as totality, determinacy, polytime chability/solvability, monotonicity, anti-monotonicity. We then capture NP in a modular setting with modules satisfying some of those properties. While the focus of this result is on NP, by no means is the expressive power of the modular framework limited to NP.

Definition 5 (Extension). *For τ-structures $\mathcal{A}$ and $\mathcal{A}'$, we say $\mathcal{A}'$* extends *$\mathcal{A}$, and write $\mathcal{A} \sqsubseteq \mathcal{A}'$, if we have: (a) $dom(\mathcal{A}) = dom(\mathcal{A}')$, (b) for predicate symbol $R \in \tau$ we have $R^{\mathcal{A}} \subseteq R^{\mathcal{A}'}$.*

We sometimes abuse the notation and, for interpretations S_1 and S_2 of symbol S in two structures with the same domain, write (1) $S_1 \sqsubseteq S_2$ to say S_2 extends S_1, (2) $S_1 \sqcap S_2$ (resp. $S_1 \sqcup S_2$) to denote $S_1 \cap S_2$ (resp. $S_1 \cup S_2$) for predicate symbol S. We also use $\sqsubset$ and $\sqsupset$ to denote proper extension, i.e., similar to $\sqsubseteq$ and $\sqsupseteq$ but without equality.

Modular Systems as Operators

Definition 6 (Total Modular Systems). *For modular system M and vocabulary τ, we say M* is τ-total w.r.t. C *(C being a class of structures) if all τ-structures in C are τ-restrictions of some structure in M.*

Our definition of totality is conceptually similar to [14] but more general because, here, τ is not neccessarily a subset of σ (instance vocabulary). We might omit writing C in Definition 6 either if it is obvious from the context or if it is not important, i.e., discussion holds for all classes of structures.

Definition 7 (Deterministic Modular Systems). *For modular system M and sets of symbols τ and τ', we say M* is τ-τ'-deterministic *if for all structures $\mathcal{B}$ and $\mathcal{B}'$ in M, we have if $\mathcal{B}|_\tau = \mathcal{B}'|_\tau$ then $\mathcal{B}|_{\tau'} = \mathcal{B}'|_{\tau'}$.*

A module M that is both τ-τ'-deterministic and τ-total can be viewed as a *mapping* from τ-structures to τ'-structures, i.e., for all τ-structures $\mathcal{A}$, there is a unique τ'-structure $\mathcal{A}'$ so that for all structures $\mathcal{B} \in M$ if $\mathcal{B}|_\tau = \mathcal{A}$ then $\mathcal{B}|_{\tau'} = \mathcal{A}'$. Note

that existence and uniqueness of $\mathcal{A}'$ are guaranteed by τ-totality and τ-τ'-determinacy of M. For such M, we write $M_{\tau,\tau'}(\mathcal{A})$ to denote the unique structure $\mathcal{A}'$ that M associates to $\mathcal{A}$. We might omit τ and τ' and just write $M(\mathcal{A})$ if they are clear from the context.

Proposition 3. *For τ-τ'-deterministic modular system M:*
1. *If $\tau'' \supseteq \tau$, then M is also τ''-τ'-deterministic.*
2. *If $\tau'' \subseteq \tau'$, then M is also τ-τ''-deterministic.*

Proposition 4. *If M is both τ_1-τ_2-deterministic and τ_1'-τ_2'-deterministic, M is also $(\tau_1 \cup \tau_1')$-$(\tau_2 \cup \tau_2')$-deterministic.*

Definition 8 (Monotonicity and Anti-Monotonicity). *For modular system M and sets of symbols τ_1, τ_2 and τ_3, we say M* is τ_1-τ_2-τ_3-monotone *(resp. τ_1-τ_2-τ_3-anti-monotone) if for all structures $\mathcal{B}$ and $\mathcal{B}'$ in M, we have:*

$$\text{if } \mathcal{B}|_{\tau_1} \sqsubseteq \mathcal{B}'|_{\tau_1} \text{ and } \mathcal{B}|_{\tau_2} = \mathcal{B}'|_{\tau_2} \text{ then}$$
$$\mathcal{B}|_{\tau_3} \sqsubseteq \mathcal{B}'|_{\tau_3} (\text{resp. } \mathcal{B}'|_{\tau_3} \sqsubseteq \mathcal{B}|_{\tau_3}).$$

Proposition 5. *Let M be a τ_1-τ_2-τ_3-monotone or a τ_1-τ_2-τ_3-anti-monotone module. Then M is $(\tau_1 \cup \tau_2)$-τ_3-deterministic.*

Proof. We prove this for the monotone case. The other case is similar. Let $\mathcal{B}, \mathcal{B}' \in M$ be such that $\mathcal{B}|_{\tau_1 \cup \tau_2} = \mathcal{B}'|_{\tau_1 \cup \tau_2}$. Then, (1) $\mathcal{B}|_{\tau_2} = \mathcal{B}'|_{\tau_2}$, (2) $\mathcal{B}'|_{\tau_1} \sqsubseteq \mathcal{B}|_{\tau_1}$, and, (3) $\mathcal{B}|_{\tau_!} \sqsubseteq \mathcal{B}'|_{\tau_1}$. Thus, by (1) and (2), we know $\mathcal{B}'|_{\tau_3} \sqsubseteq \mathcal{B}|_{\tau_3}$ and, by (1) and (3), we have $\mathcal{B}|_{\tau_3} \sqsubseteq \mathcal{B}'|_{\tau_3}$. Thus, $\mathcal{B}|_{\tau_3} = \mathcal{B}'|_{\tau_3}$.

Expressive Power. We introduced several properties that a modular system may have, i.e., totality, determinacy, monotonicity and anti-monotonicity. We also proved that determinacy is a consequence of monotonicity or anti-monotonicity. Hence, it may look like the systems composed of only total (anti-)monotone modules are of very restricted computational power. However, as Theorem 1 shows, due to the presence of loops (feedbacks), the modular framework expresses all of NP although all individual modules are polytime solvable. One can extend Theorem 1 to prove that the feedback operator causes a jump from one level of the polynomial hierarchy to the next, i.e., with modules from Δ_k^P (level k of the polynomial hieachy), and in the presence of feedbacks, modular framework expresses all of Σ_{k+1}^P.

Definition 9 (Polytime Checkability, Polytime Solvability). *Let M be a module with instance vocabulary σ and expansion vocabulary ε.* M is polytime checkable *if there is a polytime program V which, given a $(\sigma \cup \varepsilon)$-structure $\mathcal{B}$, accepts $\mathcal{B}$ if and only if $\mathcal{B} \in M$. Also,* M is polytime solvable *if there is a partial function F computable in polytime such that for all structures $\mathcal{A}$: (1) $F(\mathcal{A})$ is defined if and only if there is structure $\mathcal{B} \in M$ expanding $\mathcal{A}$, and (2) if $F(\mathcal{A})$ is defined then $F(\mathcal{A}) \in M$ and $F(\mathcal{A})$ is the only structure in M which expands $\mathcal{A}$.*

Note that polytime solvability implies determinism. In theoretical computing science, a problem is a subset of $\{0,1\}^*$. However, in descriptive complexity, the equivalent definition of a problem being a set of structures is adopted. The following theorem gives a capturing result for NP:

Theorem 1 (Capturing NP over Finite Structures). *Let $\mathcal{K}$ be a problem over the class of finite structures closed under isomorphism. Then, the following are equivalent:*

1. *$\mathcal{K}$ is in NP,*
2. *$\mathcal{K}$ is the models of a modular system where all primitive modules M are σ_M-ε_M-deterministic, σ_M-total, σ_M-$vocab(\mathcal{K})$-ε_M-anti-monotone, and polytime solvable,*
3. *$\mathcal{K}$ is the models of a modular system with polytime checkable primitive modules.*

Proof. (1) $\Rightarrow$ (2): To prove this direction, we give a modular system M' which contains only one primitive module M. Primitive module M given in the proof satisfies all conditions of totality, determinacy, anti-monotonicity and polytime solvability as required by the theorem statement. Module M' feeds M's output to part of its input and projects out some auxiliary vocabulary required by M.

The proof in this direction follows the fact that, when allowing auxiliary vocabulary, ASP programs can express first order sentences (via Lloyd-Topor transformation). Thus, as FO MX captures NP over the class of finite structures, so do ASP programs (modulo the auxiliary vocabulary).

Now, consider a problem $\mathcal{K}$ in NP with vocabulary σ, i.e., an isomorphism-closed set of finite σ-structures. By the above argument, there is an ASP program P with instance vocabulary σ and expansion vocabulary ε_P which (when restricted to σ) accepts exactly those structures in $\mathcal{K}$. We now introduce a module M with $\sigma_M := \sigma \cup \varepsilon_P$ and $\varepsilon_M := \varepsilon'_P$ (where ε'_P consists of new predicate symbols R' for each predicate symbol $R \in \varepsilon_P$). Given an instance structure $\mathcal{A}$, module M works by first computing the ground program P' of P w.r.t. $dom(\mathcal{A})$. Then, M computes the reduct of P' under $\mathcal{A}$, denoted as $P'^{\mathcal{A}}$. Finally, M takes the deductive closure of $P'^{\mathcal{A}}$ and gives it as output.

Obviously, M is σ_M-ε_M-deterministic, σ_M-total and polytime computable. Also, M is σ_M-σ-ε_M-anti-monotone because, for a fixed interpretation to σ_P, an increment in ε_P makes $P'^{\mathcal{A}}$, and thus the deductive closure, smaller. Now, we define module $M' := \pi_\sigma(M[\varepsilon_P = \varepsilon'_P])$. Observe that models of M' are exactly those accepted by P.

(2) $\Rightarrow$ (3): This direction is trivial because if a modular system uses only polytime solvable primitive modules then it also uses only polytime checkable primitive modules.

(3) $\Rightarrow$ (1): Let M be a modular system whose models coincide with $\mathcal{K}$ and whose primitive modules are polytime checkable. Then, $\mathcal{K}$ is in NP because one can nondeterministically guess all the interpretations of expansion symbols of M (the set of these symbols is equal to the union of the expansion vocabularies of all M's primitive modules) and then use polytime checkability of M's primitive modules to check if this is a good guess (according to the modules, and thus according to the system itself).

Theorem 1 demonstrates the additional power that the feedback operator has brought to us. Its proof assumes that modules are described in languages with the ability to manipulate input programs and sets of atoms, and to compute fixpoints. Examples of such languages are those that capture P in the presence of ordering relation over domain elements, or the like. However, note that, in our model-theoretic view, the language that modules are described in is not important at all.

Note that Theorem 1 shows that when basic modules are restricted to polytime checkable modules, the modular system's expressive power is limited to NP. Without this restriction, the modular framework can represent Turing-complete problems. As an

example, one can encode Turing machines as finite structures and have modules that accept a finite structure if and only if it corresponds to a halting Turing machine.

Theorem 1 shows that the feedback operator causes a jump in expressive power from P to NP (or, more generally, from Δ_k^P to Σ_{k+1}^P). The proof uses a translation from ASP programs to deterministic, total, anti-monotone and polytime modules. The following running example elaborates more in this direction.

Example 4 (Stable Model Semantics). Let P be a normal logic program. We know S is a stable model for P iff $S = Dcl(P^S)$ where P^S is the reduct of P under set S of atoms (a positive program) and Dcl computes the deductive closure of a positive program, i.e., the smallest set of atoms satisfying it. Now, let $M_1(S, P, Q)$ be the module that given a set of atoms S and ASP program P computes the reduct Q of P under S. Observe that M_1 is $\{S\}$-total and $\{S\}$-$\{P\}$-$\{Q\}$-anti-monotone, and polytime solvable. Also, let $M_2(Q, S')$ be a module that, given a positive logic program Q, returns the smallest set of atoms S' satisfying Q. Again, M_2 is $\{Q\}$-total, $\{Q\}$-$\{\}$-$\{S'\}$-monotone and polytime solvable. However, $M := \pi_{\{P,S\}}((M_1 \rhd M_2)[S = S'])$ is a module which, given ground ASP program P, returns all and only the stable models of P. Therefore, the NP-complete problem of finding a stable model for a normal logic program is defined by combining total, deterministic, polytime solvable, and monotone or anti-monotone modules.

Example 4 shows that the computational power of stable models is included in the modular framework. As we will see later, this phenomenon is not accidental but is a consequence of anti-monotone loops (feedbacks). Moreover, we already know that the modular framework does not impose minimality constraint on the solution to its modules (while stable model semantics does). Thus, this framework can define sets of structures that cannot be defined in ASP.

5 Approximating Solutions

Until now, we introduced modular systems and talked about their expressive power. However, an important question associated with every modeling language is how one can find a solution to a specification in such a language. While we will address this question in a future work, here, we give some results on how to intelligently reduce the space we have to search in order to find a solution. We call this space the *candidate solution space*. To do so, we start with simple properties about extending monotonicity and anti-monotonicity to complex modules. We prove that, in the presence of loops and monotone or anti-monotone primitive modules, the combined systems satisfy many interesting properties such as existence of smallest solutions or minimality of solutions. We then develop methods for intelligently reducing the candidate solution space.

Proposition 6. *Let M be a τ_1-τ_2-τ_3-monotone (resp. anti-monotone) module. Then:*

1. *If $\tau' \subseteq \tau_1$ then M is also a τ'-$(\tau_2 \cup (\tau_1 \backslash \tau'))$-$\tau_3$-monotone (resp. anti-monotone) module.*
2. *For a set ν of symbols such that $\tau_3 \cap \nu = \emptyset$, we have M is also $(\tau_1 \cup \nu)$-τ_2-τ_3-monotone (resp. anti-monotone).*

3. *For a set ν of symbols, we have that M is also τ_1-$(\tau_2 \cup \nu)$-τ_3-monotone (resp. anti-monotone).*
4. *If $\tau' \subseteq \tau_3$ then M is also a τ_1-τ_2-τ'-monotone (resp. anti-monotone) module.*

Proposition 7. *Let M be a module that is both τ_1-τ_2-τ_3-monotone and τ_1'-τ_2'-τ_3'-monotone (resp. τ_1-τ_2-τ_3-anti-monotone and τ_1'-τ_2'-τ_3'-anti-monotone) such that $(\tau_1 \cup \tau_1') \cap (\tau_3 \cup \tau_3') = \emptyset$. Then, M is also $(\tau_1 \cup \tau_1')$-$(\tau_2 \cup \tau_2')$-$(\tau_3 \cup \tau_3')$-monotone (resp. $(\tau_1 \cap \tau_1')$-$(\tau_2 \cup \tau_2')$-$(\tau_3 \cup \tau_3')$-anti-monotone).*

Proposition 8 ((Anti-)Monotonicity Preservation). *For τ_1-τ_2-τ_3-monotone (resp. anti-monotone) modular system M and general modular system M', we have:*

1. *$M \rhd M'$ is τ_1-τ_2-τ_3-monotone (resp. anti-monotone).*
2. *$M' \rhd M$ is τ_1-τ_2-τ_3-monotone (resp. anti-monotone).*
3. *If M' is ν-τ_2-deterministic for some ν, then $M' \rhd M$ is τ_1-ν-τ_3-monotone (resp. anti-monotone).*
4. *If $\tau_1 \cup \tau_2 \subseteq \nu$ then $\Pi_\nu M$ is τ_1-τ_2-$(\nu \cap \tau_3)$-monotone (resp. anti-monotone).*
5. *$M[S_1 = S_2]$ is τ_1-τ_2-τ_3-monotone (resp. anti-monotone)*

Proposition 9 (Monotonicity under Composition). *For modular systems M and M' and vocabularies τ_1, τ_1', τ_2, τ_2', τ_3 and τ_3' such that $\tau_1' \subseteq \tau_3$:*

1. *If M is τ_1-τ_2-τ_3-monotone and M' is τ_1'-τ_2'-τ_3'-monotone, $M \rhd M'$ is τ_1-$(\tau_2 \cup \tau_2')$-τ_3'-monotone.*
2. *If M is τ_1-τ_2-τ_3-anti-monotone and M' is τ_1'-τ_2'-τ_3'-monotone, $M \rhd M'$ is τ_1-$(\tau_2 \cup \tau_2')$-τ_3'-anti-monotone.*
3. *If M is τ_1-τ_2-τ_3-monotone and M' is τ_1'-τ_2'-τ_3'-anti-monotone, $M \rhd M'$ is τ_1-$(\tau_2 \cup \tau_2')$-τ_3'-anti-monotone.*
4. *If M is τ_1-τ_2-τ_3-anti-monotone and M' is τ_1'-τ_2'-τ_3'-anti-monotone, $M \rhd M'$ is τ_1-$(\tau_2 \cup \tau_2')$-τ_3'-monotone.*

Proof. We prove the first case. The rest is similar. For $P := M \rhd M'$, let $\mathcal{B}, \mathcal{B}' \in P$ be such that $\mathcal{B}|_{\tau_2 \cup \tau_2'} = \mathcal{B}'|_{\tau_2 \cup \tau_2'}$ and $\mathcal{B}|_{\tau_1} \sqsubseteq \mathcal{B}'|_{\tau_1}$. By monotonicity of M, we have $\mathcal{B}|_{\tau_3} \sqsubseteq \mathcal{B}'|_{\tau_3}$. So, as $\tau_1' \subseteq \tau_3$, we also have $\mathcal{B}|_{\tau_1'} \sqsubseteq \mathcal{B}'|_{\tau_1'}$. Hence, by monotonicity of M', we have $\mathcal{B}|_{\tau_3'} \sqsubseteq \mathcal{B}'|_{\tau_3'}$.

These properties give us ways of deriving that a complex modular system is monotone or anti-monotone by looking at similar properties of basic constraint modules. For instance, for our two previous examples, we have:

Example 5 (Composition in Hamiltonian Path). Modules M_1, M_2, M_3 and M_4 in Example 3 are respectively $\{H_1, E\}$-$\{\}$-$\{H_2\}$-monotone, $\{H_2\}$-$\{\}$-$\{H_3\}$-monotone, $\{H_2\}$-$\{\}$-$\{H_4\}$-monotone and $\{H_3, H_4\}$-$\{\}$-$\{H_5\}$-monotone. So, by Proposition 9, $M' := M_1 \rhd (M_2 \cap M_3)$ is both $\{H_1, E\}$-$\{\}$-$\{H_3\}$-monotone and $\{H_1, E\}$-$\{\}$-$\{H_4\}$-monotone. Thus, Proposition 7 asserts M' is also $\{H_1, E\}$-$\{\}$-$\{H_3, H_4\}$-monotone. Thus, $M'' := M' \rhd M_4$ has to be $\{H_1, E\}$-$\{\}$-$\{H_5\}$-monotone (by Proposition 9).

Example 6 (Composition in ASP Programs). Modules M_1 and M_2 in Example 4 are respectively $\{S\}$-$\{P\}$-$\{Q\}$-anti-monotone and $\{Q\}$-$\{\}$-$\{S'\}$-monotone. So, by Proposition 9, $M' := M_1 \rhd M_2$ is $\{S\}$-$\{P\}$-$\{S'\}$-anti-monotone.

The rest of this section considers the important case of monotone or anti-monotone loops, i.e., monotone or anti-monotone modules under the feedback operator. Note that, although our theorems concern modules feeding their outputs back to their inputs, these modules are usually not primitive modules, but composite modules whose monotonicity or anti-monotonicity is derived by our previous propositions.

Theorem 2 (Smallest Solution). *Let M be a $(\tau \cup \{S\})$-total and $\{S\}$-τ-$\{R\}$-monotone modular system and $M' := M[S = R]$. Then, for a fixed interpretation to τ, M' has exactly one smallest solution with respect to predicate symbol R.*

Proof. Standard Tarski proof.

Theorem 2 relates smallest solutions of monotone loops in modular systems to least fixpoints of monotone operators. Therefore, many natural problems such as transitivity or connectivity are smallest solutions of some monotone modules under feedbacks. However, Theorem 2 only states that a smallest solution exists and is unique but it does not limit the models to it. The smallest solution is used to prune the candidate solution space by discarding all candidate solutions that do not extend the smallest solution.

Proposition 10 (Anti-Monotonicity and Minimality). *For $\{S\}$-τ-$\{R\}$-anti-monotone modular system M and for modular system $M' := M[S = R]$, we have that when interpretation to τ is fixed, all models of M' are minimal with respect to the interpretations of R.*

Proof. Let $\mathcal{B}_1, \mathcal{B}_2 \in M'$ be such that $B_1|_\tau = B_2|_\tau$ and $R^{\mathcal{B}_1} \sqsubseteq R^{\mathcal{B}_2}$. So, because, in M', R is fed back to S, we have $S^{\mathcal{B}_1} \sqsubseteq S^{\mathcal{B}_2}$. Hence, by $\{S\}$-τ-$\{R\}$-anti-monotonicity of M, we have that $R^{\mathcal{B}_1} \sqsupseteq R^{\mathcal{B}_2}$. Thus, $R^{\mathcal{B}_1} = R^{\mathcal{B}_2}$ and $B_1|_{\tau\cup\{R\}} = B_2|_{\tau\cup\{R\}}$, i.e., there does not exist any two structures in M' which agree on the interpretation to τ but, in one of them, interpretation of R properly extends R's interpretation in the other one.

The minimality of solutions to anti-monotone loops means that these loops may not have a smallest solution. Nevertheless, we are still able to prune the candidate solution space by finding lower and upper bounds for all the solutions to such a loop. Consider the following process for a $(\tau \cup \{S\})$-total and $\{S\}$-τ-$\{R\}$-anti-monotone modular system M where S and R are relational symbols of arity n:

$$L_0 = \emptyset, U_0 = [dom(\mathcal{A})]^n,$$
$$L_{i+1} = R^{M(\mathcal{A} \,||\, \mathcal{U}_i)}, U_{i+1} = R^{M(\mathcal{A} \,||\, \mathcal{L}_i)},$$

where $dom(\mathcal{L}_i) = dom(\mathcal{U}_i) = dom(\mathcal{A})$, $S^{\mathcal{L}_i} = L_i$, $S^{\mathcal{U}_i} = U_i$ and, for two structures $\mathcal{A}_1$ and $\mathcal{A}_2$ over the same domain but distinct vocabularies, $\mathcal{A}_1||\mathcal{A}_2$ is defined to be the structure over the same domain as $\mathcal{A}_1$ and $\mathcal{A}_2$ and with the same interpretation as them.

Theorem 3 (Bounds on Solutions to Anti-Monotone Loops). *For $(\tau \cup \{S\})$-total and $\{S\}$-τ-$\{R\}$-anti-monotone modular system M (where $S \in \sigma_M$ and $R \in \varepsilon_M$ are symbols of arity n), and for modular system $M' := M[S = R]$ and τ-structure $\mathcal{A}$, the approximation process above has a fixpoint $(L^*_{\mathcal{A}}, U^*_{\mathcal{A}})$ such that for all $\mathcal{B} \in M'$ with $\mathcal{B}|_\tau = \mathcal{A}$, we have $L^*_{\mathcal{A}} \sqsubseteq R^{\mathcal{B}}$ and $R^{\mathcal{B}} \sqsubseteq U^*_{\mathcal{A}}$.*

Proof. We prove this for relational symbols. Extending it to function symbols is straightforward. Given τ-structure $\mathcal{A}$, consider the set $S = \{\mathcal{B} \in M' \mid \mathcal{B}|_\tau = \mathcal{A}\}$. We first prove (by induction on i) that, for all i, we have: $L_i \sqsubseteq L_{i+1}$, $U_i \sqsupseteq U_{i+1}$, $L_i \sqsubseteq \bigsqcap_{\mathcal{B}\in S} R^{\mathcal{B}}$, and $U_i \sqsupseteq \bigsqcup_{\mathcal{B}\in S} R^{\mathcal{B}}$.

The base case is easy because L_0 is the empty set and U_0 contains all possible tuples. For the inductive case:

1. By induction hypothesis, $U_i \sqsupseteq U_{i+1}$. So, by anti-monotonicity of M, we have: $L_{i+1} = L^{M(\mathcal{A} \mid\mid U_i)} \sqsubseteq L^{M(\mathcal{A} \mid\mid U_{i+1})} = L_{i+2}$. Similarly, $U_{i+1} \sqsupseteq U_{i+2}$.
2. Again, by induction hypothesis, $U_i \sqsupseteq \bigsqcup_{\mathcal{B}\in S} R^{\mathcal{B}}$. So, for all structures $\mathcal{B} \in S$, we have: $U_i \sqsupseteq R^{\mathcal{B}}$. Therefore, $L_{i+1} = L^{M(\mathcal{A} \mid\mid U_i)} \sqsubseteq R^{\mathcal{B}}$. Thus, $L_{i+1} \sqsubseteq \bigsqcap_{\mathcal{B}\in S} R^{\mathcal{B}}$. Similarly, we also have $U_{i+1} \sqsupseteq \bigsqcup_{\mathcal{B}\in S} R^{\mathcal{B}}$.

So, as $\bigsqcap_{\mathcal{B}\in S} R^{\mathcal{B}} \sqsubseteq \bigsqcup_{\mathcal{B}\in S} R^{\mathcal{B}}$, we have that, for all i, $L_i \sqsubseteq U_i$. Thus, there exists ordinal α where (L_α, U_α) is the fixpoint of the sequence of pairs (L_i, U_i). Denote this pair by $(L^*_{\mathcal{A}}, U^*_{\mathcal{A}})$. Observe that, by above properties, $L^*_{\mathcal{A}} \sqsubseteq R^{\mathcal{B}}$ and $R^{\mathcal{B}} \sqsubseteq U^*_{\mathcal{A}}$ for all $\mathcal{B} \in S$ (as required).

Similar to Theorem 2, Theorem 3 also prunes the search space by limiting the candidate solutions to only those that are both supersets of the lower bound obtained by the process and subsets of the upper bound obtained by it.

Example 7 (Well-Founded Models). As discussed in Example 6, the module $M' := M_1 \rhd M_2$ is $\{S\}$-$\{P\}$-$\{S'\}$-anti-monotone. Thus, by Proposition 10, the module M defined in Example 4 can only have minimal solutions with respect to symbol S for a fixed input P. Moreover, by Proposition 3, we can find lower and upper bounds to all the solutions of module M for a fixed P. Unsurprisingly, these bounds coincide with the well-founded model of the logic program P.

6 Related Work

The work that has motivated our current paper is [14]. There, the authors define a framework in which different modelling languages such as ASP, CP and SAT can be combined on equal terms. We considerably extend their work mostly due to to the introduction of loops and results that follow. Detailed comparison is in Section 3.

An early work on adding modularity to logic programs is [7]. The authors derive a semantics for modular logic programs by viewing a logic program as a generalized quantifier. One generalization considers the concept of modules in declarative programming [18]. The authors introduce the concept of modular equivalence in normal logic programs under the stable model semantics. Their work is motivated by the fact that weak equivalence in logic programs fails to give a congruence relation and strong equivalence is not fully appropriate either. They define a weaker form of equivalence which gives rise to a congruence relation which they call modular equivalence. This work, in turn, is extended to define modularity in disjunctive programs in [13]. These two works focus on bringing the concept of modular programming into the context of logic programs and dealing with difficulties that arise there. On the other hand, our work focuses on the abstract notion of a module and what can be inferred about a modular system based on what is known about modules and how they are combined. There are several other

approaches to adding modularity to ASP languages and ID-Logic as those described in [3,1,6]. These works also put an emphasis on extending a specific language with the modularity concept. However, in our work, we are mostly concerned with mixing several knowledge representation languages. In addition, modular programming enables ASP languages to be extended by constraints or other external relations. This view is explored in [8,9,20,4,16]. While this view is advantageous in its own right, our work is different because we use a completely model-theoretic approach. Some practical modelling languages incorporate other modelling languages. For example, X-ASP [19] and ASP-PROLOG [10] extend prolog with ASP. Also ESRA [11], ESSENCE [12] and Zinc [2] are CP languages extended with features from other languages. However, these approaches give priority to the host language while our approach gives equal weight to all modelling languages that are involved. Yet another direction is the multi-context systems. In [5], the authors introduced non-monotonic bridge rules to the contextual reasoning and originated an interesting and active line of research followed by many others for solving or inconsistency explanation in non-monotonic multi-context systems. In this field, motivation comes from distributed knowledge (such as interacting agents) or partial knowledge (where trust or privacy issues are important).

7 Conclusion and Future Work

In this paper, we presented our first steps towards developing a modular approach to solving model expansion task, a task which is very common in applications, and is generally easier than satisfiability for the same logic. We described an algebra of modular systems, which includes a new operation of feedback (loop). We have shown that the loop operation adds a significant expressive power – even when all compound modules are polytime, one can can express all (and only) problems in NP. This property does not hold without the loop operation. We have also shown that the solution space of modular systems can be reduced under a natural condition on the individual modules.

In this paper, we talked about structures in general. However, in computing science, on one hand, we are interested in only the finitely representable structures and, on the other hand, most practical problems have numeric parts without any explicit bound on how big the numbers are. Therefore, for us, another interesting direction is to focus on finitely representable structures, and on structures embedded into a background structure with an infinite domain, and to understand how the framework should be modified in this setting.

Our semantics-based formalism is the first step towards developing a logic of modular systems. The logic will have a counterpart of our algebraic operations on the syntactic level. The main goal of the logic would be to address the issue of how a modular system can be "solved" – a formula would describe the system, and its models would be abstract representations of solutions to the entire system, as a function of solutions to individual modules.

Another direction is to model the task of searching for a solution to a modular system such as SMT and similar systems, while focusing on model expansion task. We plan to develop an algorithm that finds a solution through accumulating a set of constraints that a model has to satisfy. One of the interesting consequences of this work would be

to equip that algorithm with the results obtained here so that it can search the solution space more effectively. We believe that this direction may contribute to practical solvers design for declarative modelling languages.

Acknowledgement. This work is generously funded by NSERC, MITACS and D-Wave Systems. We also express our gratitude towards the anonymous referees for their useful comments.

References

1. Balduccini, M.: Modules and signature declarations for a-prolog: Progress report. In: Workshop on Software Engineering for Answer Set Programming (SEA 2007), pp. 41–55 (2007)
2. de la Banda, M., Marriott, K., Rafeh, R., Wallace, M.: The modelling language zinc. In: Benhamou, F. (ed.) CP 2006. LNCS, vol. 4204, pp. 700–705. Springer, Heidelberg (2006)
3. Baral, C., Dzifcak, J., Takahashi, H.: Macros, macro calls and use of ensembles in modular answer set programming. In: Etalle, S., Truszczyński, M. (eds.) ICLP 2006. LNCS, vol. 4079, pp. 376–390. Springer, Heidelberg (2006)
4. Baselice, S., Bonatti, P., Gelfond, M.: Towards an integration of answer set and constraint solving. In: Gabbrielli, M., Gupta, G. (eds.) ICLP 2005. LNCS, vol. 3668, pp. 52–66. Springer, Heidelberg (2005)
5. Brewka, G., Eiter, T.: Equilibria in heterogeneous nonmonotonic multi-context systems. In: Proceedings of the 22nd National Conference on Artificial Intelligence, vol. 1, pp. 385–390. AAAI Press, Menlo Park (2007)
6. Denecker, M., Ternovska, E.: A logic of non-monotone inductive definitions. Transactions on Computational Logic 9(2), 1–51 (2008)
7. Eiter, T., Gottlob, G., Veith, H.: Modular logic programming and generalized quantifiers. In: Fuhrbach, U., Dix, J., Nerode, A. (eds.) LPNMR 1997. LNCS, vol. 1265, pp. 289–308. Springer, Heidelberg (1997)
8. Eiter, T., Ianni, G., Schindlauer, R., Tompits, H.: A uniform integration of higher-order reasoning and external evaluations in answer-set programming. In: Proceedings of the 19th International Joint Conference on Artificial Intelligence, pp. 90–96. Morgan Kaufmann Publishers Inc., San Francisco (2005)
9. Elkabani, I., Pontelli, E., Son, T.C.: Smodels A – a system for computing answer sets of logic programs with aggregates. In: Baral, C., Greco, G., Leone, N., Terracina, G. (eds.) LPNMR 2005. LNCS (LNAI), vol. 3662, pp. 427–431. Springer, Heidelberg (2005)
10. Elkhatib, O., Pontelli, E., Son, T.C.: Asp – prolog: A system for reasoning about answer set programs in prolog. In: Jayaraman, B. (ed.) PADL 2004. LNCS, vol. 3057, pp. 148–162. Springer, Heidelberg (2004)
11. Flener, P., Pearson, J., Ågren, M.: Introducing ESRA, a relational language for modelling combinatorial problems. In: Bruynooghe, M. (ed.) LOPSTR 2004. LNCS, vol. 3018, pp. 214–232. Springer, Heidelberg (2004)
12. Frisch, A.M., Harvey, W., Jefferson, C., Martínez-Hernández, B., Miguel, I.: Essence: A constraint language for specifying combinatorial problems. Constraints 13, 268–306 (2008)
13. Janhunen, T., Oikarinen, E., Tompits, H., Woltran, S.: Modularity aspects of disjunctive stable models. Journal of Artificial Intelligence Research 35, 813–857 (2009)
14. Järvisalo, M., Oikarinen, E., Janhunen, T., Niemelä, I.: A module-based framework for multi-language constraint modeling. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR 2009. LNCS, vol. 5753, pp. 155–168. Springer, Heidelberg (2009)

15. Kolokolova, A., Liu, Y., Mitchell, D., Ternovska, E.: On the complexity of model expansion. In: Fermüller, C.G., Voronkov, A. (eds.) LPAR-17. LNCS, vol. 6397, pp. 447–458. Springer, Heidelberg (2010)
16. Mellarkod, V., Gelfond, M., Zhang, Y.: Integrating answer set programming and constraint logic programming. Annals of Mathematics and Artificial Intelligence 53, 251–287 (2008)
17. Mitchell, D.G., Ternovska, E.: A framework for representing and solving np search problems. In: Proceedings of the 20th National Conference on Artificial Intelligence, vol. 1, pp. 430–435. AAAI Press, Menlo Park (2005)
18. Oikarinen, E., Janhunen, T.: Modular equivalence for normal logic programs. In: Proceeding of the 2006 Conference on ECAI 2006: 17th European Conference on Artificial Intelligence, August 29 - September 1, pp. 412–416. IOS Press, Amsterdam (2006)
19. Swift, T., Warren, D.S.: The XSB System (2009), http://xsb.sourceforge.net/
20. Tari, L., Baral, C., Anwar, S.: A language for modular answer set programming: Application to ACC tournament scheduling. In: Proc. of Answer Set Programming: Advances in Theory and Implementation. CEUR-WS, pp. 277–292 (2005)

Author Index