Technical Communications of the 32nd International Conference on Logic Programming

ICLP 2016, October 16–21, 2016, New York City, USA

Edited by

Manuel Carro
Andy King
Neda Saeedloei
Marina De Vos

OASICS

Editors

Manuel Carro
Computer Science School
Technical University of Madrid
and IMDEA Software Institute
`manuel.carro@{upm.es,imdea.org}`

Andy King
Computer Science Department

University of Kent
`A.M.King@kent.ac.uk`

Neda Saeedloei
Computer Science Department
University of Minessota at Duluth
`nsaeedlo@d.umn.edu`

Marina De Vos
Computer Science Department
University of Bath
`M.D.Vos@bath.ac.uk`

ACM Classification 1998
D.1.6 Programming, D.3.1 Formal Definitions and Theory, D.3.2 Language Classifications, D.3.3 Language Constructs and Features, F.1.1 Models of Computation, F.3.2 Semantics of Programming Languages, F.4.1 Mathematical Logic, F.4.2 Grammars and Other Rewriting Systems, H.2.7 Database Administration, H.2.8 Data Mining, I.2.1 Applications and Expert Systems, I.2.11 Distributed Artificial Intelligence, I.2.2 Automatic Programming, I.2.3 Deduction and Theorem Proving, I.2.4 Knowledge Representation Formalisms and Methods, I.2.8 Graph and Tree Search Strategies

ISBN 978-3-95977-007-1

Published online and open access by

Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at http://www.dagstuhl.de/dagpub/978-3-95977-007-1.

Publication date
November, 2016

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at http://dnb.d-nb.de.

Digital Object Identifier: 10.4230/OASIcs.ICLP.2016.0

ISBN 978-3-95977-007-1 **ISSN 1868-8969** **http://www.dagstuhl.de/oasics**

OASIcs – OpenAccess Series in Informatics

OASIcs aims at a suitable publication venue to publish peer-reviewed collections of papers emerging from a scientific event. OASIcs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.

ISSN 2190-6807

http://www.dagstuhl.de/oasics

This volume is dedicated to our families and loved ones, who managed to understand us doing what they really did not understand very well.

Contents

Preface

The Thirty Second International Conference on Logic Programming (ICLP'16) took place in New York City, USA, from the 16th to the 21st October 2016. The main conference track run from the 18th to the 21st, and Doctoral Consortium took place on the 18th. This volume collects the Technical Communications corresponding to the presentations accepted to the Doctoral Consortium and the papers submitted to the main track of ICLP which the program committee judged of good quality, but not yet of the standard required to be accepted as conference full papers and published in the journal *Theory and Practice of Logic Programming* (`http://journals.cambridge.org/action/displayJournal?jid=TLP`). All the papers in this volume were presented in specific sessions of ICLP'16. In addition, the best Doctoral Consortium talk was given the opportunity to be presented in a slot of the main conference.

We solicited papers in all areas of logic programming, including:

- Theory: Semantic Foundations, Formalisms, Non-monotonic Reasoning, Knowledge Representation.
- Implementation: Compilation, Virtual Machines, Parallelism, Constraint Handling Rules, Tabling.
- Environments: Program Analysis, Transformation, Validation, Verification, Debugging, Profiling, Testing.
- Language Issues: Concurrency, Objects, Coordination, Mobility, Higher Order, Types, Modes, Assertions, Programming Techniques.
- Related Paradigms: Inductive and Co-inductive Logic Programming, Constraint Logic Programming, Answer-Set Programming, SAT-Checking.
- Applications: Databases, Big Data, Data Integration and Federation, Software Engineering, Natural Language Processing, Web and Semantic Web, Agents, Artificial Intelligence, Bioinformatics, and Education.

and, besides the papers for the Doctoral Consortium, we accepted three kinds of papers:

- Technical papers for technically sound, innovative ideas that can advance the state of logic programming;
- Application papers that impact interesting application domains;
- System and tool papers which emphasise novelty, practicality, usability, and availability of the systems and tools described.

We received 88 submissions of abstracts for the main conference, of which the Program Committee recommended 15 to be accepted as technical communications (TCs). The Doctoral Consortium, with a separate Program Committee, received 8 submissions, all of which were finally accepted.

We are of course indebted to the members of both Program Committees and external referees for their professionalism, enthusiasm, hard work, and promptness, despite the high load of the two rounds of refereeing. The Program Committee for ICLP and the DC were:

Marcello Balduccini	Michael Codish	Fabio Fioravanti
Mutsunori Banbara	Marina De Vos	Thom Frühwirth
Roman Barták	Agostino Dovier	John Gallagher
Pedro Cabalar	Gregory Duck	Marco Gavanelli
Mats Carlsson	Esra Erdem	Martin Gebser
Manuel Carro	Wolfgang Faber	Michael Hanus

Technical Communications of the 32nd International Conference on Logic Programming (ICLP 2016).
Editors: Manuel Carro, Andy King, Neda Saeedloei, and Marina De Vos
OASICS Open Access Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Katsumi Inoue	Francesco Ricca	Mirek Truszczyński
Gerda Janssens	Ricardo Rocha	Frank Valencia
Andy King	Neda Saeedloei	Alicia Villanueva
Ekaterina Komendantskaya	Takehide Soh	Jan Wielemaker
Michael Leuschel	Zoltan Somogyi	Stefan Woltran
Vladimir Lifschitz	Harald Søndergaard	Fangkai Yang
José F. Morales	Theresa Swift	Jia-Huai You
Enrico Pontelli	Francesca Toni	
Jörg Pührer	Irina Trubitsyna	

The external reviewers were:

Shqiponja Ahmetaj	Daniel Gall	Charlie Ann Page
Marco Alberti	Graeme Gange	Gilberto Pérez
Dalal Alrajeh	Michael Gelfond	Carla Piazza
Bernhard Bliem	Mayer Goldberg	Christoph Redl
Carl Friedrich Bolz	Sergio Greco	Chiaki Sakama
Davide Bresolin	Amelia Harrison	Taisuke Sato
Luciano Caroprese	Laurent Janssens	Peter Schachte
Md Solimul Chowdhury	Roland Kaminski	Nada Sharaf
Oana Cocarascu	Benjamin Kaufmann	Takehide Soh
Giuseppe Cota	Angelika Kimmig	Tran Cao Son
Kristijonas Čyras	Sebastian Krings	Nataliia Stulova
Alessandro Dal Palù	Evelina Lamma	Sophie Tourret
Ingmar Dasseville	Emily Leblanc	Guy Van den Broeck
Bart Demoen	Tingting Li	Matthias van der Hallen
Stefan Ellmauthaler	Morgan Magnin	Pedro Vasconcelos
Jorge Fandiño	Theofrastos Mantadelis	Germán Vidal
Johannes Klaus Fichte	Yunsong Meng	Yisong Wang
Andrea Formisano	Cristian Molinaro	Philipp Wanko
Michael Frank	Michael Morak	Antonius Weinzierl
Peng Fu	Eugenio Omodeo	Amira Zaki
Murdoch Gabbay	Max Ostrowski	Heng Zhang

We would also like to thank the full ICLP 2016 organisation committee, namely Michael Kifer and Neng-Fa Zhou, who acted as general chairs; Marcello Balduccini, who served as workshop chair; Peter Schüller, who acted as publicity chair; Paul Fodor, who organised the programming contest; and, finally, Joaquín Arias, who designed the web pages (and also raised the bar on ICLP logos).

Our gratitude must be extended to Torsten Schaub, who is serving in the role of President of the Association of Logic Programming, to all the members of the ALP Executive Committee and to Mirek Truszczyński, Editor-in-Chief of Theory and Practice of Logic Programming. Also, to the personnel at Schloss Dagstuhl – Leibniz Zentrum für Informatik, especially Marc Herbstritt, for their timely assistance. We would also like to thank Andrei Voronkov and his staff for the EasyChair system, which helped us coordinate submission, review, discussion, and notification.

Finally, we would like to thank our generous sponsors: LogicBlox Inc., Semantic Systems, The University of Texas at Dallas and, of course, the Association for Logic Programming.

Andy King was partially supported by EPSRC grant EP/N020243/1. Manuel Carro was partially supported by Comunidad de Madrid project S2013/ICE-2731 *N-Greens Software* and MINECO projects TIN2012-39391-C04-03 *StrongSoft* and TIN2015-67522-C3-1-R *TRACES*.

Manuel Carro Liñares, Andy King, Neda Saeedloei, Marina De Vos
Program Committee Chairs
August 2016

List of Authors

Joaquín Arias
IMDEA Software Institute
Spain
joaquin.arias@imdea.org

Marcello Balduccini
Drexel University
United States
marcello.balduccini@drexel.edu

Sotiris Batsakis
University of Huddersfield
United Kingdom
sbatsakis@gmail.com

Christopher Béatrix
LERIA – University of Angers
France
beatrix@info.univ-angers.fr

Bart Bogaerts
KU Leuven
Belgium
bart.bogaerts@cs.kuleuven.be

Jori Bomanson
Aalto University
Finland
jori.bomanson@aalto.fi

Christopher Brenton
University of Huddersfield
United Kingdom
christopher.brenton@hud.ac.uk

Tran Cao Son
New Mexico State University
United States
tson@cs.nmsu.edu

Zhuo Chen
University of Texas at Dallas
United States
zxc130130@utdallas.edu

Luís Cruz-Filipe
Dept. of Mathematics and Computer Science
University of Southern Denmark
Denmark
lcfilipe@gmail.com

Ingmar Dasseville
KUL
Belgium
ingmar.dasseville@cs.kuleuven.be

Broes De Cat
Department of Computer Science
K.U. Leuven
Belgium
broes.decat@gmail.com

Marc Denecker
KU Leuven
Belgium
marc.denecker@cs.kuleuven.be

Besik Dundua
Institute of Applied Mathematics
Tbilisi State University
Georgia
bdundua@gmail.com

Wolfgang Faber
University of Huddersfield
United Kingdom
wf@wfaber.com

Michael Frank
Ben-Gurion University of the Negev
Israel
frankm@post.bgu.ac.il

Graeme Gange
Department of Computing and Information Systems
University of Melbourne
Australia
gkgange@unimelb.edu.au

Laurent Garcia
LERIA – University of Angers
France
garcia@info.univ-angers.fr

Tiantian Gao
Stony Brook University
United States
tiagao@cs.stonybrook.edu

Martin Gebser
University of Potsdam
Germany
gebser@cs.uni-potsdam.de

Michael Gelfond
Texas Tech University
United States
michael.gelfond@ttu.edu

Michell Guzmán
LIX
Ecole Polytechnique
France
michellrad@gmail.com

Matthias van der Hallen
KU Leuven
Belgium
matthias.vanderhallen@cs.kuleuven.be

Miguel Isabel
Complutense University of Madrid
Spain
miguelis@ucm.es

Tomi Janhunen
Aalto University
Finland
tomi.janhunen@aalto.fi

Gerda Janssens
Katholieke Universiteit Leuven
Belgium
gerda.janssens@cs.kuleuven.be

Benjamin Kaufmann
University of Potsdam
Germany
kaufmann@cs.uni-potsdam.de

Roland Kaminski
University of Potsdam
Germany
kaminski@cs.uni-potsdam.de

Temur Kutsia
RISC
Johannes Kepler University Linz
Austria
kutsia@risc.jku.at

Claire Lefèvre
LERIA – University of Angers
France
claire@info.univ-angers.fr

Yuliya Lierler
University of Nebraska at Omaha
United States
ylierler@unomaha.edu

Vladimir Lifschitz
University of Texas
United States
vl@cs.utexas.edu

Patrick Lühne
University of Potsdam
Germany
patrick.luehne@cs.uni-potsdam.de

Arun Nampally
Department of Computer Science
Stony Brook University
United States
anampally@cs.stonybrook.edu

Max Ostrowski
University of Potsdam
Germany
ostrowsk@cs.uni-potsdam.de

Enrico Pontelli
New Mexico State University
United States
epontell@cs.nmsu.edu

C. R. Ramakrishnan
University at Stony Brook
United States
cram@cs.stonybrook.edu

Klaus Reisenberger-Hagmayer
Johannes Kepler University Linz
Austria
klaus.reisenberger@gmx.at

Javier Romero
University of Potsdam
Germany
javier@cs.uni-potsdam.de

Torsten Schaub
University of Potsdam
Germany
`torsten@cs.uni-potsdam.de`

Igor Stéphan
LERIA – University of Angers
France
`stephan@info.univ-angers.fr`

Peter J. Stuckey
University of Melbourne
Australia
`peter.stuckey@nicta.com.au`

Benjamin Susman
University of Nebraska at Omaha
United States
`bensusman@gmail.com`

Frank Valencia
CNRS-LIX École Polytechnique
France
`frank.valencia@lix.polytechnique.fr`

Alexander Vandenbroucke
KU Leuven
Belgium
`alexander.vandenbroucke@kuleuven.be`

Philipp Wanko
University of Potsdam
Germany
`wanko@cs.uni-potsdam.de`

SMT-Based Constraint Answer Set Solver EZSMT (System Description)

Benjamin Susman[1] **and Yuliya Lierler**[2]

1 **University of Nebraska at Omaha, Dept. of Computer Science, Omaha, USA**
`bensusman@gmail.com`

2 **University of Nebraska at Omaha, Dept. of Computer Science, Omaha, USA**
`ylierler@unomaha.edu`

Abstract

Constraint answer set programming is a promising research direction that integrates answer set programming with constraint processing. Recently, the formal link between this research area and satisfiability modulo theories (or SMT) was established. This link allows the cross-fertilization between traditionally different solving technologies. The paper presents the system EZSMT, one of the first SMT-based solvers for constraint answer set programming. It also presents the comparative analysis of the performance of EZSMT in relation to its peers including solvers EZCSP, CLINGCON, and MINGO. Experimental results demonstrate that SMT is a viable technology for constraint answer set programming.

1998 ACM Subject Classification D.1.6 [Programming Techniques] Logic Programming, D.3.2 [Programming Languages] Language Classifications – Constraint and Logic Languages, F.4.1 [Mathematical Logic and Formal Languages] Mathematical Logic – Logic and Constraint Programming

Keywords and phrases constraint answer set programming, constraint satisfaction processing, satisfiability modulo theories

Digital Object Identifier 10.4230/OASIcs.ICLP.2016.1

1 Introduction

Constraint answer set programming (CASP) is an answer set programming paradigm that integrates traditional answer set solving techniques with constraint processing. In some cases this approach yields substantial performance gains. Such systems as CLINGCON [11], EZCSP [1], and INCA [7] are fine representatives of CASP technology. Satisfiability Modulo Theories (SMT) is a related paradigm that integrates traditional satisfiability solving techniques with specialized "theory solvers" [4]. It was shown that under a certain class of SMT theories, these areas are closely related [14].

Lierler and Susman [14] presented theoretical grounds for using SMT systems for computing answer sets of a broad class of "tight" constraint answer set programs. Here we develop a system EZSMT that roots on that theoretical basis. The EZSMT solver takes as an input a constraint answer set program and translates it into so called constraint formula using the method related to forming logic program's completion. Lierler and Susman illustrated that constraint formulas coincide with formulas that SMT systems process. To interface with the SMT solvers, EZSMT utilizes the standard SMT-LIB language [4]. The empirical results carried out within this project suggest that SMT technology forms a powerful tool for finding solutions to programs expressed in CASP formalism. In particular, we compare the performance of EZSMT with such systems as EZCSP, CLINGCON, MINGO [16], and CMODELS [12]

Table 1 Example definitions for signature, lexicon, valuation.

Σ_1	$\{s, r, E, Q\}$
D_1	$\{1, 2, 3\}$
ρ_1	a function that maps $\Sigma_{1\|r}$ into relations $E^{\rho_1} = \{\langle 1\rangle\}$, $\quad Q^{\rho_1} = \{\langle 1,1\rangle, \langle 2,2\rangle, \langle 3,3\rangle\}$
$\mathcal{L}_1$	lexicon (Σ_1, D_1, ρ_1)
ν_1	valuation over $\mathcal{L}_1$, where $s^{\nu_1} = 1$ and $r^{\nu_1} = 1$
ν_2	valuation over $\mathcal{L}_1$, where $s^{\nu_2} = 2$ and $r^{\nu_2} = 1$

on six benchmarks that have been previously shown to be challenging for "pure" answer set programming approaches (exhibited by answer set solver CMODELS in experiments of this paper). The experimental analysis of this work compares and contrasts three distinct areas of automated reasoning, namely, (constraint) answer set programming, SMT, and integer mixed programming. This can be seen as one more distinct contribution of this work.

The outline of the paper follows. We review the concepts of generalized constraint satisfaction problems, logic programs, and input completion. We next introduce so called EZ constraint answer set programs and EZ constraint formulas. We subsequently present the EZSMT solver and conclude by discussing empirical results comparing EZSMT to other leading CASP systems.

2 Preliminaries

Generalized Constraint Satisfaction Problems. We now present primitive constraints as defined by Marriott and Stuckey [17] using the notation convenient for our purposes. We refer to this concept as a constraint dropping the word "primitive". We then define a generalized constraint satisfaction problem that Marriott and Stuckey refer to as "constraint".

Signature, lexicon, constraints. We adopt the following convention: for a function ν and an element x, by x^ν we denote the value that function ν maps x to (in other words, $x^\nu = \nu(x)$).

A *domain* is a *nonempty* set of elements (values). A *signature* Σ is a set of *variables*, *predicate symbols*, and *function symbols (or f-symbols)*. Predicate and function symbols are associated with a positive integer called *arity*. By $\Sigma_{|v}$, $\Sigma_{|r}$, and $\Sigma_{|f}$ we denote the subsets of Σ that contain all variables, all predicate symbols, and all f-symbols respectively. For instance, Table 1 includes the definition of sample signature Σ_1, where s and r are variables, E is a predicate symbol of arity 1, and Q is a predicate symbol of arity 2. Then, $\Sigma_{1|v} = \{s, r\}$, $\Sigma_{1|r} = \{E, Q\}$, $\Sigma_{1|f} = \emptyset$.

A *lexicon* is a tuple (Σ, D, ρ, ϕ), where (i) Σ is a signature, (ii) D is a domain, (iii) ρ is a function from $\Sigma_{|r}$ to relations on D so that an n-ary predicate symbol is mapped into an n-ary relation on D, and (iv) ϕ is a function from $\Sigma_{|f}$ to functions so that an n-ary f-symbol is mapped into a function $D^n \to D$. For a lexicon $\mathcal{L} = (\Sigma, D, \rho, \phi)$, we call any function $\nu : \Sigma_{|v} \to D$ a *valuation over* $\mathcal{L}$. Table 1 presents definitions of sample domain D_1, function ρ_1, lexicon $\mathcal{L}_1$, and valuations ν_1 and ν_2 over $\mathcal{L}_1$. A *term* over signature Σ and domain D (or a lexicon (Σ, D, ρ, ϕ)) is either (i) a variable in $\Sigma_{|v}$, (ii) a domain element in D, or (iii) an expression $f(t_1, \ldots, t_n)$, where f is an f-symbol of arity n in $\Sigma_{|f}$ and $t_1, \ldots, t_n$ are terms over $[\Sigma, D]$.

A *constraint* is defined over lexicon $\mathcal{L} = (\Sigma, D, \rho, \phi)$. *Syntactically*, a constraint is an expression of the form

$$P(t_1,\ldots,t_n),\ \text{or} \tag{1}$$
$$\neg P(t_1,\ldots,t_n), \tag{2}$$

where P is a predicate symbol from $\Sigma_{|r}$ of arity n and $t_1,\ldots,t_n$ are terms over $\mathcal{L}$. *Semantically*, we first specify recursively a value that valuation ν over lexicon (Σ, D, ρ, ϕ) assigns to a term τ over the same lexicon. We denote this value by $\tau^{\nu,\phi}$ and define it as follows:

- for a term that is a variable x in $\Sigma_{|v}$, $x^{\nu,\phi} = x^{\nu}$,
- for a term that is a domain element d in D, $d^{\nu,\phi}$ is d itself,
- for a term τ of the form $f(t_1,\ldots,t_n)$, $f(t_1,\ldots,t_n)^{\nu,\phi} = f^{\phi}(t_1^{\nu,\phi},\ldots,t_n^{\nu,\phi})$.

Second, we define what it means for valuation to be a solution of a constraint over the same lexicon. We say that ν over lexicon $\mathcal{L}$ *satisfies (is a solution to)* constraint of the form (1) over $\mathcal{L}$ when $\langle t_1^{\nu,\phi},\ldots,t_n^{\nu,\phi}\rangle \in P^{\rho}$. Valuation ν *satisfies* constraint of the form (2) when $\langle t_1^{\nu,\phi},\ldots,t_n^{\nu,\phi}\rangle \notin P^{\rho}$.

For instance, expressions

$$\neg E(s),\quad \neg E(2),\quad Q(r,s) \tag{3}$$

are constraints over lexicon $\mathcal{L}_1$. Valuation ν_1 satisfies constraints $\neg E(2)$, $Q(r,s)$, but does not satisfy $\neg E(s)$. Valuation ν_2 satisfies constraints $\neg E(s)$, $\neg E(2)$, but does not satisfy $Q(r,s)$.

A *generalized constraint satisfaction problem (GCSP)* $\mathcal{C}$ is a finite set of constraints over the same lexicon. By $\mathcal{L}^{\mathcal{C}}$ we denote the lexicon of constraints in GCSP $\mathcal{C}$. We say that a valuation ν over $\mathcal{L}^{\mathcal{C}}$ *satisfies (is a solution to)* GCSP $\mathcal{C}$, when ν is a solution to every constraint in $\mathcal{C}$. For example, any subset of constraints in (3) forms a GCSP. Sample valuation ν_2 from Table 1 satisfies GCSP composed of the first two constraints in (3). Neither ν_1 nor ν_2 satisfies the GCSP composed of all of the constraints in (3).

It is worth noting that syntactically, constraints are similar to *ground* literals of SMT. (In predicate logic, variables as defined here are referred to as *object constants* or *function symbols of 0 arity*.) Lierler and Susman [14] illustrated that given a GCSP $\mathcal{C}$ one can construct the "uniform" Σ-theory $U(\mathcal{C})$ based on the last three elements the GCSP's lexicon. Semantically, a GCSP $\mathcal{C}$ can be understood as a conjunction of its elements so that $U(\mathcal{C})$-models (as understood in SMT) of this conjunction coincide with solutions of $\mathcal{C}$.

Linear and Integer Linear Constraints. We now define "numeric" signatures and lexicons and introduce linear constraints that are commonly used in practice. The EZSMT system provides support for such constraints.

A *numeric signature* is a signature that satisfies the following requirements (i) its only predicate symbols are $<$, $>$, $\leq$, $\geq$, $=$, $\neq$ of arity 2, and (ii) its only f-symbols are $+$, $\times$ of arity 2. We use the symbol $\mathcal{A}$ to denote a numeric signature. Let $\mathbb{Z}$ and $\mathbb{R}$ be the sets of integers and real numbers respectively. Let $\rho_{\mathbb{Z}}$ and $\rho_{\mathbb{R}}$ be functions that map predicate symbols $\{<,>,\leq,\geq,=,\neq\}$ into usual arithmetic relations over integers and over reals, respectively. Let $\phi_{\mathbb{Z}}$ and $\phi_{\mathbb{R}}$ be functions that map f-symbols $\{+,\times\}$ into usual arithmetic functions over integers and over reals, respectively. We can now define the following lexicons (i) an *integer lexicon* of the form $(\mathcal{A},\mathbb{Z},\rho_{\mathbb{Z}},\phi_{\mathbb{Z}})$, and a *numeric lexicon* of the form $(\mathcal{A},\mathbb{R},\rho_{\mathbb{R}},\phi_{\mathbb{R}})$.

A *(numeric) linear expression* has the form

$$a_1x_1 + \cdots + a_nx_n, \tag{4}$$

where $a_1, \ldots, a_n$ are real numbers and $x_1, \ldots, x_n$ are variables over real numbers. When $a_i = 1$ $(1 \leq i \leq n)$ we may omit it from the expression. We view expression (4) as an abbreviation for the following term $+(\times(a_1, x_1), +(\times(a_2, x_2), \cdots + (\times(a_{n-1}, x_{n-1}), \times(a_n, x_n)) \ldots)$ over some numeric lexicon $(\mathcal{A}, \mathbb{R}, \rho_{\mathbb{R}}, \phi_{\mathbb{R}})$, where $\mathcal{A}$ contains $x_1, \ldots, x_n$ as its variables. For instance, $2x + 3y$ is an abbreviation for the expression $+(\times(2, x), \times(3, y))$. An *integer linear expression* has the form (4), where $a_1, \ldots, a_n$ are integers, and $x_1, \ldots, x_n$ are variables over integers.

We call a constraint *linear (integer linear)* when it is defined over some numeric (integer) lexicon and has the form $\bowtie (e, k)$ where e is a linear (integer linear) expression, k is a real number (an integer), and $\bowtie$ belongs to $\{<, >, \leq, \geq, =, \neq\}$. We can denote $\bowtie (e, k)$ using usual infix notation as follows $e \bowtie k$.

We call a GCSP a *(integer) linear constraint satisfaction problem* when it is composed of (integer) linear constraints. For instance, consider integer linear constraint satisfaction problem composed of two constraints $x > 4$ and $x < 5$ (here signature $\mathcal{A}$ is implicitly defined by restricting its variable to contain x). When the lexicon of GCSP is clear from the context, as in this example, we omit an explicit reference to it. It is easy to see that this problem has no solutions. On the other hand, linear constraint satisfaction problem composed of the same two constraints has infinite number of solutions, including valuation that assigns x to 4.1.

Logic Programs and Input Completion. A *vocabulary* is a set of propositional symbols also called atoms. As customary, a *literal* is an atom a or its negation $\neg a$. A *(propositional) logic program* over vocabulary σ is a set of *rules* of the form

$$a \leftarrow b_1, \ldots, b_\ell,\ \mathit{not}\ b_{\ell+1}, \ldots,\ \mathit{not}\ b_m,\ \ \mathit{not}\ \ \mathit{not}\ b_{m+1}, \ldots,\ \mathit{not}\ \ \mathit{not}\ b_n \tag{5}$$

where a is an atom over σ or $\bot$, and each b_i, $1 \leq i \leq n$, is an atom in σ. We will sometimes use the abbreviated form for rule (5), i.e., $a \leftarrow B$, where B stands for the right hand side of the arrow in (5) and is also called a *body.* We sometimes identify body B with the propositional formula

$$b_1 \wedge \ldots \wedge b_\ell \wedge \neg b_{\ell+1} \wedge \ldots \wedge \neg b_m \wedge \neg\neg b_{m+1} \wedge \ldots \wedge \neg\neg b_n. \tag{6}$$

and rule (5) with the propositional formula $B \rightarrow a$. We call expressions $b_1 \wedge \ldots \wedge b_\ell$ and $\neg b_{\ell+1} \wedge \ldots \wedge \neg b_m$ in (6) *strictly positive* and *strictly negative* respectively. The expression a is the *head* of the rule. When a is $\bot$, we often omit it. We call such a rule *a denial.* We write $hd(\Pi)$ for the set of nonempty heads of rules in Π. We refer the reader to the paper [15] for details on the definition of an *answer set.*

We call a rule whose body is empty a *fact.* In such cases, we drop the arrow. We sometimes may identify a set X of atoms with a set of facts $\{a. \mid a \in X\}$. Also, it is customary for a given vocabulary σ, to identify a set X of atoms over σ with (i) a complete and consistent set of literals over σ constructed as $X \cup \{\neg a \mid a \in \sigma \setminus X\}$, and respectively with (ii) an assignment function or interpretation that assigns truth value *true* to every atom in X and *false* to every atom in $\sigma \setminus X$.

We now restate the definitions of input answer set and input completion [14]. These concepts are instrumental in defining constraint answer set programs and building a link towards using SMT solvers for solving such programs. In particular, it is well known that for the large class of logic programs, referred to as "tight" programs, its answer sets coincide with models of its completion, as shown by Fages [8]. A similar relation holds between input answer sets of a program and models of input completion.

▶ **Definition 1.** For a logic program Π over vocabulary σ, a set X of atoms over σ is an *input answer set* of Π relative to vocabulary $\iota \subseteq \sigma$, when X is an answer set of the program $\Pi \cup ((X \cap \iota) \setminus hd(\Pi))$.

▶ **Definition 2.** For a program Π over vocabulary σ, the *input-completion* of Π relative to vocabulary $\iota \subseteq \sigma$, denoted by $IComp(\Pi, \iota)$, is defined as the set of propositional formulas that consists of the implications $B \to a$ for all rules (5) in Π and the implications $a \to \bigvee_{a \leftarrow B \in \Pi} B$ for all atoms a occurring in $(\sigma \setminus \iota) \cup hd(\Pi)$.

Tightness is a syntactic condition on a program that can be verified by means of its dependency graph. The *dependency graph* of Π is the directed graph G such that (i) the vertices of G are the atoms occurring in Π, and (ii) for every rule $a \leftarrow B$ in Π whose head is not $\bot$, G has an edge from a to each atom in strictly positive part of B. A program is called *tight* if its dependency graph is acyclic.

▶ **Theorem 3.** *For a tight program Π over vocabulary σ and vocabulary $\iota \subseteq \sigma$, a set X of atoms from σ is an input answer set of Π relative to ι if and only if X satisfies program's input-completion $IComp(\Pi, \iota)$.*

3 EZ Constraint Answer Set Programs and Constraint Formulas

We now introduce EZ programs accepted by the CASP solver EZCSP. The EZSMT system accepts subclass of these programs (as it poses additional tightness restrictions).

Let σ_r and σ_i be two disjoint vocabularies. We refer to their elements as *regular* and *irregular* atoms respectively. For a program Π and a propositional formula F, by $At(\Pi)$ and $At(F)$ we denote the set of atoms occurring in Π and F, respectively.

▶ **Definition 4.** An *EZ constraint answer set program (or EZ program)* over vocabulary $\sigma_r \cup \sigma_i$ is a triple $\langle \Pi, \mathcal{B}, \gamma \rangle$, where

- Π is a logic program over the vocabulary $\sigma_r \cup \sigma_i$ such that
  - atoms from σ_i only appear in strictly negative part of the body[1] and
  - any rule that contains atoms from σ_i is a denial,
- $\mathcal{B}$ is a set of constraints over the same lexicon, and
- γ is an injective function from the set σ_i of irregular atoms to the set $\mathcal{B}$ of constraints.

For an EZ program $P = \langle \Pi, \mathcal{B}, \gamma \rangle$ over $\sigma_r \cup \sigma_i$, a set $X \subseteq At(\Pi)$ is an *answer set* of P if

- X is an input answer set of Π relative to σ_i, and
- the GCSP $\{\gamma(a) | a \in X \cap \sigma_i\}$ has a solution.

This form of a definition of EZ programs is inspired by the definition of constraint answer set programs presented in [14] that generalize the CLINGCON language by Gebser et al. [11]. There are two major differences between EZCSP and CLINGCON languages. First, the later allows irregular atoms to appear in non-denials (though such atoms cannot occur in heads). Second, the third condition on answer sets of CLINGCON programs states that the GCSP $\{\gamma(a) | a \in X \cap \sigma_i\} \cup \{\neg\gamma(a) | a \in (At(\Pi) \cap \sigma_i) \setminus X\}$ has a solution.

Ferraris and Lifschitz [9] showed that a choice rule $\{a\} \leftarrow B$[2] can be seen as an abbreviation for a rule $a \leftarrow$ *not not* a, B. We adopt this abbreviation.

[1] The fact that atoms from σ_i only appear in strictly negative part of the body rather than in any part of the body is inessential for the kind of constraints the EZCSP system allows.

[2] Choice rules were introduced in [19] and are commonly used in answer set programming.

Π_1	Reading of a rule
$\{switch\}.$	Action *switch* is exogenous
$lightOn \leftarrow\ switch, not\ am.$	Light is on (*lightOn*) during the night (*not am*) when *switch* has occurred.
$\leftarrow not\ lightOn.$	The light must be on.
$\{am\}.$	It is night (*not am*) or morning (*am*)
$\leftarrow not\ am,\ not\ \lvert x \geq 12 \rvert.$	It must be *am* when it is not the case that $x \geq 12$ (x is understood as the hours).
$\leftarrow am,\ not\ \lvert x < 12 \rvert.$	It must be *am* when it is $x < 12$.
$\leftarrow not\ \lvert x \geq 0 \rvert.$	Variable x must be nonnegative.
$\leftarrow not\ \lvert x \leq 23 \rvert.$	Variable x must be less than or equal to 23.

Figure 1 Program Π_1 and annotations of its rules.

▶ **Example 5.** Let us consider sample EZ program. Let Σ_2 be the numeric signature containing a single variable x. By $\mathcal{L}_2$ we denote the integer lexicon $([\Sigma_2, \mathbb{Z}], \rho_{\mathbb{Z}})$. We are now ready to define an EZ program $P_1 = \langle \Pi_1, \mathcal{B}_{\mathcal{L}_2}, \nu_1 \rangle$ over lexicon $\mathcal{L}_2$, where

- Π_1 is the program presented in Figure 1. The set of irregular atoms of Π_1 is $\{|x \geq 12|, |x < 12|, |x \geq 0|, |x \leq 23|\}$. (We use vertical bars in our examples to mark irregular atoms.) The remaining atoms form the regular set.
- $\mathcal{B}_{\mathcal{L}_2}$ is the set of all integer linear constraints over $\mathcal{L}_2$, which obviously includes constraints $\{x \geq 12, x < 12, x \geq 0, x \leq 23\}$, and
- function γ_1 is defined in intuitive manner so that for instance irregular atom $|x \geq 12|$ is mapped to integer linear constraint $x \geq 12$.

Consider the set

$$\{switch,\ lightOn, |x \geq 12|, |x \geq 0|, |x \leq 23|\} \tag{7}$$

over atoms $At(\Pi_1)$. This set is the only input answer set of Π_1 relative to its irregular atoms. Also, the integer linear constraint satisfaction problem with constraints

$$\{\gamma_1(|x \geq 12|), \gamma_1(|x \geq 0|), \gamma_1(|x \leq 23|)\} = \{x \geq 12, x \geq 0, x \leq 23\} \tag{8}$$

has a solution. There are 12 valuations $\nu_1 \ldots \nu_{12}$ over $\mathcal{L}_2$, which satisfy this GCSP: $x^{\nu_1} = 12$, $\ldots$, $x^{\nu_{12}} = 23$. It follows that set (7) is an answer set of P_1.

Just as we defined EZ constraints answer set programs, we can define EZ constraint formulas.

▶ **Definition 6.** An *EZ constraint formula* over the vocabulary $\sigma_r \cup \sigma_i$ is a triple $\langle F, \mathcal{B}, \gamma \rangle$, where

- F is a propositional formula over the vocabulary $\sigma_r \cup \sigma_i$,
- $\mathcal{B}$ is a set of constraints over the same lexicon, and
- γ is an injective function from the set σ_i of irregular atoms to the set $\mathcal{B}$ of constraints.

For a constraint formula $\mathcal{F} = \langle F, \mathcal{B}, \gamma \rangle$ over $\sigma_r \cup \sigma_i$, a set $X \subseteq At(F)$ is a *model* of $\mathcal{F}$ if

- X is a model of F, and
- the GCSP $\{\gamma(a) | a \in X \cap \sigma_i\}$ has a solution.

Following theorem captures a relation between EZ programs and EZ constraint formulas. This theorem is an immediate consequence of Theorem 3.

▶ **Theorem 7.** *For an EZ program $P = \langle \Pi, \mathcal{B}, \gamma \rangle$ over the vocabulary $\sigma = \sigma_r \cup \sigma_i$ and a set X of atoms over σ, when Π is tight, X is an answer set of P if and only if X is a model of EZ constraint formula $\langle IComp(\Pi, \sigma_i), \mathcal{B}, \gamma \rangle$ over σ.*

In the sequel, we will abuse the term "tight". We will refer to an EZ program $P = \langle \Pi, \mathcal{B}, \gamma \rangle$ as *tight* when its first member Π has this property.

Linear and Integer Linear EZ Programs. We now review the more refined details behind programs supported by EZCSP. These EZ programs are of particular form:

1. $\langle \Pi, \mathcal{B}_{\mathcal{L}}, \gamma \rangle$, where $\mathcal{L}$ is a numeric lexicon and $\mathcal{B}_{\mathcal{L}}$ is the set of all linear constraints over $\mathcal{L}$, or
2. $\langle \Pi, \mathcal{B}_{\mathcal{L}}, \gamma \rangle$, where $\mathcal{L}$ is an integer lexicon and $\mathcal{B}_{\mathcal{L}}$ is the set of all integer linear constraints over $\mathcal{L}$.

We refer to the former as *EZ programs modulo linear constraints (or EZ(L) programs)*, whereas to the latter as *EZ programs modulo integer linear constraints (or EZ(IL) programs)*. Similarly, we can define EZ constraint formulas modulo linear constraints and EZ constraint formulas modulo integer linear constraints. Lierler and Susman [14] showed that such constraint formulas coincide with formulas in satisfiability modulo linear arithmetic, or SMT(L), and satisfiability modulo integer linear arithmetic, or SMT(IL), respectively.

The EZ program P_1 from Example 5 is an EZ(IL) program. Listing 1 presents this program in the syntax accepted by the EZCSP solver. We refer to this syntax as the EZCSP language. Line 1 in Listing 1 specifies that this is an EZ(IL) program. Line 2, first, declares that variable x is in the signature of program's integer lexicon. Second, it specifies that x may be assigned values in range from 0 to 23. Thus, Line 2 essentially encodes the last two rules in Π_1 presented in Figure 1. Lines 3-6 follow the first four lines of Π_1 modulo replacement of symbol $\leftarrow$ with symbols `:-`. In the EZCSP language, all irregular atoms are enclosed in a "required" statement and are syntactically placed in the head of their rules. So that Lines 7 and 8 encode the last two rules of Π_1, respectively. If a denial of an EZ program contains more than one irregular atom then in the EZCSP language disjunction in required statement is used to encode such rules. For instance, an EZ rule

$$\leftarrow \ \ not\ |x > 5|,\ not\ |x < 12|$$

has the form $required(x > 5 \vee x < 12)$. in the EZCSP syntax. (One may also use conjunction and implication within the required syntax.)

```
1 cspdomain(fd).
2 cspvar(x,0,23).
3 {switch}.
4 lightOn :- switch not am.
5 :- not lightOn.
6 {am}.
7 required(x ≥ 12) :- not am.
8 required(x < 12) :- am.
```

Listing 1 EZCSP Program.

4 The EZSMT Solver

By Theorem 7, it follows that answer sets of a tight EZ program coincide with models of a constraint formula that corresponds to the input completion of the EZ program relative to its irregular atoms. Thus, tight EZ(L) and EZ(IL) programs can be converted to "equivalent" SMT(L) and SMT(IL) formulas, respectively. This fact paves a way to utilizing SMT technology for solving tight EZ programs. The EZSMT system introduced in this work roots on these ideas.

In a nutshell, the EZSMT system takes a tight EZ(L) or EZ(IL) program written in the EZCSP language and produces an equivalent SMT(L) or SMT(IL) formula written in the SMT-LIB language that is a common input language for SMT solvers [4]. Subsequently,

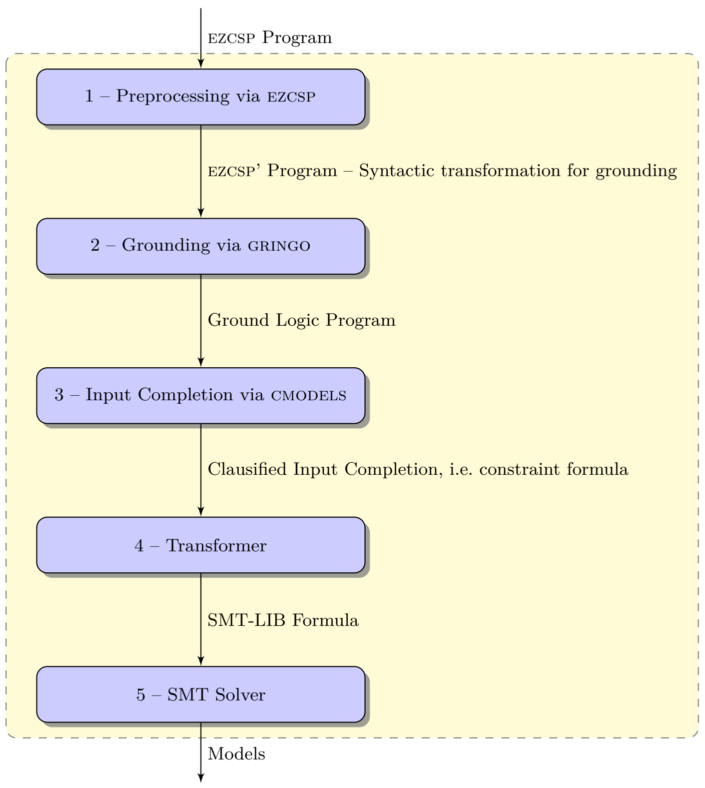

EZCSP Program
1 – Preprocessing via EZCSP
EZCSP' Program – Syntactic transformation for grounding
2 – Grounding via GRINGO
Ground Logic Program
3 – Input Completion via CMODELS
Clausified Input Completion, i.e. constraint formula
4 – Transformer
SMT-LIB Formula
5 – SMT Solver
Models

Figure 2 EZSMT Pipeline.

EZSMT runs a compatible SMT solver, such as CVC4 [3] or Z3 [6], to compute models of the program.

Few remarks are due with respect to the SMT-LIB language. This language allows the SMT research community to develop benchmarks and run solving competitions using standard interface of common input language. Barret et al. [4] define the syntax and usage of SMT-LIB. As opposed to constraint answer set programming languages, which are regarded as declarative *programming* languages, SMT-LIB is a low-level specification language. It is not intended to be a modeling language, but geared to be easily interpretable by SMT solvers and serve as a standard interface to these systems. As such, this work provides an alternative to SMT-LIB for utilizing SMT technology. It advocates the use of tight EZ programs as a declarative programming interface for SMT solvers. Also the availability of SMT-LIB immediately enables its users to interface multiple SMT-solvers as off-the-shelve tools without the need to utilize their specialized APIs.

The EZSMT Architecture. We now present details behind the EZSMT system. Figure 2 illustrates its pipeline. We use the EZ program from Example 5 to present a sample workflow of EZSMT.

Preprocessing and Grounding. In this paper, we formally introduced EZ programs over a signature that allows propositional atoms or irregular atoms. In practice, EZCSP language, just as traditional answer set programming languages, allows the users to utilize non-irregular atoms with schematic-variables. The process of eliminating these variables is referred to as *grounding* [10]. It is a well understood process in answer set programming and off the shelf grounders exist, e.g., the GRINGO system[3] [10]. The EZSMT solver also allows schematic-variables (as they are part of the EZCSP language) and relies on GRINGO to eliminate these variables.

Prior to applying GRINGO, all irregular atoms in the input program must be identified to be properly processed while grounding. The "required" keyword in the EZCSP language allows us to achieve this so that the rules with the "required" expression in the head are converted into an intermediate language. The invocation of the EZCSP system with the `--preparse-only` flag performs the conversion. The preprocessing performed by EZCSP results in a valid input program for the grounder GRINGO.

For instance, the application of EZCSP with `--preparse-only` flag on the program in Listing 1 results in the program that replaces last two rules of original program by the following rules

```
required(ezcsp__geq(x, 12)) :- not am.
required(ezcsp__lt(x, 12)) :- am.
```

Program's Completion. The third block in the pipeline in Figure 2 is responsible for three tasks. First, it determines whether the program is tight or not. Given a non tight program the system will exit with the respective message. Second, it computes the input completion of a given program (recall, that this input completion can be seen as an SMT program). Third, the input completion is clausified using Tseitin transformations so that the resulting formula is in conjunctive normal form. This transformation preserves the models of the completion modulo original vocabulary. The output from this step is a file in a DIMACS[4]-inspired format. System CMODELS [12] is used to perform the described steps. It is invoked with the `--cdimacs` flag.

For example, given the grounding produced by GRINGO for the preprocessed program in Listing 1, CMODELS will produce the output presented in Listing 2. This output encodes the clausified input completion of the EZ program in Example 5 and can be viewed as an SMT formula.

```
smt cnf 5 8
-switch switch 0
-switch lightOn 0
-lightOn switch 0
cspdomain(fd) 0
cspvar(x,0,23) 0
switch 0
lightOn 0
required(ezcsp__geq(x,12)) 0
```

Listing 2 Completion of EZCSP Program.

The first line in Listing 2 states that there are 5 atoms and 8 clauses in the formula. Each other line stands for a clause, for instance, line `-switch switch 0` represents clause $\neg switch \vee switch$.

It is important to note that just as the EZCSP language accepts programs with schematic variables, it also accepts programs with so called weight and cardinality constraint rules introduced in [19]. System CMODELS eliminates such rules in favor of rules of the form (5)

[3] `http://potassco.sourceforge.net`
[4] `http://www.satcompetition.org/2009/format-benchmarks2009.html`

discussed here. (The translation used by CMODELS was introduced in [9].) Thus, solver EZSMT is capable of accepting programs that contain weight and cardinality constraint rules.

Transformation. The output program from CMODELS serves as input to the Transformer block in the EZSMT pipeline. Transformer converts the SMT formula computed by CMODELS into the SMT-LIB syntax. For instance, given the SMT program presented in Listing 2, the Transformer produces the following SMT-LIB code.

```
1
2    (set-option :interactive-mode true)
3    (set-option :produce-models true)
4    (set-option :produce-assignments true)
5    (set-option :print-success false)
6    (check-sat)
7    (get-model)
8    (set-logic QF_LIA)
9    (declare-fun |lightOn| () Bool)
10   (declare-fun |required(ezcsp__geq(x,12))| () Bool)
11   (declare-fun |switch| () Bool)
12   (declare-fun |cspvar(x,0,23)| () Bool)
13   (assert (or (not |switch|) |switch|))
14   (assert (or (not |switch|) |lightOn|))
15   (assert (or (not |lightOn|) |switch|))
16   (assert |cspvar(x,0,23)|)
17   (assert |switch|)
18   (assert |lightOn|)
19   (assert |required(ezcsp__geq(x,12))|)
20   (declare-fun |x| () Int)
21   (assert (=> |required(ezcsp__geq(x,12))| (>= |x|  12)))
22   (assert (=> |cspvar(x,0,23)| (<= 0 |x|)))
23   (assert (=> |cspvar(x,0,23)| (>= 23 |x|)))
```

The resultant SMT-LIB specification can be described as follows:

(i) Lines 1–6 are responsible for setting directives necessary to indicate to an SMT solver that it should find a model of the program after satisfiability is determined [4].

(ii) In line 7, the Transformer instructs an SMT solver to use quantifier-free linear integer arithmetic (*QF_LIA*) [4] to solve given SMT(IL) formula. (The clause `cspdomain(fd) 0` from Listing 2 serves as an indicator that the given formula is an SMT(IL) formula.)

(iii) Lines 8–11 are declarations of the atoms in our sample program as boolean variables (called functions in the SMT-LIB parlance).

(iv) Lines 12–18 assert the clauses from Listing 2 to be true.

(v) Line 19 declares variable x to be an integer.

(vi) Line 20 expresses the fact that if the irregular atom `required(ezcsp__geq(x,12))` holds then the constraint $x \geq 12$ must also hold. In other words, it plays a role of a mapping γ_1 from Example 5.

(vii) Lines 21–22 declare the domain of variable x to be in range from 0 to 23 (recall how Listing 1 encodes this information with `cspvar(x,0,23)`).

SMT Solver. The final step is to use an SMT solver that accepts input in SMT-LIB. The output produced by CVC4[5] given the SMT-LIB program listed last follows:

[5] We note that the output format of the SMT solver Z3 is of the same style as that of CVC4.

```
sat
(model
(define-fun lightOn () Bool true)
(define-fun |required(ezcsp__geq(x,12))| () Bool true)
(define-fun switch () Bool true)
(define-fun |cspvar(x,0,23)| () Bool true)
(define-fun x () Int 12))
```

The first line of the output indicates that a satisfying assignment exists. The subsequent lines present a model that satisfies the given SMT-LIB program. Note how this model corresponds to answer set (7). Also, the solver identified one of the possible valuations for x that satisfies integer linear constraint satisfaction problem (8), this valuation maps x to 12.

Limitations. Due to the fact that the EZSMT solver accepts programs in the EZCSP language, it is natural to compare the system to the EZCSP solver. The EZSMT system faces some limitations relative to EZCSP. The EZSMT solver accepts only a subset of the EZCSP language. In particular, it supports a limited set of its global constraints [2]. Only, the global constraints *all_different* and *sum* are supported by EZSMT. Also, EZSMT can only be used on tight EZCSP programs. Yet, we note that this is a large class of programs. No support for minimize and maximize statements of EZCSP or GRINGO languages is present. In addition, solver EZSMT computes only a single answer set. Modern SMT solvers are often used for establishing satisfiability of a given formula rather than for finding its models. For instance, the SMT-LIB language does not provide a directive to instruct an SMT solver to find all models for its input. To bypass this obstacle one has to promote (i) the extensions of the SMT-LIB standard to allow a directive for computing multiple models as well as (ii) the support of this functionality by SMT solvers. Alternatively, one may abandon the use of SMT-LIB and utilize the specialized APIs of SMT solvers in interfacing these systems. The later solution seems to lack the generality as it immediately binds one to peculiarities of APIs of distinct software systems. Addressing mentioned limitations of EZSMT is a direction of future work.

5 Experimental Results

In order to demonstrate the efficacy of the EZSMT system and to provide a comparison to other existing CASP solvers, six problems have been used to benchmark EZSMT. The first three benchmarks stem from the Third Answer Set Programming Competition, 2011[6] (ASPCOMP). The selected encodings are: *weighted sequence, incremental scheduling,* and *reverse folding.* Balduccini and Lierler [2] use these three problems to assess performance of various configurations of the EZCSP and CLINGCON systems. We utilize the encodings for EZCSP and CLINGCON stemming from this earlier work for these problems. We also adopted these encodings to fit the syntax of the MINGO language to experiment with this system. The last three benchmarks originate from the assessment of solver MINGO [16]. This system translates CASP programs into mixed integer programming formalism and utilizes IBM ILOG CPLEX[7] system to find solutions. The selected problems are: *job shop, newspaper,* and *sorting.* We used the encodings provided in [16] for MINGO, CLINGCON, and CMODELS. We adopted the CLINGCON encoding to fit the syntax of the EZCSP language to experiment with EZCSP and EZSMT. All six mentioned benchmarks do not scale when using traditional answer

[6] `https://www.mat.unical.it/aspcomp2011`
[7] `http://www.ibm.com/software/commerce/optimization/cplex-optimizer/`

Table 2 ASPCOMP 2011 and MINGO Benchmarks

Benchmark (number of instances)	EZSMT CVC4	EZSMT Z3	CLINGCON	EZCSP	MINGO	CMODELS
	Cumulative Time (timeout)					
Reverse folding (50)	47948 (22)	4873 (2)	2014 (1)	**559**	14962 (1)	84616 (47)
Weighted Seq. (30)	24.2	**23.3**	187	13879	1330	54000 (30)
Incr. scheduling (30)	10277 (5)	**9135 (5)**	20417 (11)	37332 (20)	13626 (7)	54000 (30)
Job shop (100)	106	48.8	**2.77**	180000 (100)	1137	163106 (90)
Newspaper (100)	7.68	3.77	**0.02**	3.53	54.2	111615 (53)
Sorting (189)	646	233	**31.7**	103	8282	271004 (141)

set solvers. The EZSMT system, encodings, and instances used for benchmarking are available at the EZSMT site: `http://unomaha.edu/nlpkr/software/ezsmt/`.

All experiments were conducted on a computer with an Intel Core i7-940 processor running Ubuntu 14.04 LTS (64-bit) operating system. Each benchmark was allocated 4 GB RAM, a single processor core, and given an 1,800 second timeout. No benchmarks were run simultaneously.

Five CASP solvers and one answer set (ASP) solver were benchmarked:

- EZSMT v. 1.0 with CVC4 v. 1.4 as the SMT solver (EZSMT– CVC4),
- EZSMT v. 1.0 with Z3 v. 4.4.2 – 64 bit as the SMT solver (EZSMT– Z3),
- CLINGCON v. 2.0.3 with constraint solver GECODE v. 3.7.3 and ASP solver CLASP v. 1.3.10,
- EZCSP v. 1.6.20 with constraint solver B-Prolog v. 7.4 #3 and ASP solver CMODELS v. 3.86,
- MINGO v. 2012-09-30 with mixed integer solver CPLEX v. 12.5.1.0, and
- ASP solver CMODELS v. 3.86 [12].

All of these systems invoke grounder GRINGO versions 3.0.+ during their executions. Time spent in grounding is reported as part of the solving time. The best performing EZCSP configuration, as reported in [2], was used for each run of the ASPCOMP benchmarks. All other systems were run under their default configurations. We note that for systems EZSMT-CVC4, EZSMT-Z3, and EZCSP identical encodings across the benchmarks were used. The formalizations for other solvers can be seen as syntactically different versions of these encodings.

At a high-level abstraction, one may summarize the architectures of the CLINGCON and EZCSP solvers as *ASP-based solvers plus constraint solver*. Given a constraint answer set program $\langle \Pi, \mathcal{B}, \gamma \rangle$, both CLINGCON and EZCSP first use an answer set solver to (partially) compute an input answer set of Π. Second, they contact a constraint solver to verify whether respective GCSP has a solution. As mentioned earlier, MINGO's solving is based on mixed integer programming.

Table 2 presents the experimental results. Each name of a benchmark is annotated with the number of instances used in the experiments. The collected running times are reported in cumulative fashion. The number in parenthesis annotates the number of timeouts or memory outs (that we do not distinguish). Any instance which timed-out/memory-out is represented in cumulative time by adding the maximum allowed time for an instance (1,800 seconds). For instance, answer set solver CMODELS timed out on all 30 instances of the weighted sequence benchmark so that the cumulative time of 54,000 is reported. Bold font is used to mark the best performing solver.

In the *reverse folding* benchmark, the difference between SMT solvers used for EZSMT becomes very apparent. In this case, the Z3 solver performed better than CVC4 by an order of magnitude. This underlines both the importance of solver selection and difference between SMT solvers. These observations mark the significance of the flexibility that EZSMT provides to its users as they are free to select different SMT solvers as appropriate to the instances and encodings. Indeed, SMT solvers are interfaced via the standard SMT-LIB language by EZSMT.

In the *weighted sequence* benchmark, we note that no CASP system timed out. In this case, the EZSMT system features a considerable speedup. It noticeably outperforms CLINGCON and EZCSP by multiple orders of magnitude.

In *incremental scheduling*, the original EZCSP encoding includes a global constraint, cumulative, which is not supported by EZSMT. To benchmark EZSMT on this problem, the encoding was rewritten to mimic a method used in the CLINGCON encoding that also does not support the cumulative global constraint. Columns EZSMT-CVC4, EZSMT-Z3, EZCSP in Table 2 represent instances run on the rewritten encoding. Solver EZSMT times out the least, followed by CLINGCON timing out on over one-third the instances, and finally EZCSP, which times out on about half the instances. We note that on the original encoding with cumulative constraint EZCSP performance is captured by the following numbers 26691 (14). Thus, the use of the cumulative global constraint allowed EZCSP to run more instances to completion. All solvers time out on the same 5 instances, which EZSMT-CVC4 and EZSMT-Z3 times out on.

The last three lines in Table 2 report on the three benchmarks from [16]. In general, we observe that CLINGCON features the fastest performance, followed by EZSMT and MINGO for these benchmarks.

Overall, the benchmarks reveal several aspects of the EZSMT solver. First, as demonstrated by the reverse folding results in Table 2, the underlying SMT solving technology selected for the SMT-LIB program produced by EZSMT is important. Next, the weighted sequence and the incremental scheduling results demonstrate the efficacy of EZSMT approach. Furthermore, Table 2 shows that EZSMT outperforms MINGO across the board.

The ASPMT2SMT system [5] is closely related to EZSMT in a sense that it utilizes SMT technology for finding solutions to first order formulas under stable model semantics forming so called ASPMT language. The EZ programs can be seen as a special case of ASPMT formulas. Just as EZSMT poses restriction on its programs to be tight, ASPMT2SMT poses a similar restriction on its theories. The ASPMT2SMT solver utilizes SMT solver Z3 to find models of ASPMT theories by interfacing this system via its API. This tight integration with Z3 allows ASPMT2SMT to find multiple/all models of its theories in contrast to EZSMT. Yet, the fact that EZSMT advocates the use of the standard SMT-LIB language makes its approach more open towards new developments in the SMT solving technology as it is not tied to any particular SMT solver via its specific API. We do not present the times for the ASPMT2SMT system as the ASPMT language differs from the input languages of other systems that we experimented with so that encodings of our benchmarks for ASPMT2SMT are not readily available. Yet, EZSMT-Z3 times should mimic these by ASPMT2SMT as both systems rely on forming program's completion in the process of translating inputs in their respective languages into SMT formulas. Verifying this claim is part of the future work.

6 Conclusions and Future Work

This work presents the EZSMT system, which is able to take tight constraint answer set programs and rewrite them into the SMT-LIB formulas that can be then processed by SMT solvers. The EZSMT solver parallels the efforts of the ASPMT2SMT system [5] that utilizes SMT technology for solving programs in related formalism. Our experimental analysis illustrates that the EZSMT system is capable of outperforming other cutting-edge CASP solvers. Niemela [18] characterized answer sets of "normal" logic programs in terms of "level rankings" and developed a mapping from such programs to so called difference logic. Mapping of the kind has been previously exploited in the design of solvers DINGO [13] and MINGO [16]. We believe that these ideas are applicable in the settings of EZ(IL) and EZ(L) programs. Verifying this claim and adopting the results within EZSMT to allow this solver to process non tight programs is the direction of future work.

Acknowledgments. We would like to thank Martin Brain for the discussions that led us to undertake this research.

References

1 Marcello Balduccini. Representing constraint satisfaction problems in answer set programming. In *ICLP Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP)*, 2009. URL: `https://www.mat.unical.it/ASPOCP09/`.

2 Marcello Balduccini and Yuliya Lierler. Constraint answer set solver EZCSP and why integration schemas matter. Unpublished draft, available at `https://works.bepress.com/yuliya_lierler/64/`, 2016.

3 Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV'11), volume 6806 of LNCS*. Springer, 2011.

4 Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.5. Technical report, Department of Computer Science, The University of Iowa, 2015.

5 Michael Bartholomew and Joohyung Lee. System aspmt2smt: Computing aspmt theories by smt solvers. In *European Conference on Logics in Artificial Intelligence, JELIA*, pages 529–542. Springer, 2014. `doi:10.1007/978-3-319-11558-0_37`.

6 Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

7 Christian Drescher and Toby Walsh. A translational approach to constraint answer set solving. *Theory and Practice of Logic programming (TPLP)*, 10(4-6):465–480, 2010.

8 François Fages. Consistency of Clark's completion and existence of stable models. *Journal of Methods of Logic in Computer Science*, 1:51–60, 1994.

9 Paolo Ferraris and Vladimir Lifschitz. Weight constraints as nested expressions. *Theory and Practice of Logic Programming*, 5:45–74, 2005.

10 Martin Gebser, Roland Kaminski, Arne König, and Torsten Schaub. Advances in gringo series 3. In *Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, pages 345–351. Springer, 2011. `doi:10.1007/978-3-642-20895-9_39`.

11 Martin Gebser, Max Ostrowski, and Torsten Schaub. Constraint answer set solving. In *Proceedings of 25th International Conference on Logic Programming*, pages 235–249. Springer, 2009.

12 Enrico Giunchiglia, Yuliya Lierler, and Marco Maratea. Answer set programming based on propositional satisfiability. *Journal of Automated Reasoning*, 36:345–377, 2006.

13 Tomi Janhunen, Guohua Liu, and Ilkka Niemela. Tight integration of non-ground answer set programming and satisfiability modulo theories. In *Proceedings of the 1st Workshop on Grounding and Transformations for Theories with Variables*, 2011.

14 Yuliya Lierler and Benjamin Susman. Constraint answer set programming versus satisfiability modulo theories. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI)*, 2016.

15 Vladimir Lifschitz, Lappoon R. Tang, and Hudson Turner. Nested expressions in logic programs. *Annals of Mathematics and Artificial Intelligence*, 25:369–389, 1999.

16 Guohua Liu, Tomi Janhunen, and Ilkka Niemela. Answer set programming via mixed integer programming. In *Knowledge Representation and Reasoning Conference*, 2012. URL: `https://www.aaai.org/ocs/index.php/KR/KR12/paper/view/4516`.

17 Kim Marriott and Peter J. Stuckey. *Programming with Constraints: An Introduction*. MIT Press, 1998.

18 Ilkka Niemelä. Stable models and difference logic. *Annals of Mathematics and Artificial Intelligence*, 53:313–329, 2008.

19 Ilkka Niemelä and Patrik Simons. Extending the Smodels system with cardinality and weight constraints. In Jack Minker, editor, *Logic-Based Artificial Intelligence*, pages 491–521. Kluwer, 2000.

Theory Solving Made Easy with *Clingo* 5*

Martin Gebser[1], Roland Kaminski[2], Benjamin Kaufmann[3], Max Ostrowski[4], Torsten Schaub[5], and Philipp Wanko[6]

1 University of Potsdam, Potsdam, Germany
2 University of Potsdam, Potsdam, Germany
3 University of Potsdam, Potsdam, Germany
4 University of Potsdam, Potsdam, Germany
5 University of Potsdam, Potsdam, Germany; and INRIA, Rennes, France
6 University of Potsdam, Potsdam, Germany

Abstract

Answer Set Programming (ASP) is a model, ground, and solve paradigm. The integration of application- or theory-specific reasoning into ASP systems thus impacts on many if not all elements of its workflow, viz. input language, grounding, intermediate language, solving, and output format. We address this challenge with the fifth generation of the ASP system *clingo* and its grounding and solving components by equipping them with well-defined generic interfaces facilitating the manifold integration efforts. On the grounder's side, we introduce a generic way of specifying language extensions and propose an intermediate format accommodating their ground representation. At the solver end, this is accompanied by high-level interfaces easing the integration of theory propagators dealing with these extensions.

1998 ACM Subject Classification D.1.6 Logic Programming

Keywords and phrases Answer Set Programming, Theory Language, Theory Propagation

Digital Object Identifier 10.4230/OASIcs.ICLP.2016.2

1 Introduction

The *clingo* system, along with its grounding and solving components *gringo* and *clasp*, is nowadays among the most widely used tools for Answer Set Programming (ASP; [22]). This does not only apply to end-users, but more and more to system developers who build upon *clingo*'s infrastructure for developing their own systems. Among them, we find (alphabetically) *clasp-nk* [13], *clingcon* [25], *dflat* [1], *dingo* [21], *dlvhex* [14], *inca* [12], and *mingo* [23]. None of these systems can use *clingo* or its components without workarounds or even involved modifications to realize the desired functionality. Moreover, since ASP is a model, ground, and solve paradigm, such modifications are rarely limited to a single component but often spread throughout the whole workflow. This begins with the addition of new language constructs to the input language, requiring in turn amendments to the grounder as well as syntactic means for passing the ground constructs to a downstream system. In case they are to be dealt with by an ASP solver, it must be enabled to treat the specific input and incorporate corresponding solving capacities. Finally, each such extension is application-specific and requires different means at all ends.

* This work was partially supported by DFG-SCHA-550/9.

We address this challenge with the new *clingo* series 5 and its components. This is accomplished by introducing generic interfaces that allow for accommodating extensions to ASP at the salient stages of its workflow. To begin with, we extend *clingo*'s grounder component *gringo* with means for specifying simple theory grammars in which new theories can be represented. As theories are expressed using constructs close to ASP's basic modeling language, the existing grounding machinery takes care of instantiating them. This also involves a new intermediate ASP format that allows for passing the enriched information from grounders to solvers in a transparent way. (Since this format is mainly for settings with stand-alone grounders and solvers, and thus outside the scope of *clingo*, we delegate details to [17].) For a complement, *clingo* 5 provides several interfaces for reasoning with theory expressions. On the one hand, the existing Lua and Python APIs are extended by high-level interfaces for augmenting propagation in *clasp* with so-called *theory propagators*. Several such propagators can be registered with *clingo*, each implementing an interface of four basic methods. Our design is driven by the objective to provide means for rapid prototyping of dedicated reasoning procedures while enabling effective implementations. To this end, the interface supports, for instance, stateful theory propagators as well as multi-threading in the underlying solver. On the other hand, the functionality of the aforementioned extended APIs is now also offered via a C interface. This is motivated by the wide availability of foreign function interfaces for C, which enable the import of *clingo* in programming languages like Java or Haskell. A first application of this is the integration of *clingo* 5 into SWI-Prolog.[1]

2 Input Language

This section introduces the novel features of *clingo* 5's input language. All of them are situated in the underlying grounder *gringo* 5 and can thus also be used independently of *clingo*. We start with a detailed description of *gringo* 5's generic means for defining theories and afterwards summarize further new features.

Our generic approach to theory specification rests upon two languages: the one defining theory languages and the theory language itself. Both borrow elements from the underlying ASP language, foremost an aggregate-like syntax for formulating variable length expressions. To illustrate this, consider Listing 1, where a logic program is extended by constructs for handling difference and linear constraints. While the former are binary constraints of the form $x_1 - x_2 \leq k$, the latter have a variable size and are of form $a_1x_1 + \cdots + a_nx_n \circ k$, where x_i are integer variables, a_i and k are integers, and $\circ \in \{\leq, \geq, <, >, =\}$ for $1 \leq i \leq n$.[2] Note that solving difference constraints is polynomial, while solving linear equations (over integers) is NP-hard. The theory language for expressing both types of constraints is defined in Lines 1–13 and preceded by the directive `#theory`. The elements of the resulting theory language are preceded by `&` and used as regular atoms in the logic program in Lines 15–21.

To be more precise, a *theory definition* has the form

$$\texttt{\#theory}\ T\ \texttt{\{}D_1\texttt{;}\ldots\texttt{;}D_n\texttt{\}.}$$

where T is the theory name and each D_i is a definition for a theory term or a theory atom for $1 \leq i \leq n$. The language induced by a theory definition is the set of all theory atoms constructible from its theory atom definitions.

A *theory atom definition* has form

$$\texttt{\&}p\texttt{/}k\ \texttt{:}\ t\texttt{,}o \quad \text{or} \quad \texttt{\&}p\texttt{/}k\ \texttt{:}\ t\texttt{,\{}\diamond_1\texttt{,}\ldots\texttt{,}\diamond_m\texttt{\},}t'\texttt{,}o$$

[1] `https://github.com/JanWielemaker/clingo`
[2] For simplicity, we consider normalized difference constraints rather than general ones of form $x_1 - x_2 \circ k$.

```
 1 #theory difference {
 2   constant    { - : 0, unary };
 3   diff_term   { - : 0, binary, left };
 4   linear_term { + : 2, unary; - : 2, unary;
 5                 * : 1, binary, left;
 6                 + : 0, binary, left; - : 0, binary, left };
 7   domain_term { .. : 1, binary, left };
 8   show_term   { / : 1, binary, left };
 9   &dom/0 : domain_term, {=}, linear_term, any;
10   &sum/0 : linear_term, {<=,=,>=,<,>,!=}, linear_term, any;
11   &diff/0 : diff_term, {<=}, constant, any;
12   &show/0 : show_term, directive
13 }.

15 #const n=2.  #const m=1000.
16 task(1..n).  duration(T,200*T) :- task(T).
17 &dom  { 1..m } = start(T)  :- task(T).
18 &dom  { 1..m } = end(T)    :- task(T).
19 &diff { end(T)-start(T) } <= D :- duration(T,D).
20 &sum  { end(T) : task(T); -start(T) : task(T) } <= m.
21 &show { start/1; end/1 }.
```

Listing 1 Logic program enhanced with difference and linear constraints (`diff.lp`).

where p is a predicate name and k its arity, t, t' are names of theory term definitions, each $\diamond_i$ is a theory operator for $m \geq 1$, and $o \in \{$`head`, `body`, `any`, `directive`$\}$ determines where theory atoms may occur in a rule. Examples of theory atom definitions are given in Lines 9–12 of Listing 1. The language of a theory atom definition as above contains all *theory atoms* of form

$$\&a\ \{C_1 : L_1 ; \dots ; C_n : L_n\} \quad \text{or} \quad \&a\ \{C_1 : L_1 ; \dots ; C_n : L_n\} \diamond c$$

where a is an atom over predicate p of arity k, each C_i is a tuple of theory terms in the language for t, c is a theory term in the language for t', $\diamond$ is a theory operator among $\{\diamond_1, \dots, \diamond_m\}$, and each L_i is a regular condition (i.e., a tuple of regular literals) for $1 \leq i \leq n$. Whether the last part '$\diamond c$' is included depends on the form of a theory atom definition. Five occurrences of theory atoms can be found in Lines 17–21 of Listing 1.

A *theory term definition* has form

$$t\ \{D_1 ; \dots ; D_n\}$$

where t is a name for the defined terms and each D_i is a theory operator definition for $1 \leq i \leq n$. A respective definition specifies the language of all theory terms that can be constructed via its operators. Examples of theory term definitions are given in Lines 2–8 of Listing 1. Each resulting *theory term* is one of the following:

- a constant term: c
- a variable term: v
- a binary theory term: $t_1 \diamond t_2$
- a unary theory term: $\diamond t_1$
- a function theory term: $f(t_1, \dots, t_k)$
- a tuple theory term: $(t_1, \dots, t_l,)$
- a set theory term: $\{t_1, \dots, t_l\}$
- a list theory term: $[t_1, \dots, t_l]$

```
1  task(1).                                  task(2).
2  duration(1,200).                          duration(2,400).

4  &dom  { 1..1000 } = start(1).      &dom  { 1..1000 } = start(2).
5  &dom  { 1..1000 } = end(1).        &dom  { 1..1000 } = end(2).
6  &diff { end(1)-start(1) } <= 200.  &diff { end(2)-start(2) } <= 400.
7  &sum  { end(1); end(2); -start(1); -start(2) } <= 1000.
8  &show { start/1; end/1 }.
```

Listing 2 Human-readable result of grounding Listing 1 via '`gringo -text diff.lp`'.

where each t_i is a theory term, $\diamond$ is a theory operator defined by some D_i, c and f are symbolic constants, v is a first-order variable, $k \geq 1$, and $l \geq 0$. (The trailing comma in tuple theory terms is optional if $l \neq 1$.) Parentheses can be used to specify operator precedence.

A *theory operator definition* has form

```
◇ : p,unary     or     ◇ : p,binary,a
```

where $\diamond$ is a unary or binary theory operator with precedence $p \geq 0$ (determining implicit parentheses). Binary theory operators are additionally characterized by an associativity $a \in \{$`right`, `left`$\}$. As an example, consider Line 5 of Listing 1, where the `binary` operator `*` is defined with precedence `1` and `left` associativity. In total, Lines 2–8 of Listing 1 include nine theory operator definitions. Particular *theory operators* can be assembled (written consecutively without spaces) from the symbols '`!`', '`<`', '`=`', '`>`', '`+`', '`-`', '`*`', '`/`', '`\`', '`?`', '`&`', '`|`', '`.`', '`:`', '`;`', '`~`', and '`^`'. For instance, in Line 7 of Listing 1, the operator '`..`' is defined as the concatenation of two periods. The tokens '`.`', '`:`', '`;`', and '`:-`' must be combined with other symbols due to their dedicated usage. Instead, one may write '`..`', '`::`', '`;;`', '`::-`', etc.

While theory terms are formed similar to regular ones, theory atoms rely upon an aggregate-like construction for forming variable-length theory expressions. In this way, standard grounding techniques can be used for gathering theory terms. (However, the actual atom within a theory atom comprises regular terms only.) The treatment of theory terms still differs from their regular counterparts in that the grounder skips simplifications like, e.g., arithmetic evaluation. This can be nicely seen on the different results in Listing 2 of grounding terms formed with the regular and theory-specific variants of operator '`..`'. Observe that the fact `task(1..n)` in Line 16 of Listing 1 results in `n` ground facts, viz. `task(1)` and `task(2)` because of `n=2`. Unlike this, the theory expression `1..m` stays structurally intact and is only transformed into `1..1000` in view of `m=1000`. That is, the grounder does not evaluate the theory term `1..1000` and leaves its interpretation to a downstream theory solver.

A similar situation is encountered when comparing the treatment of the regular term '`200*T`' in Line 16 of Listing 1 to the theory term '`end(T)-start(T)`' in Line 19. While each instance of '`200*T`' is evaluated during grounding, instances of the theory term are left in Line 6 of Listing 2. In fact, if '`200*T`' had been a theory term as well, it would have resulted in the unevaluated instances '`200*1`' and '`200*2`'.

The remainder of this section is dedicated to other language extensions of *gringo* 5 aiming at a disentanglement of the various uses of `#show` directives (and their induced symbol table). Such directives were beforehand used for controlling the output of stable models, delineating the scope of reasoning modes (e.g., intersection, union, projection, etc.), and for passing special-purpose information to downstream systems. For instance, theory and heuristic information was passed to *clasp* via dedicated predicates like `_edge` and `_heuristic`. This entanglement brought about several shortcomings. In fact, passing information via a symbol table did not only scramble the output, but also provoked overhead in grounding and filtering “artificial” symbolic information.

Now, in *gringo* 5, the sole purpose of `#show` is to furnish an output directive. There are three different kinds of such statements:

```
#show.      #show p/n.     #show t : l1,...,ln.
```

The first form hides all atoms, and the second all except those over predicates p/n indicated by `#show` statements. The third form does not hide any atoms and can be used to output arbitrary terms t, whenever the literals $l_1, \dots, l_n$ in the condition after ‘:’ hold. This is particularly useful in meta-programming, e.g., ‘`#show A : holds(A).`’ can be used to map back reified atoms.

Atoms used in reasoning modes are indicated by `#project` directives, having two forms:

```
#project p/n.    #project a : l1,...,ln.
```

Here, p is a predicate name with arity n, a is an atom, and $l_1, \dots, l_n$ are literals. While the first form declares all atoms over predicate p/n as subject to projection, the second includes instances of a obtained via grounding, as detailed in [18] for `#external` directives.

The last two new directives of interest abolish the need for the special-purpose predicates `_edge` and `_heuristic`, previously used in conjunction with the ASP solver *clasp*:

```
#edge (u,v) : l1,...,ln.
#heuristic a : l1,...,ln. [k@p,m]
```

As above, a is an atom, and $l_1, \dots, l_n$ are literals. Moreover, u, v, k, p, m are terms, where ‘`(u,v)`’ stands for an edge from u to v in an acyclicity extension [8]. Integer values for k and p along with `init`, `factor`, `level`, `sign`, `true`, or `false` for m determine a heuristic modifier [19]. Finally, note that zero is taken as default priority when the optional ‘`@p`’ part in ‘`[k@p,m]`’, resembling the syntax of ranks for weak constraints [10], is omitted.

3 Logical Characterization

The semantics of logic programs modulo theories rests upon ground programs P over two disjoint alphabets, $\mathcal{A}$ and $\mathcal{T}$, consisting of regular and *theory atoms*. Accordingly, P is a set of rules r of the form $h \leftarrow a_1, \dots, a_m, {\sim}a_{m+1}, \dots, {\sim}a_n$, where the head h is constant $\bot$, a_0 or $\{a_0\}$ for an atom $a_0 \in \mathcal{A} \cup \mathcal{T}$, and $\{a_1, \dots, a_n\} \subseteq \mathcal{A} \cup \mathcal{T}$. If $h = \bot$, r is called an *integrity constraint*, a *normal rule* if $h = a_0$, or a *choice rule* if $h = \{a_0\}$; as usual, we skip $\bot$ when writing integrity constraints. We let $h(r) = \emptyset$ for an integrity constraint r, $h(r) = \{a_0\}$ for a normal or choice rule r, and define $h(P) = \bigcup_{r \in P} h(r)$ as the *head atoms* of P. In analogy to inputs atoms from `#external` directives [18], we partition $\mathcal{T}$ into *defined* theory atoms $\mathcal{T} \cap h(P)$ and *external* theory atoms $\mathcal{T} \setminus h(P)$.

Given a collection $\mathbb{T}$ of theories, we associate each $T \in \mathbb{T}$ with a *scope* $\mathcal{T}^T$ of atoms relevant to T, and let $\mathcal{T} = \bigcup_{T \in \mathbb{T}} \mathcal{T}^T$ be the corresponding set of theory atoms. Reconsidering the input language in Section 2, a natural choice for $\mathcal{T}^T$ consists of all (ground) atoms declared within a `#theory` directive for T. However, as we see in Section 5, a scope may in general include atoms written in regular as well as extended syntax (the latter preceded by ‘`&`’) in the input language.

In order to reflect different forms of theory propagation, we further consider a partition of the scope $\mathcal{T}^T$ of a theory T into *strict* theory atoms $\mathcal{T}^T_e$ and *non-strict* theory atoms $\mathcal{T}^T_i$ such that $\mathcal{T}^T_e \cap \mathcal{T}^T_i = \emptyset$ and $\mathcal{T}^T_e \cup \mathcal{T}^T_i = \mathcal{T}^T$. The strict theory atoms in $\mathcal{T}^T_e$ resemble equivalences as expressed by the constraint atoms of *clingcon* [25], which must be assigned to true iff their associated constraints hold. This is complemented by viewing the non-strict theory atoms in $\mathcal{T}_i^T$ as implications similar to the constraint statements of *ezcsp* [2], where only statements assigned to true impose requirements, while constraints associated with false ones are free to hold or not. Given the distinction of respective kinds of theory atoms, a combined theory T may integrate constraints according to the semantics of *clingcon* and *ezcsp*, e.g., indicated by dedicated predicates or arguments thereof in T's theory language.

We now turn to mapping the semantics of logic programs modulo theories back to regular stable models. In the abstract sense, we call any $\mathcal{S}^T \subseteq \mathcal{T}^T$ a *T-solution* if T is consistent with the conditions expressed by elements of $\mathcal{S}^T$ as well as the complements of conditions associated with the false strict theory atoms in $\mathcal{T}_e^T \setminus \mathcal{S}^T$.[3] Generalizing this concept to a collection $\mathbb{T}$ of theories, we say that $\mathcal{S} \subseteq \mathcal{T}$ is a *$\mathbb{T}$-solution* if $\mathcal{S} \cap \mathcal{T}^T$ is a T-solution for each $T \in \mathbb{T}$. Then, we define a set $X \subseteq \mathcal{A} \cup \mathcal{T}$ of (regular and theory) atoms as a $\mathbb{T}$-*stable model* of a ground program P if there is some $\mathbb{T}$-solution $\mathcal{S}$ such that X is a (regular) stable model of the program

$$P \cup \{a \leftarrow \mid T \in \mathbb{T}, a \in (\mathcal{T}_e^T \setminus h(P)) \cap \mathcal{S}\} \cup \{\leftarrow {\sim}a \mid T \in \mathbb{T}, a \in (\mathcal{T}_e^T \cap h(P)) \cap \mathcal{S}\} \tag{1}$$

$$\cup \{\{a\} \leftarrow \mid T \in \mathbb{T}, a \in (\mathcal{T}_i^T \setminus h(P)) \cap \mathcal{S}\} \cup \{\leftarrow a \mid T \in \mathbb{T}, a \in (\mathcal{T}^T \cap h(P)) \setminus \mathcal{S}\}. \tag{2}$$

That is, the rules added to P in (1) and (2) express conditions aligning $X \cap \mathcal{T}$ with an underlying $\mathbb{T}$-solution $\mathcal{S}$. First, the facts in (1) make sure that external theory atoms that are strict, i.e., included in $\mathcal{T}_e^T \setminus h(P)$ for some $T \in \mathbb{T}$, and hold in $\mathcal{S}$ belong to X as well. Unlike this, the corresponding set of choice rules in (2) merely says that non-strict external theory atoms from $\mathcal{S}$ may be included in X, thus not insisting on a perfect match between non-strict theory atoms and elements of $\mathcal{S}$. Moreover, the integrity constraints in (1) and (2) take care of defined theory atoms belonging to $h(P)$. The respective set in (1) again focuses on strict theory atoms and stipulates the ones from $\mathcal{S}$ to be included in X as well. In addition, for both strict and non-strict defined theory atoms, the integrity constraints in (2) assert the falsity of atoms that do not hold in $\mathcal{S}$.

For example, consider a program $P = \{a \leftarrow b, {\sim}c\}$ subject to some theory T with the strict and non-strict theory atoms $\mathcal{T}_e^T = \{a, b\}$ and $\mathcal{T}_i^T = \{c\}$, and let $\mathcal{S} = \{a, b, c\}$ be a T-solution. Then, the extended program for $\mathcal{S}$ is $P \cup \{b \leftarrow ; \{c\} \leftarrow ; \leftarrow {\sim}a\}$, whose (only) regular stable model $X = \{a, b\}$ is a $\{T\}$-stable model of P. Note that $\mathcal{S}$ assigns the non-strict theory atom c to true, while X excludes it to keep $a \leftarrow b, {\sim}c$ applicable for the (strict) defined theory atom a.

To summarize the main principles of the $\mathbb{T}$-stable model concept, strict theory atoms (for some $T \in \mathbb{T}$) must exactly match their interpretation in a $\mathbb{T}$-solution $\mathcal{S}$, while non-strict ones (not strict for any $T \in \mathbb{T}$) in X are only required not to exceed $\mathcal{S}$. Second, external theory atoms that hold in $\mathcal{S}$ are mapped to facts or choice rules, while conditions on defined ones are enforced by means of integrity constraints. As a result, $\mathbb{T}$-stable models are understood as regular stable models, yet relative to extensions of a given program P determined by underlying $\mathbb{T}$-solutions. Notably, the concept of $\mathbb{T}$-stable models also carries on to logic programs allowing for further constructs, such as weight constraints and disjunction, which have not been discussed here for brevity (cf. [26]).

[3] Although we omit formal details, atoms in Satisfiability Modulo Theories (SMT; [4]) belong to first-order predicates interpreted in a theory T, and the ones that hold in some model of T provide a T-solution $\mathcal{S}^T \subseteq \mathcal{T}^T$.

4 Algorithmic Characterization

As detailed in [20], a ground program P induces *completion* and *loop nogoods*, given by $\Delta_P = \Delta_{B(P)} \cup \Delta_{\mathcal{A}\cup(\mathcal{T}\cap h(P))}$ or $\Lambda_P = \bigcup_{\emptyset\subset U\subseteq\mathcal{A}\cup(\mathcal{T}\cap h(P))}\{\lambda(a,U) \mid a \in U\}$, respectively, where $B(P)$ is the set of rule bodies occurring in P. Note that both sets of nogoods are restricted by regular atoms and defined theory atoms, while external theory atoms can a priori be assigned freely, although any occurrences in rule bodies are subject to evaluation via respective nogoods in $\Delta_{B(P)}$. A (partial) *assignment* $\mathbf{A}$ is a consistent set of *(signed) literals* of the form $\mathbf{T}v$ or $\mathbf{F}v$ for $v \in (\mathcal{A}\cup\mathcal{T})\cup B(P)$, i.e., $\{\mathbf{T}v,\mathbf{F}v\} \not\subseteq \mathbf{A}$ for all $v \in (\mathcal{A}\cup\mathcal{T})\cup B(P)$; $\mathbf{A}$ is *total* if $\{\mathbf{T}v,\mathbf{F}v\}\cap\mathbf{A} \neq \emptyset$ for all $v \in (\mathcal{A}\cup\mathcal{T})\cup B(P)$. We say that some nogood δ is *violated* by $\mathbf{A}$ if $\delta \subseteq \mathbf{A}$. When $\mathbb{T} = \emptyset$, so that $\mathcal{T} = \emptyset$ as well, each total assignment $\mathbf{A}$ that does not violate any nogood $\delta \in \Delta_P \cup \Lambda_P$ yields a regular stable model of P, and such an assignment $\mathbf{A}$ is called a *solution* (for $\Delta_P \cup \Lambda_P$).

We now extend the concept of a solution to $\mathbb{T}$-stable models. To this end, we follow the idea of external propagators in [12] and identify a theory $T \in \mathbb{T}$ with a set $\Delta_T \subseteq 2^{\{\mathbf{T}a \mid a\in\mathcal{T}^T\}\cup\{\mathbf{F}a \mid a\in\mathcal{T}^T_e\}}$ of *theory nogoods* such that, given a total assignment $\mathbf{A}$, we have that $\delta \subseteq \mathbf{A}$ for some $\delta \in \Delta_T$ iff there is no T-solution $\mathcal{S}^T$ such that $\{a \in \mathcal{T}^T \mid \mathbf{T}a \in \mathbf{A}\} \subseteq \mathcal{S}^T$ and $\{a \in \mathcal{T}^T_e \mid \mathbf{F}a \in \mathbf{A}\} \cap \mathcal{S}^T = \emptyset$. That is, the nogoods in Δ_T must reject $\mathbf{A}$ iff no T-solution (i) includes all theory atoms in $\mathcal{T}^T$ that are assigned to true by $\mathbf{A}$ and (ii) excludes all strict theory atoms in $\mathcal{T}^T_e$ assigned to false by $\mathbf{A}$. This semantic condition establishes a (one-to-one) correspondence between $\mathbb{T}$-stable models of P and solutions for $(\Delta_P \cup \Lambda_P) \cup \bigcup_{T\in\mathbb{T}} \Delta_T$. A formal elaboration can be found in [17].

The nogoods in $(\Delta_P \cup \Lambda_P) \cup \bigcup_{T\in\mathbb{T}} \Delta_T$ provide the logical fundament for the Conflict-Driven Constraint Learning (CDCL) procedure (cf. [24, 20]) outlined in Figure 1. While the completion nogoods in Δ_P are usually made explicit and subject to unit propagation, the loop nogoods in Λ_P as well as theory nogoods in Δ_T are typically handled by dedicated propagators and particular members are selectively recorded, i.e., when a respective propagator identifies some nogood δ such that $|\delta \setminus \mathbf{A}| \leq 1$ (and $(\{\mathbf{T}v \mid \mathbf{F}v \in \delta\} \cup \{\mathbf{F}v \mid \mathbf{T}v \in \delta\}) \cap \mathbf{A} = \emptyset$), and we say that such a nogood is *unit*. In fact, a unit nogood δ yields either a conflict, if δ is violated by $\mathbf{A}$, or otherwise a literal to be assigned by unit propagation.

While the dedicated propagator for loop nogoods is built-in in systems like *clingo* 5, those for theories are provided via the interface detailed in Section 5. To utilize custom propagators, Figure 1 includes an *initialization* step in Line (I). In addition to the "registration" of a propagator for a theory T as an extension of the basic CDCL procedure, common tasks performed in this step include setting up internal data structures and so-called watches for (a subset of) the theory atoms in $\mathcal{T}^T$, so that the propagator will be invoked (only) when some watched literal gets assigned.

The main CDCL loop starts with unit propagation on completion and loop nogoods, the latter handled by the respective built-in propagator, as well as any nogoods already recorded. If this results in a non-total assignment without conflict, theory propagators for which some of their watched literals have been assigned are invoked in Line (P). A propagator for a theory T can then inspect the current assignment, update its data structures accordingly, and most importantly, perform *theory propagation* determining theory nogoods $\delta \in \Delta_T$ to record. Usually, any such nogood δ is unit in order to trigger a conflict or unit propagation, although this is not a necessary condition. The interplay of unit and theory propagation continues until a conflict or total assignment arises, or no (further) watched literals of theory propagators get assigned by unit propagation. In the latter case, some non-deterministic decision is made to extend the partial assignment at hand and then to proceed with unit and theory propagation.

```
(I)   initialize                                 // register theory propagators and initialize watches
      loop
          propagate completion, loop, and recorded nogoods        // deterministically assign
          if no conflict then
              if all variables assigned then
(C)               if some $\delta \in \Delta_T$ is violated for $T \in \mathbb{T}$ then record $\delta$      // check $\Delta_T$
                  else return variable assignment                 // $\mathbb{T}$-stable model found
              else
(P)               propagate theories $T \in \mathbb{T}$     // possibly record theory nogoods from $\Delta_T$
                  if no nogood recorded then decide         // non-deterministically assign
          else
              if top-level conflict then return unsatisfiable
              else
                  analyze                 // resolve conflict and record a conflict constraint
(U)               backjump              // undo assignments until conflict constraint is unit
```

Figure 1 Basic algorithm for Conflict-Driven Constraint Learning (CDCL) modulo theories.

If no conflict arises and an assignment **A** is total, in Line (C), theory propagators are called, one by one, for a final *check* of **A**. The idea is that, e.g., a "lazy" propagator for a theory T that does not exhaustively test violations of its theory nogoods by partial assignments can make sure that **A** is indeed a solution for Δ_T, or record some violated nogood(s) from Δ_T otherwise. Even in case theory propagation on partial assignments is exhaustive and a final check is not needed to detect conflicts, the information that search led to a total assignment can be useful in practice, e.g., to store values for integer variables like `start(1)`, `start(2)`, `end(1)`, and `end(2)` in Listing 2 that witness the existence of a T-solution.

Finally, in case of a conflict, i.e., some completion or recorded nogood is violated by the current assignment, provided that some non-deterministic decision is involved in the conflict, a new conflict constraint is recorded and utilized to guide backjumping in Line (U), as usual with CDCL. In a similar fashion as the assignment of watched literals serves as trigger for theory propagation, theory propagators are informed when they become unassigned upon backjumping. This allows them to *undo* earlier operations, e.g., internal data structures can be reset to return to a state taken prior to the assignment of watches.

In summary, the basic CDCL procedure is extended in four places to account for custom propagators: initialization, propagation of (partial) assignments, final check of total assignments, and undo steps upon backjumping.

5 Propagator Interface

We now turn to the implementation of theory propagation in *clingo* 5 and detail the structure of its interface depicted in Figure 2.

The interface `Propagator` has to be implemented by each custom propagator. After registering such a propagator with *clingo*, its functions are called during initialization and search as indicated in Figure 1. Function `Propagator.init`[4] is called once before solving (Line (I) in Figure 1) to allow for initializing data structures used during theory propagation.

[4] For brevity, we below drop the qualification `Propagator` and use its function names unqualified.

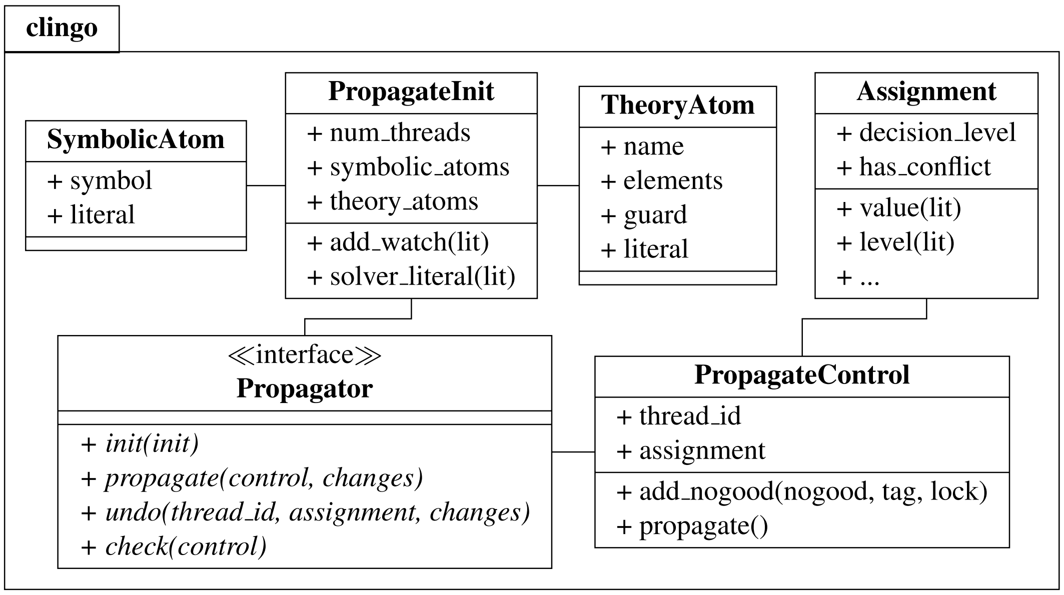


Figure 2 Class diagram of *clingo*'s (theory) propagator interface.

It is invoked with a `PropagateInit` object providing access to symbolic (`SymbolicAtom`) as well as theory (`TheoryAtom`) atoms. Both kinds of atoms are associated with program literals,[5] which are in turn associated with solver literals.[6] Program as well as solver literals are identified by non-zero integers, where positive and negative numbers represent positive or negative literals, respectively. In order to get notified about assignment changes, a propagator can set up watches on solver literals during initialization.

During search, function `propagate` is called with a `PropagateControl` object and a (non-empty) list of watched literals that got assigned in the recent round of unit propagation (Line (P) in Figure 1). The `PropagateControl` object can be used to inspect the current assignment, record nogoods, and trigger unit propagation. Furthermore, to support multi-threaded solving, its `thread_id` property identifies the currently active thread, each of which can be viewed as an independent instance of the CDCL algorithm in Figure 1.[7] Function `undo` is the counterpart of `propagate` and called whenever the solver retracts assignments to watched literals (Line (U) in Figure 1). In addition to the list of watched literals that have been retracted (in chronological order), it receives the identifier and the assignment of the active thread. Finally, function `check` is similar to `propagate`, yet invoked without a list of changes. Instead, it is (only) called on total assignments (Line (C) in Figure 1), independently of watches. Overriding the empty default implementations of propagator methods is optional. For brevity, we below focus on implementations of the methods in Python, while Lua or C could be used as well.

For illustration, consider Listing 3 giving a propagator for (half of) the pigeon-hole problem.

[5] Program literals are also used in the *aspif* format (see [17]).
[6] Note that *clasp*'s preprocessor might associate a positive or even negative solver literal with multiple atoms.
[7] Depending on the configuration of *clasp*, threads can communicate with each other. For example, some of the recorded nogoods can be shared. This is transparent from the perspective of theory propagators.

```
 1  #script (python)

 3  class Pigeonator:
 4      def __init__(self):
 5          self.place = {} # shared state
 6          self.state = [] # per thread state

 8      def init(self, init):
 9          for atom in init.symbolic_atoms.by_signature("place", 2):
10              lit = init.solver_literal(atom.literal)
11              self.place[lit] = atom.symbol.args[1]
12              init.add_watch(lit)
13          self.state = [ {} for _ in range(init.num_threads) ]

15      def propagate(self, control, changes):
16          holes = self.state[control.thread_id]
17          for lit in changes:
18              hole = self.place[lit]
19              prev = holes.setdefault(hole, lit)
20              if prev != lit and not control.add_nogood([lit, prev]):
21                  return

23      def undo(self, thread_id, assignment, changes):
24          holes = self.state[thread_id]
25          for lit in changes:
26              hole = self.place[lit]
27              if holes.get(hole) == lit:
28                  del holes[hole]

30  def main(prg):
31      prg.register_propagator(Pigeonator())
32      prg.ground([("base", [])])
33      prg.solve()

35  #end.

37  1 { place(P,H) : H = 1..h } 1 :- P = 1..p.
38  % { place(P,H) : P = 1..p } 1 :- H = 1..h.
```

Listing 3 Propagator for the pigeon-hole problem.

Although this setting is constructed, it showcases central aspects that are also relevant when implementing more complex propagators, e.g., the `Pigeonator` is both stateful and can be used with multiple threads. The underlying ASP encoding is given in Line 37: A (choice) rule generates solution candidates by placing each of the p pigeons in exactly one among h holes. While the rule commented out in Line 38 would ensure that there is at most one pigeon per hole, this constraint is handled by the `Pigeonator` class implementing the `Propagator` interface (except for `check`) in Lines 8–28. Whenever two pigeons are placed in the same hole, it adds a binary nogood forbidding the placement. To this end, it maintains data structures for, given a newly placed pigeon, detecting whether there is a conflict. More precisely, the propagator has two data members: The `self.place` dictionary in Line 5 maps solver literals for `place`/2 atoms to their corresponding holes, and the `self.state` list in

Line 6 stores for each solver thread its current placement of pigeons as a mapping from holes to true solver literals for `place`/2 atoms.

Function `init` in Lines 8–13 sets up watches as well as the dictionaries in `self.place` and `self.state`. To this end, it traverses (symbolic) atoms over `place`/2 in Lines 9–12. Each such atom is associated with a solver literal, obtained in Line 10. The mapping from the solver literal to its corresponding hole is then stored in the `self.place` dictionary in Line 11. In the last line of the loop, a watch is added for each solver literal at hand, so that the solver calls `propagate` whenever a pigeon is placed. Finally, in Line 13, the `self.state` list of placements per thread, subject to change upon propagation and backjumping, is initialized with empty dictionaries.

Function `propagate`, given in Lines 15–21, accesses `control.thread_id` in Line 16 to obtain the `holes` dictionary storing the active thread's current placement of pigeons. The loop in Lines 17–21 then iterates over the list of changes, i.e., solver literals representing newly placed pigeons. After in Line 18 determining the `hole` associated with a recently assigned literal, Python's `setdefault` function is used to update the state: Depending on whether `hole` already appears as a key in the `holes` dictionary, the function either retrieves its associated literal or inserts the new literal under key `hole`. While the latter case amounts to updating the placement of pigeons, the former signals a conflict, triggered by recording a binary nogood in Line 20. Given that the solver has to resolve the conflict and backjump, the call to `add_nogood` always yields false, so that propagation stops without processing remaining changes any further.[8]

Function `undo` in Lines 23–28 resets a thread's placement of pigeons upon backjumping. Similar to `propagate`, the active thread's current placement is obtained in Line 24, and changes are traversed in Lines 25–28. The latter correspond to retracted solver literals, for which the condition in Line 27 makes sure that exactly those stored in Line 19 before are cleared, thus reflecting that the `hole` determined in Line 26 is free again. Finally, function `main` in Lines 30–33 first registers the `Pigeonator` propagator in Line 31, and then initiates grounding and solving with *clingo*.

6 Experiments

Our approach aims at a simple yet general framework for incorporating theory reasoning into ASP solving. Hence, it leaves room for various ways of encoding a problem and of implementing theory propagation. To reflect this from a practical perspective, we empirically explore several options for solving problems with difference logic (DL) constraints. To be more precise, we contrast an encoding relying on *defined* theory atoms with one leaving them *external* (cf. Section 3), and a *stateless* with a *stateful* propagator implementation. As a *non-strict* interpretation of DL constraints is sufficient for the problems given below, we stick to this option and do not vary it.

The consistency of a set C of DL constraints can be checked by mapping them to a weighted directed graph $G(C)$. The nodes of $G(C)$ are the (integer) variables occurring in C, and for each $x_1 - x_2 \leq k$ in C, $G(C)$ includes an edge from x_1 to x_2 with weight k. Then, C is DL-consistent iff $G(C)$ contains no cycle whose sum of edge weights is negative. The difference between a stateless and stateful DL-propagator amounts to whether the

[8] The optional arguments `tag` and `lock` of `add_nogood` can be used to control the scope and lifetime of recorded nogoods. Furthermore, in a propagator that does not add violated nogoods only, function `control.propagate` can be invoked to trigger unit propagation.

Table 1 Comparison between different encodings and *DL*-propagators for scheduling problems.

		ASP		ASP modulo *DL* (stateless)				ASP modulo *DL* (stateful)			
				defined		external		defined		external	
Problem	#	T	TO	T	TO	T	TO	T	TO	T	TO
Flow shop	120	569	110	283	40	382	70	**177**	**30**	281	50
Job shop	80	600	80	600	80	600	80	**37**	**0**	43	**0**
Open shop	60	405	40	214	20	213	20	**2**	**0**	**2**	**0**
Total	260	525	230	366	140	398	170	**72**	**30**	109	50

corresponding graph is built from scratch upon each invocation or only once and updated subsequently. In our experiments, we use the Bellman-Ford algorithm [6, 16] as basis for a stateless propagator, and the one in [11] for update operations in the stateful case. Both propagator implementations detect negative cycles and record (solver) literals corresponding to their weighted edges as nogoods.

Theory atoms corresponding to *DL* constraints are formed as described in Section 2. The difference between using defined and external theory atoms boils down to their occurrence in the head of a rule, as in Line 19 of Listing 1, viz.

```
&diff { end(T)-start(T) } <= D :- duration(T,D).
```

or in the body, as in

```
:- duration(T,D), not &diff { end(T)-start(T) } <= D.
```

Note that the defining usage constrains *DL*-atoms firmer than the external one: A defined *DL*-atom is true iff at least one of its bodies holds, while an external one may vary whenever its truth is *DL*-consistent yet not imposed by integrity constraints (with further problem-specific literals).

To evaluate the different options, we expressed (decision versions of) several scheduling problems [27], typically aiming at the minimization of schedules' makespan, by logic programs in the language of Section 2. *Flow shop*: A schedule corresponds to a permutation of n jobs, each including m sequential subtasks allocating machines $1, \dots, m$ for specific amounts of time. *Job shop*: Again considering n jobs with m sequential subtasks each, where the order in which subtasks allocate machines $1, \dots, m$ for given amounts of time is job-specific, a schedule arranges the subtasks of different jobs in one sequence per machine. *Open shop*: Given the same setting as in the job shop problem, the sequential order of the subtasks of a job is not fixed, but augments a schedule arranging the subtasks of different jobs per machine. For reasons of scalability, we refrain from optimizing the makespan of schedules, but are only interested in some feasible schedule per instance along with the corresponding earliest start times of subtasks.

The results of our experiments, run sequentially under Linux on an Intel Xeon E5520 2.27 GHz machine equipped with 24 GB main memory, are summarized in Table 1. Each *clingo* 5 run was restricted to 600 seconds wall-clock time, while memory was never exceeded. Subcolumns headed by 'T' report average runtimes, taking timeouts as 600 seconds, and those with 'TO' numbers of timeouts over '#' instances of each scheduling problem and in total. Respective results in the column headed by 'ASP' reflect the bottom-line performance obtained with plain ASP encodings, which is obviously not competitive due to the ineffectiveness of grounding problems over large numeric domains. The remaining columns consider the four combinations of encoding and *DL*-propagator features of interest. First, we observe that the stateful propagator (on the right) has a clear edge over its stateless counterpart (in the middle). Second, with both propagator implementations, the firm encoding using defined *DL*-atoms outperforms the one leaving them external on instances of the flow shop problem. While this experiment is not meant to be universal, it demonstrates that different features have an impact on the resulting performance. In how far the tuning of theory propagators matters also depends on the use case at hand, e.g., solving a challenging application problem versus rapid prototyping of dedicated reasoning procedures.

7 Discussion

The *clingo* 5 system provides a comprehensive infrastructure for enhancing ASP with theory reasoning. This ranges from generic means for expressing theories along with their support by *gringo*, over a theory-aware intermediate format, to simple yet powerful interfaces in C, Lua, and Python. In each case, a propagator can specify (up to) four basic functions to customize its integration into *clasp*'s propagation, where an arbitrary number of (independent) theory propagators can be incorporated. Logically, ASP encodings may build upon defined or external theory atoms, and their associated conditions may be strict or non-strict. In practice, *clingo* 5 supports stateless and stateful theory propagators, which can be controlled in a fine-grained way. For instance, propagators are thread-sensitive, watches can be set to symbolic as well as theory literals, and the scope and lifetime of nogoods stemming from theory propagation can be configured.

A first step toward a more flexible ASP infrastructure was done with *clingo* 4 [18] by introducing Lua and Python APIs for multi-shot solving. Although this allows for fine-grained control of complex ASP reasoning processes, the functionality provided no access to *clasp*'s propagation and was restricted to inspecting (total) stable models. The extended framework for theory propagation relative to partial assignments (cf. Figure 1) follows the canonical approach of SMT [4]. While *dlvhex* implicitly provides access to *clasp*'s propagation, this is done on the more abstract level of higher-order logic programs. Also, *dlvhex* as well as many other systems, such as *clingcon* or *inca*, implement specialized propagation via *clasp*'s internal interfaces, whose usage is more involved and subject to change with each release. Although the new high-level interfaces may not yet fully cover all desired features, they provide a first step toward easing the development of such dedicated systems and putting them on a more stable basis. Currently, *clingo* 5's infrastructure is already used as a basis for *clingcon* 3 [3], *lc2casp* [9], and its integration with SWI-Prolog. Finally, we believe that the extended grounding capacities along with the intermediate format supplemented in [17] will also be beneficial for non-native approaches and ease the overall development of ASP-oriented solvers. This applies to systems like *dingo*, *mingo*, and *aspmt* [5], the latter implementing ASP with theory reasoning by translation to SMT, which so far had to resort to specific input formats and meta-programming to bypass the grounder.

References

1 M. Abseher, B. Bliem, G. Charwat, F. Dusberger, M. Hecher, and S. Woltran. The D-FLAT system for dynamic programming on tree decompositions. In Fermé and Leite [15], pages 558–572.

2 M. Balduccini. Representing constraint satisfaction problems in answer set programming. In *Proceedings of the Second Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP'09)*, pages 16–30, 2009.

3 M. Banbara, B. Kaufmann, M. Ostrowski, and T. Schaub. Clingcon: The next generation. *Submitted for publication*, 2016.

4 C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli. Satisfiability modulo theories. In Biere et al. [7], pages 825–885.

5 M. Bartholomew and J. Lee. System aspmt2smt: Computing ASPMT theories by SMT solvers. In Fermé and Leite [15], pages 529–542.

6 R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.

7 A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability.* IOS Press, 2009.

8 J. Bomanson, M. Gebser, T. Janhunen, B. Kaufmann, and T. Schaub. Answer set programming modulo acyclicity. In *Proceedings of the Thirteenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'15)*, pages 143–150. Springer, 2015.

9 P. Cabalar, R. Kaminski, M. Ostrowski, and T. Schaub. An ASP semantics for default reasoning with constraints. In *Proceedings of the Twenty-fifth International Joint Conference on Artificial Intelligence (IJCAI'16)*, pages 1015–1021. IJCAI/AAAI Press, 2016.

10 F. Calimeri, W. Faber, M. Gebser, G. Ianni, R. Kaminski, T. Krennwallner, N. Leone, F. Ricca, and T. Schaub. ASP-Core-2: Input language format. Available at `https://www.mat.unical.it/aspcomp2013/ASPStandardization/`, 2012.

11 S. Cotton and O. Maler. Fast and flexible difference constraint propagation for DPLL(T). In *Proceedings of the Ninth International Conference on Theory and Applications of Satisfiability Testing (SAT'06)*, pages 170–183. Springer, 2006.

12 C. Drescher and T. Walsh. Answer set solving with lazy nogood generation. In *Technical Communications of the Twenty-eighth International Conference on Logic Programming (ICLP'12)*, pages 188–200. Leibniz International Proceedings in Informatics, 2012.

13 T. Eiter, E. Erdem, H. Erdogan, and M. Fink. Finding similar/diverse solutions in answer set programming. *Theory and Practice of Logic Programming*, 13(3):303–359, 2013.

14 T. Eiter, M. Fink, T. Krennwallner, and C. Redl. Conflict-driven ASP solving with external sources. *Theory and Practice of Logic Programming*, 12(4-5):659–679, 2012.

15 E. Fermé and J. Leite, editors. *Proceedings of the Fourteenth European Conference on Logics in Artificial Intelligence (JELIA'14)*, volume 8761 of *Lecture Notes in Artificial Intelligence.* Springer, 2014.

16 L. Ford and D. Fulkerson. *Flows in networks.* Princeton University Press, 1962.

17 M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and P. Wanko. Theory solving made easy with clingo 5 (extended version). Available at `http://www.cs.uni-potsdam.de/wv/publications/`, 2016.

18 M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. *Clingo* = ASP + control: Preliminary report. In *Technical Communications of the Thirtieth International Conference on Logic Programming (ICLP'14)*, 2014. Available at `http://arxiv.org/abs/1405.3694`.

19 M. Gebser, B. Kaufmann, R. Otero, J. Romero, T. Schaub, and P. Wanko. Domain-specific heuristics in answer set programming. In *Proceedings of the Twenty-Seventh National Conference on Artificial Intelligence (AAAI'13)*, pages 350–356. AAAI Press, 2013.

20 M. Gebser, B. Kaufmann, and T. Schaub. Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence*, 187-188:52–89, 2012.

21 T. Janhunen, G. Liu, and I. Niemelä. Tight integration of non-ground answer set programming and satisfiability modulo theories. In *Proceedings of the First Workshop on Grounding and Transformation for Theories with Variables (GTTV'11)*, pages 1–13, 2011.

22 V. Lifschitz. What is answer set programming? In *Proceedings of the Twenty-third National Conference on Artificial Intelligence (AAAI'08)*, pages 1594–1597. AAAI Press, 2008.

23 G. Liu, T. Janhunen, and I. Niemelä. Answer set programming via mixed integer programming. In *Proceedings of the Thirteenth International Conference on Principles of Knowledge Representation and Reasoning (KR'12)*, pages 32–42. AAAI Press, 2012.

24 J. Marques-Silva, I. Lynce, and S. Malik. Conflict-driven clause learning SAT solvers. In Biere et al. [7], pages 131–153.

25 M. Ostrowski and T. Schaub. ASP modulo CSP: The clingcon system. *Theory and Practice of Logic Programming*, 12(4-5):485–503, 2012.

26 P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1-2):181–234, 2002.

27 E. Taillard. Benchmarks for basic scheduling problems. *European Journal of Operational Research*, 64(2):278–285, 1993.

Computing Diverse Optimal Stable Models[*]

Javier Romero[1], Torsten Schaub[2], and Philipp Wanko[3]

1 **University of Potsdam, Potsdam, Germany**
2 **University of Potsdam, Potsdam, Germany; and INRIA, Rennes, France**
3 **University of Potsdam, Potsdam, Germany**

Abstract

We introduce a comprehensive framework for computing diverse (or similar) solutions to logic programs with preferences. Our framework provides a wide spectrum of complete and incomplete methods for solving this task. Apart from proposing several new methods, it also accommodates existing ones and generalizes them to programs with preferences. Interestingly, this is accomplished by integrating and automating several basic ASP techniques — being of general interest even beyond diversification. The enabling factor of this lies in the recent advance of multi-shot ASP solving that provides us with fine-grained control over reasoning processes and abolishes the need for solver modifications and wrappers that were indispensable in previous approaches. Our framework is implemented as an extension to the ASP-based preference handling system *asprin*. We use the resulting system *asprin* 2 for an empirical evaluation of the diversification methods comprised in our framework.

1998 ACM Subject Classification D.1.6 Logic Programming, F.4.1 Mathematical Logic

Keywords and phrases Answer Set Programming, Diversity, Similarity, Preferences

Digital Object Identifier 10.4230/OASIcs.ICLP.2016.3

1 Introduction

Answer Set Programming (ASP; [5]) has become a prime paradigm for solving combinatorial problems in Knowledge Representation and Reasoning. As a matter of fact, such problems have an exponential number of solutions in the worst-case. A first means to counterbalance this is to impose preference relations among solutions to filter out optimal ones. Often enough, this still leaves us with a large number of optimal models. A typical example is the computation of Pareto frontiers for multi-objective optimization problems [19], as we encounter in design space exploration [3] or timetabling [4]. Other examples include configuration, planning, and phylogeny, as discussed in [9]. This calls for computational support that allows for identifying small subsets of diverse solutions. The computation of diverse stable models was first considered in ASP by [9]. The analogous problem regarding optimal stable models is addressed in [25] in the case of answer set optimization [8]. Beyond ASP, the computation of diverse solutions is also studied in CP [17] and SAT [18].

In this paper, we introduce a comprehensive framework for computing diverse (or similar) solutions to logic programs with preferences. One of its distinguishing factors is that it allows for dealing with aggregated (or plain) qualitative and quantitative preferences among stable models of logic programs. This is accomplished by building on the preference handling capacities of *asprin* [6]. The other appealing factor of our framework is that it covers a wide

[*] This work was partially supported by DFG-SCHA-550/9 and 11.

spectrum of methods for diversification. Apart from new techniques, it also accommodates and generalizes existing approaches by lifting them to programs with preferences. Interestingly, this is done by taking advantage of several existing basic ASP techniques that we automate and integrate in our framework. The enabling factor of this is the recent advance of multi-shot ASP solving that allows for an easy yet fine-grained control of ASP-based reasoning processes (cf. [13]). In particular, this abolishes the need for internal solver modifications or singular solver wrappers that were often unavoidable in previous approaches. We have implemented our approach as an extension to the preference handling framework *asprin*. The resulting system *asprin* 2 is then used for an empirical evaluation contrasting several alternative approaches to computing diverse solutions. Last but not least, note that although we concentrate on diversity, our approach applies just as well to the dual concept of *similarity*. This is also reflected by its implementation supporting both settings.

2 Background

In ASP, problems are described as (disjunctive) *logic programs*, being sets of *rules* of the form

$$\texttt{a}_1\texttt{;}\ldots\texttt{;a}_m \texttt{ :- } \texttt{a}_{m+1}\texttt{,}\ldots\texttt{,a}_n\texttt{,not a}_{n+1}\texttt{,}\ldots\texttt{,not a}_o$$

where each $\texttt{a}_i$ is a propositional atom and `not` stands for *default negation*. We call a rule a *fact* if $m = o = 1$, *normal* if $m = 1$, and an *integrity constraint* if $m = 0$. We may reify a rule r with the set of facts $R(r) = \{\texttt{rule}(r)\} \cup \{\texttt{head}(r\texttt{,a}_i) \mid 1 \leq i \leq m\} \cup \{\texttt{body}(r\texttt{,pos,a}_i) \mid m+1 \leq i \leq n\} \cup \{\texttt{body}(r\texttt{,neg,a}_i) \mid n+1 \leq i \leq o\}$, and we reify a program by joining its reified rules. Semantically, a logic program induces a collection of *stable models*, which are distinguished models of the program determined by stable models semantics; see [16] for details.

To ease the use of ASP in practice, several extensions have been developed. First of all, rules with variables are viewed as shorthands for the set of their ground instances. Further language constructs include *conditional literals* and *cardinality constraints* [24]. The former are of the form $\texttt{a:b}_1\texttt{,}\ldots\texttt{,b}_m$, the latter can be written as $\texttt{s\{c}_1\texttt{;}\ldots\texttt{;c}_n\texttt{\}t}$, where `a` and $\texttt{b}_i$ are possibly default-negated literals and each $\texttt{c}_j$ is a conditional literal; `s` and `t` provide lower and upper bounds on the number of satisfied literals in the cardinality constraint. We refer to $\texttt{b}_1\texttt{,}\ldots\texttt{,b}_m$ as a *condition*. The practical value of both constructs becomes apparent when used with variables. For instance, a conditional literal like `a(X):b(X)` in a rule's antecedent expands to the conjunction of all instances of `a(X)` for which the corresponding instance of `b(X)` holds. Similarly, `2{a(X):b(X)}4` is true whenever at least two and at most four instances of `a(X)` (subject to `b(X)`) are true. Specifically, we rely in the sequel on the input language of the ASP system *clingo* [13]; further language constructs are explained on the fly.

In what follows, we go beyond plain ASP and deal with *logic programs with preferences*. More precisely, we consider programs P over some set $\mathcal{A}$ of atoms along with a strict partial order $\succ \subseteq \mathcal{A} \times \mathcal{A}$ among their stable models. Given two stable models X, Y of P, $X \succ Y$ means that X is preferred to Y. Then, a stable model X of P is *optimal* wrt$\succ$, if there is no other stable model Y such that $Y \succ X$. In what follows, we often leave the concrete order implicit and simply refer to a program with preferences and its optimal stable models. We restrict ourselves to partial orders and distance measures (among pairs of stable models) that can be computed in polynomial time. For simplicity, we focus on the Hamming distance, defined for two stable models X, Y of a program P over $\mathcal{A}$ as $d(X, Y) = |(\mathcal{A} - X) - Y| + |X \cap Y|$. Given a logic program P with preferences and a positive integer n, we define a set $\mathcal{X}$ of optimal stable models of P as *most diverse*, if $\min\{\, d(X, Y) \mid X, Y \in \mathcal{X}, X \neq Y\} > \min\{\, d(X, Y) \mid X, Y \in \mathcal{X}', X \neq Y\}$ for every other set

$\mathcal{X}'$ of optimal stable models of P. We are thus interested, following [9], in the problem *n Most Diverse Optimal Models*: Given a logic program P with preferences and a positive integer n, find n most diverse optimal stable models of P.

For representing logic programs with complex preferences and computing their optimal models, we built upon the preference framework of *asprin* [6], a system for dealing with aggregated qualitative and quantitative preferences. In *asprin*, the above mentioned *preference relations* are represented by declarations of the form `#preference(p,t){t`$_1$`:b`$_1$`,...,t`$_n$`:b`$_n$`}` where `p` and `t` are the name and type of the preference relation, and `t`$_i$ and `b`$_i$ are tuples of terms and conditions, respectively,[1] serving as arguments of `p`. The directive `#optimize(p)` instructs *asprin* to search for stable models that are optimal wrtthe strict partial order $\succ_{\mathtt{p}}$ associated with `p`. While *asprin* already comes with a library of predefined primitive and aggregate preference types, like `subset` or `pareto`, respectively, it also allows for adding customized preferences. We illustrate this by implementing preference type `maxmin` in Section 4.

Finally, we investigate whether the heuristic capacities of *clingo* allow for boosting our approach. In fact, *clingo* 5 features heuristic directives of the form '`#heuristic c. [`k`,`m`]`' where c is a conditional atom, k is a term evaluating to an integer, and m is a heuristic modifier among `init`, `factor`, `level`, or `sign`. The effect of the heuristic modifiers is to bias the score of *clasp*'s heuristic by initially adding or multiplying the score, prioritizing variables, or preferably assigning a truth value, respectively. The value of k serves as argument to the respective modification. A more detailed description can be found in [15].

3 Our Diversification Framework at a Glance

We begin with an overview over the various techniques integrated in our framework.

3.1 Basic solving techniques

We first summarize several basic solving techniques that provide essential pillars of our framework and that are also of interest for other application areas.

Maxmin optimization is a popular strategy in game theory and beyond that is not supported by existing ASP systems. We address this issue and consider *maxmin* (and *minmax*) optimization that, given a set of sums, aims at maximizing the value of the minimum sum. We have implemented both preference types and made them available via *asprin* 2's library.

Guess and Check automation. [11] defined a framework for representing and solving Σ_2^p problems in ASP. Given two normal logic programs P and Q capturing a guess-and-check (G&C) problem, X is a solution to $\langle P, Q\rangle$ if X is a stable model of P and $Q \cup X$ is unsatisfiable. We *automatize* this by using reification along with the meta-encoding methodology of *metasp* [14]. In this way, the two normal programs P and Q are transformed into a single disjunctive logic program. The resulting mini-system *metagnc* is implemented in Python and available at [1]. We build upon this approach for computing optimal models of logic programs with preferences, providing an alternative method to the iterative one of [6]. For this, *asprin* translates a logic program with preferences into a G&C problem, which is then translated by *metagnc* into a disjunctive logic program and solved by an ASP system.

Querying programs with preferences consists of deciding whether there is an optimal stable model of a program P with preferences that contains a given query atom q. To this end, we elaborate upon four alternatives:

[1] See [6] for more general preference elements.

Q-1. Enumerate *models* of $P \cup \{\bot \leftarrow \textit{not } q\}$ until one is an optimal model of P.

Q-2. Enumerate *optimal models* of P until one contains q.

Q-3. Enumerate *optimal models* of $P \cup \{\bot \leftarrow \textit{not } q\}$ until one is an optimal model of P.

Q-4. Enumerate *optimal models* of P until one contains q while alternately adding $\{\bot \leftarrow \textit{not } q\}$ or $\{\bot \leftarrow q\}$ during model-driven optimization.

The first two methods were implemented by [25] in the case of programs with *aso* preferences [8]. We generalize both to arbitrary preferences, propose two novel ones, and provide all four methods in *asprin* 2. Applications of querying programs with preferences are clearly of greater interest and go well beyond diversification.

Preferences over optimal models allow for further narrowing down the stable models of interest by imposing a selection criterion among the optimal models of a logic program with preferences. For one thing, this is different from a lexicographic preference, since the secondary preference takes into account all optimal models wrtthe first preference, no matter whether they are equal or incomparable. For another, it aims at preference combinations whose complexity goes beyond the expressiveness of ASP and thus cannot be addressed via an encoding in *asprin*. Rather, we conceived a nested variant of *asprin*'s optimization algorithm that computes the preferred optimal models. Interestingly, this makes use of our querying capacities in posing the "improvement constraint" as a query.

3.2 Advanced diversification techniques

We elaborate upon three ways of diversification, viz. enumeration, replication, and approximation, for solving the *n Most Diverse Optimal Models* problem. While the two former return an optimal solution, the latter simply approximates it.

Enumeration consists of two steps:

1. Enumerate all optimal models of the logic program P with preferences.
2. Find among all computed optimal models, the n most diverse ones.

While we carry out the first step by means of *asprin*'s enumeration mode, we cast the second one as an optimization problem and express it as a logic program with preferences. This method was first used by [9] for addressing diversity in the context of logic programs without preferences; we lift it here to programs with preferences.

Replication consists of three steps:

1. Translate a normal logic program P with preferences into a disjunctive logic program D by applying the aforementioned guess-and-check method.
2. Reify D into $\mathcal{R}(D)$, and add a meta-encoding M replicating D such that each stable model of $M \cup \mathcal{R}(D)$ corresponds to n optimal models of the original logic program P.
3. Turn the disjunctive logic program $M \cup \mathcal{R}(D)$ into a *maxmin* optimization problem by applying the aforementioned method such that its optimal stable models correspond to n most diverse optimal stable models of the original program P with preferences.

This method was outlined for logic programs without preferences in [9] but not automated. We generalize this approach to normal programs with preferences and provide a fully automated approach.

Approximation. Our approximation techniques can be understood as instances of the following algorithm, whose input is a logic program with preferences P:

1. Find an optimal model X of P. If P is unsatisfiable then return $\{\bot\}$, else assign $\mathcal{X} = \{X\}$.
2. While $\textit{test}(\mathcal{X})$ is true, call $\textit{solve}(P, \mathcal{X})$ and add the solution to $\mathcal{X}$.
3. Return $\textit{solution}(\mathcal{X})$.

In the basic case, $\mathit{test}(\mathcal{X})$ returns *true* until there are n solutions in $\mathcal{X}$, $\mathit{solution}(\mathcal{X})$ returns the set $\mathcal{X}$, and the algorithm simply computes n solutions by successively calling $\mathit{solve}(P, \mathcal{X})$. More elaborate approaches are obtained, for example, computing $n+k$ solutions and returning the n most diverse among them in $\mathit{solution}(\mathcal{X})$.

The implementation of $\mathit{solve}(P, \mathcal{X})$ leads to different approaches:

A-1. $\mathit{solve}(P, \mathcal{X})$ returns an optimal model of P maximizing the minimum distance to the solutions in $\mathcal{X}$. We accomplish this by defining a *maxmin* preference, and imposing this on top of the optimal models of P by applying the two aforementioned approaches to *maxmin* optimization and preferences over optimal models. This method was first used by [9] for addressing diversity in the context of logic programs without preferences; we lift it here to programs with preferences.

A-2. $\mathit{solve}(P, \mathcal{X})$ first computes a partial interpretation I of P maximizing the minimum distance to the solutions in $\mathcal{X}$, and then returns an optimal model of P closest to I:

- **(a)** Select a partial interpretation I of P in one of the following ways: (i) a random one, (ii) a heuristically chosen one, (iii) one most diverse wrtthe solutions in $\mathcal{X}$, or (iv) one complementary to the last computed optimal model.
- **(b)** Use a cardinality-based preference minimizing the distance to I, and apply the aforementioned approach to preferences over optimal models to enforce this preference among the optimal models of P.

A-3. $\mathit{solve}(P, \mathcal{X})$ approximates *A-2* using heuristics. To this end, we select a partial interpretation I as in *A-2*, and then guide the computation of the optimal model fixing the sign of the atoms to their value in I. The approach is further developed prioritizing the variables in I. A similar method was used in [18] for SAT.

4 Basic Solving Techniques

We first show how the *Most Distant (Optimal) Model* problem can be represented in *asprin* using the new preference type *maxmin*: Given a logic program P (with preferences) over $\mathcal{A}$, and a set $\mathcal{X} = \{X_1, \dots, X_m\}$ of (optimal) stable models of P, find an (optimal) stable model of P that maximizes the minimum distance to the (optimal) stable models in $\mathcal{X}$. The *Most Distant (Optimal) Model* is used by our approximation algorithms in Section 5.

Maxmin optimization in *asprin*. Let $H_{\mathcal{X}}$ be the set of facts $\{$`holds(`a`,`i`).`$\mid a \in X_i, X_i \in \mathcal{X}\}$ reifying the stable models in $\mathcal{X}$, and let `distance` be the following preference statement:

```
#preference(distance,maxmin){
  I,1,X : holds(X,0), not holds(X,I), I=1..m;
  I,1,X : not holds(X,0), holds(X,I), I=1..m }.
```

Then, the *Most Distant Model* problem is solved by the following program with preferences: $P \cup \{$`holds(`a`,0) :- `a`.` $\mid a \in \mathcal{A}\} \cup H_{\mathcal{X}} \cup$ `{distance}` $\cup \{$`#optimize(distance).`$\}$. P generates stable models that are reified with `holds(`a`,0)` for $a \in \mathcal{A}$. The preference statement `distance` represents a `maxmin` preference over m sums, where the value of each sum `I` (with `I=1..`m) amounts to the distance between the generated stable model and $X_{\mathtt{I}}$. Finally, the optimize statement selects the optimal stable models wrt$\succ_{\mathtt{distance}}$.

Formally, the preference elements of preference type *maxmin* have the restricted form '`s,w,t:B`' where `s`, `w`, `t` are terms, and `B` is a condition. Term `s` names different sums, whose value is specified by the rest of the element '`w,t:B`' (similar to aggregate elements). For defining the semantics of *maxmin*, preference elements stand for their ground instantiations, and we consider a set E of such ground preference elements. We say that `s` is (the name of) a sum of E if it is the first term of some preference element. Given a stable model X and a sum $\mathtt{s}$ of E, the value of $\mathtt{s}$ in X is:

$$v(\mathtt{s}, X) = \sum_{(\mathtt{w},\mathtt{t}) \in \{\mathtt{w},\mathtt{t} \mid \mathtt{s},\mathtt{w},\mathtt{t}:\mathtt{B} \in E, X \models \mathtt{B}\}} \mathtt{W}$$

For a set E of ground preference elements for preference statement `p`, *maxmin* defines the following preference relation: [2]

$$X \succ_{\mathtt{p}} Y \text{ if } \min\{v(\mathtt{s}, X) \mid \mathtt{s} \text{ is a sum of } E\} > \min\{v(\mathtt{s}, Y) \mid \mathtt{s} \text{ is a sum of } E\}$$

Applying this definition to the preference statement `distance` gives the partial order $\succ_{\mathtt{distance}}$.

In *asprin*, partial orders $\succ$ are implemented by so-called *preference programs.* For our example, we say that Q is a *preference program* for $\succ_{\mathtt{distance}}$ if it holds that $X \succ_{\mathtt{distance}} Y$ iff $Q \cup H_X \cup H'_Y$ is satisfiable, where $H_X = \{\mathtt{holds}(a)\mathtt{.} \mid a \in X\}$ and $H'_Y = \{\mathtt{holds'}(a)\mathtt{.} \mid a \in Y\}$. In practice, the preference program Q consists of three parts.

First, each preference statement is translated into a set of facts, and added to Q. Our example preference statement `distance` results in `preference(distance,maxmin)` and the instantiations of `preference(distance,1,(I,X),for(t₁),(I,1,X))` and `preference(distance,2,(I,X), for(t₂),(I,1,X))` where $\mathtt{t}_1$ and $\mathtt{t}_2$ are terms standing for the conditions of the two non-ground preference elements.

Second, Q contains the implementation of the preference type `p`, consisting of rules defining an atom `better(p)` that indicates whether $X \succ_{\mathtt{p}} Y$ holds for two stable models X, Y. The sets X and Y are provided by *asprin* in reified form via unary predicates `holds` and `holds'`. [3] Further rules are added by *asprin* to define `holds` and `holds'` for the conditions appearing in the preference statement ($\mathtt{t}_1$ and $\mathtt{t}_2$ in our example). The definition of `better(p)` then draws upon the instances of both predicates for deciding $X \succ_{\mathtt{p}} Y$. For the new preference type `maxmin` (being now part of *asprin* 2's library), we get the following rules:

```
#program preference(maxmin).
sum(P,S) :- preference(P,maxmin), preference(P,_,_,_,(S,_,_)).

value(P,S,V) :- preference(P,maxmin), sum(P,S),
  V = #sum { W,T : holds'(X), preference(P,_,_,for(X),(S,W,T)).

minvalue(P,V) :- preference(P,maxmin), V = #min { W : value(P,S,W) }.

better(P,S) :- preference(P,maxmin), sum(P,S), minvalue(P,V),
  V < #sum { W,T : holds(X), preference(P,_,_,for(X),(S,W,T)).

better(P) :- preference(P,maxmin), better(P,S) : sum(P,S).
```

Predicate `sum/2` stores the sums `S` of the preference statement `P`, while `value/3` collects the value `V` of every sum for the stable model Y, and `minvalue/2` stores the minimum of them. In the end, `better(P)` is obtained if `better(P,S)` holds for all sums `S`, and this is the case whenever the value of the sums for the stable model X is greater than the minimum value for the stable model Y.

Third, the constraint '`:- not better(distance).`' is added to Q, enforcing that the set of rules is satisfiable iff `better(p)` is obtained, which is the case whenever $X \succ_{\mathtt{distance}} Y$.

We can show that for any preference statement `p` of type `maxmin`, the union of the above three sets of rules constitutes a preference program for $\succ_{\mathtt{p}}$.

[2] For defining *minmax*, we simply switch min by max, and $>$ by $<$.
[3] That is, `holds(a)` (or `holds'(a)`) is true iff $\mathtt{a} \in X$ (or $\mathtt{a} \in Y$).

Automatic Guess and Check in *clingo*. Given a logic program P over $\mathcal{A}$, and a preference statement s with preference program Q_s, the optimal models of P wrt$\succ_s$ correspond to the solutions of the G&C problem $\langle P \cup R'_{\mathcal{A}}, P \cup R_{\mathcal{A}} \cup Q_s \rangle$, where R'_X stands for $\{$`holds'(`a`) :-` a`.` $\mid a \in X\}$, and R_X for $\{$`holds(`a`) :-` a`.` $\mid a \in X\}$ given some set X. [4] The guess program generates stable models X of P reified with `holds'/1`, while the check program looks for models better than X wrt$s$ reified with `holds/1`, so that X is optimal whenever the checker along with the `holds'/1` atoms of X becomes unsatisfiable. This correspondence is the basis of a method for computing optimal models in *asprin*, where the logic program with preferences is translated into a G&C problem, that *metagnc* translates into a disjunctive logic program, which is then solved by *clingo*. This allows, for example, for solving the *Most Distant Model* problem using the logic program $P \cup \{$`holds(`a`,0) :-` a`.` $\mid a \in \mathcal{A}\} \cup H_{\mathcal{X}}$ and the preference statement `distance`, with the corresponding preference program comprising the three sets of rules described before.

In general, the G&C framework [11] allows for representing Σ_2^p problems in ASP, and solving them using the *saturation* technique by Eiter and Gottlob in [10]. The idea is to re-express the problem as a positive disjunctive logic program, containing a special-purpose atom `bot`. Whenever `bot` is obtained, saturation derives all atoms (belonging to a "guessed" model). Intuitively, this is a way to materialize unsatisfiability. We automatize this process in *metagnc* by building on the meta-interpretation-based approach of [14]. For a G&C problem $\langle G, C \rangle$ over $\langle \mathcal{A}_G, \mathcal{A}_C \rangle$, the idea is to reify the program $C \cup \{$`{`a`}.`$\mid a \in \mathcal{A}_G\}$ into the set of facts $\mathcal{R}(C \cup \{$`{`a`}.`$\mid a \in \mathcal{A}_G\})$. The latter are combined with the meta-encoding $\mathcal{M}$ from [14] implementing saturation. This leads to the positive disjunctive logic program:

$$\mathcal{R}\big(C \cup \{\texttt{\{}a\texttt{\}.} \mid a \in \mathcal{A}_G\}\big) \cup \mathcal{M}$$

This program has a stable model (excluding `bot`) for each $X \subseteq \mathcal{A}_G$ such that $C \cup X$ is satisfiable, and it has a saturated stable model (including `bot`) if there is no such X. Next, we just have to add the generator program G, map the true and false atoms of G to their counterparts in the positive disjunctive logic program (represented by predicates `true/1` and `false/1`, respectively), and enforce the atom `bot` to hold:

$$\begin{aligned}&\mathcal{R}\big(C \cup \{\texttt{\{}a\texttt{\}.} \mid a \in \mathcal{A}_G\}\big) \cup \mathcal{M} \cup \\ &\quad G \cup \{\texttt{true(}a\texttt{) :- } a\texttt{.} \mid a \in \mathcal{A}_G\} \cup \{\texttt{false(}a\texttt{) :- not } a\texttt{.} \mid a \in \mathcal{A}_G\} \cup \{\texttt{:- not bot.}\}\end{aligned}$$

The stable models of the resulting program correspond to the solutions of the G&C problem.

Solving queries in *asprin*. Given a logic program with preferences P and a query atom `q`, the query problem is to decide whether there is an optimal stable model of P that contains `q`. From the point of view of complexity theory, the problem is Σ_2^p-complete when P is normal. Membership holds because for solving this problem, we can use the G&C method by translating the logic program with preferences into a disjunctive logic program and adding the query as a constraint '`:- not q.`'. Hardness can be proved by a reduction of the problem of deciding the existence of a stable model of a disjunctive logic program P (see [20]).

Alternatively to the G&C approach, we propose four enumeration-based algorithms for solving this problem. All of them search for an optimal model containing the query, and their worst case occurs when there is none and they have to enumerate all solutions.

[4] To avoid the conflict between the atoms of P appearing in both the guesser and the checker, given a model X of $P \cup R'_{\mathcal{A}}$, only the atoms of predicate `holds'/1` in X are passed to the checker. In the system *metagnc* this is declared via directive '`#guess holds'/1.`'

Algorithm **Q-1** enumerates stable models of $P \cup \{$`:- not q.`$\}$ and tests them for optimality, until one test succeeds. In the worst case, **Q-1** enumerates all stable models of the program, but still it runs in polynomial space given that it enumerates normal stable models.

Algorithm **Q-2** enumerates optimal models of P, until one contains q. In the worst case, **Q-2** enumerates all optimal models of P, and this enumeration may need exponential space (see [6]). Note that this exponential blow-up may also occur with the other algorithms **Q-3** and **Q-4**. In addition, even when **Q-2** succeeds in finding an optimal model containing the query, it may have to enumerate many optimal models without the query.

For alleviating this problem, algorithm **Q-3** enumerates optimal models of $P \cup \{$`:- not q.`$\}$, and tests whether they are also optimal for P, until one test succeeds. However, **Q-3** may have to enumerate many non optimal models of P containing the query before finding an optimal one.

Algorithm **Q-4** follows a different approach, enumerating optimal models of P (as **Q-2**) but modifying the iterative algorithm of *asprin* [6] for computing optimal models. The input of *asprin*'s algorithm is a logic program P and a preference statement s with preference program Q_s. It follows these steps:

1. Solve program P and assign the result to Y. Return Y if it is $\bot$.
2. Assign Y to X, and solve program $P \cup Q_s \cup R_{\mathcal{A}} \cup H'_X$ assigning the result to Y.
   If Y is $\bot$, return X, else repeat this step.

Step 2 searches iteratively for better models of P wrt$s$. In Algorithm **Q-3**, it may be the case that first Step 2 is repeated many times computing models of P with the query, and then the test finds a model of P without the query that is better than all those previous models. Algorithm **Q-4** tries to find earlier those models of P without the query. For this, it adds $\{$`:- not q.`$\}$ to P in Step 1 and in the even iterations of Step 2, and it adds $\{$`:- q.`$\}$ in the odd iterations of Step 2. Whenever an even iteration fails to find a model, no better model with the query exists, and the enumeration algorithm restarts the search at Step 1. On the other hand, whenever an odd iteration fails, this shows that there is no better model without the query, proving that the query holds in an optimal model.[5]

Preferences over optimal models in *asprin*. Formally, this extension of *asprin* is defined as follows. Let P be a logic program over $\mathcal{A}$, and let s and t be two preference statements. A stable model X of P is optimal wrt$s$ *and then* t if it is optimal wrt$s$, and there is no optimal model Y of P wrt$s$ such that $Y \succ_t X$. From the point of view of complexity theory, when P is normal, finding a stable model optimal wrt$s$ and then t is $F\Delta_3^p$-hard. We prove this by reducing the problem of finding an optimal stable model of a disjunctive logic program with weight minimization (see [20]). We note that finding a stable model of a normal logic program P with preferences is in $F\Sigma_2^p$, given the translation to disjunctive logic programs using the G&C method. Therefore, assuming $F\Sigma_2^p \neq F\Delta_3^p$, we cannot find a polynomial translation to a normal program with preferences.

It turns out that the *Most Distant Optimal Model* problem can be easily formulated and solved within this approach. Given a logic program P with a preference statement s, and a set $\mathcal{X} = \{X_1, \ldots, X_m\}$ of optimal stable models of P, the most distant optimal models for this problem correspond to the stable models of the logic program $P \cup \{$ `holds(`a`,0) :-` a. $\mid a \in \mathcal{A}\} \cup H_{\mathcal{X}}$ that are optimal wrt$s$ and then `distance`. In *asprin*,

[5] For *finding* an optimal model with the query and not simply *deciding* its existence, Step 2 is repeated with $\{$`:- not q.`$\}$ until the search fails, proving that an optimal model has been found.

this is represented simply by adding to the resulting logic program the preference statements s and `distance`, along with the declarations '`#optimize(`s`).`' and '`#reoptimize(distance).`'.

For computing optimal models of a logic program P over $\mathcal{A}$ wrtpreference statements s and then t, we propose a variant of *asprin*'s iterative algorithm [6]. Let $solveOpt(P, s)$ be the *asprin* procedure for computing one optimal model of P wrt$s$, and let $solveQuery(P, s, q)$ be any of our algorithms for solving the query problem given a logic program P with preference statement s and query atom q. The algorithm follows these steps:

1. Call $solveOpt(P, s)$ and assign the result to Y. Return Y if it is $\bot$.
2. Assign Y to X, and call $solveQuery(P \cup Q_t^* \cup R_{\mathcal{A}} \cup H'_X, s, \texttt{better}(t))$ assigning the result to Y. If Y is $\bot$, return X, else repeat this step.

where Q_t^* is the result of deleting the constraint '`:- not better(`t`).`' from a preference program Q_t for t. The first step of the algorithm computes an optimal model of P wrt$s$. Then Step 2, like in *asprin*'s basic algorithm, searches iteratively for better models. Specifically, it searches for optimal models of P wrt$s$ that are better than X wrt$t$. Note that by construction of Q_t^*, the stable models Y of $P \cup Q_t^* \cup R_{\mathcal{A}} \cup H'_X$ are better than X wrt$t$ iff `better(`t`)`$\in Y$. Then if $solveQuery$ returns a model Y, it contains `better(`t`)`, and therefore it is better than X wrt$t$. On the other hand, if $solveQuery$ returns $\bot$, there is no optimal model of P wrt$s$ that is better than X wrt$t$, and this implies that X is an optimal model wrt$s$ and then t.

5 Advanced Diversification Techniques

Enumeration. With this technique, we first enumerate all optimal stable models of P with *asprin* and afterwards we find, among all those stable models, the n most diverse. For the initial step, we use *asprin*'s enumeration algorithm (see [6]). For the second, let $\mathcal{X} = \{X_1, \ldots, X_m\}$ be the set of m optimal stable models of P. Then, the following encoding along with the facts $H_{\mathcal{X}}$ reifying $\mathcal{X}$ provides a correct and complete solution to the n *Most Diverse Optimal Models* problem:

```
n { sol(1..m) } n.
#preference(enumeration,maxmin) {
  (I,J),1,X : holds(X,I), not holds(X,J), sol(I), sol(J), I < J;
  (I,J),1,X : not holds(X,I), holds(X,J), sol(I), sol(J), I < J;
  (I,J),#sup,0 : sol(I), not sol(J), I < J ;
  (I,J),#sup,0 : not sol(I), sol(J), I < J }.
#optimize(enumeration).
```

The choice rule guesses `n` solutions among `m` in $\mathcal{X}$, and the `enumeration` preference statement selects the optimal ones. In `enumeration`, there is a sum for every pair `(I,J)` with `I < J`. If both `I` and `J` are chosen (first two preference elements) then the sum represents their actual distance. In the other case (last two elements) the sum has the maximum possible value in *asprin* (viz. `#sup`). This allows for comparing only sums of pairs `(I,J)` of selected solutions.

Replication. With this technique *asprin* begins translating a normal logic program with preferences P into a disjunctive logic program D applying the G&C method. Next, D is reified onto $\mathcal{R}(D)$ and combined with a meta-encoding $\mathcal{M}_n$ replicating D: [6]

```
sol(1..n).
holds(A,S) : head(R,A) :- rule(R); sol(S); holds(A,S) : body(R,pos,A);
                                       not holds(A,S) : body(R,neg,A).
```

[6] The actual encoding handles the whole *clingo* language [13] and is more involved.

The stable models of $\mathcal{M}_n \cup \mathcal{R}(D)$ correspond one to one to the elements of $Opt(P)^n$, where $Opt(P)$ stands for the set of all optimal models of P. Further rules are added for having exactly one stable model for every set of n optimal stable models, but we do not detail them here for space reasons. Finally, adding the following preference and optimize statements results in a correct and complete solution to the n *Most Diverse Optimal Models* problem:

```
#preference(replication,maxmin) {
  (I,J),1,X : hold(A,I), not hold(A,J), sol(I), sol(J), I < J ;
  (I,J),1,X : not hold(A,I), hold(A,J), sol(I), sol(J), I < J }.
#optimize(replication).
```

The preference statement is similar to the one for Enumeration, but now the n solutions are generated by the meta-encoding, and all of them are used for calculating the sums.

Approximation. We describe the different implementations of the procedure $solve(P, \mathcal{X})$ outlined in Section 3.

In Algorithm **A-1**, $solve(P, \mathcal{X})$ solves the *Most Distant Optimal Model* problem given the optimal stable models in $\mathcal{X}$, applying the solution described at the end of Section 4.

In Algorithm **A-2**, $solve(P, \mathcal{X})$ first computes a partial interpretation I distant to $\mathcal{X}$ in one of the following ways:

1. A random one (named *rd*).
2. A heuristically chosen one, following the *pguide* heuristic from [18] (*pg*): for an atom a, a is added to I if it is true in $\mathcal{X}$ more often than false, $\neg a$ is added in the opposite case, and nothing happens if there is a tie.
3. One most distant to the solutions in $\mathcal{X}$ (*dist*), computed applying the solution to the *Most Distant Model* problem described at the beginning of Section 4, where the program P is '$\{$`{holds(`a`)}.`$\mid a \in \mathcal{A}\}$'.
4. One complementary to the last computed optimal model L taking into account either true ($\{\neg a \mid a \in L\}$), false ($\{a \mid a \notin L\}$), or both types of atoms ($\{\neg a \mid a \in L\} \cup \{a \mid a \notin L\}$). They are named *true*, *false* and *all*, respectively.

For selecting an optimal model closest to I, the technique is similar to the one for the *Most Distant Optimal Model* problem: we start with the logic program P with preference statement s, and we add the rules $\{$ `holds(`a`,0) :-`a`.` $\mid a \in \mathcal{A}\}$ reifying the atoms of P, the facts $\{$ `holds(`a`,1).` $\mid a \in I \cap \mathcal{A}\} \cup \{$ `nholds(`a`,1).` $\mid \neg a \in I, a \in \mathcal{A}\}$ reifying I, and define the following preference statement:

```
#preference(partial,less(cardinality)) {
  holds(X,0), nholds(X,1); not holds(X,0), holds(X,1) }.
```

Finally, we compute the optimal models of this program wrt$s$ and then `partial` using the method for preferences over optimal models described in Section 4. In **A-3**, we select a distant solution I as we do for **A-2**, and we add the same reifying rules, along with the following heuristic rules for approximating an optimal model of P close to I:

```
#heuristic hold(X,0) :  holds(X,1). [  1, sign ]
#heuristic hold(X,0) : nholds(X,1). [ -1, sign ]
```

For prioritizing the variables in I, we add another two heuristic rules like the previous ones, but replace both `[ 1, sign ]` and `[ -1, sign ]` by `[ 1, level ]`, respectively.

Table 1 Comparison of approximation techniques by (a) runtime and timeouts, (b) diversification quality, and (c) minimum distance.

Class	T	TO
A-3	**165**	**70**
A-3-*true*	200	113
A-3-*all*	202	118
A-3-*rd*	277	280
A-3-*pg*	317	351
A-3-*pg-l-rd*	354	442
A-3-*false*	351	443
A-3-*pg-l*	351	443
A-2-*true*	482	618
A-2-*rd*	474	648
A-1	482	672
A-2-*dist-to*	528	689
A-2-*all*	515	696
A-2-*false*	532	696
A-2-*pg*	542	708
A-2-*dist*	572	773

Class	S	avg
A-1	**15**	0.13
A-2-*dist-to*	14	0.14
A-2-*pg*	13	**0.18**
A-3-*pg-l*	11	0.17
A-3-*pg-l-rd*	10	0.16
A-2-*all*	10	0.15
A-2-*dist*	8	0.07
A-2-*false*	8	0.15
A-2-*true*	7	0.12
A-3-*false*	6	0.16
A-2-*rd*	5	0.12
A-3-*all*	5	0.08
A-3-*true*	4	0.08
A-3-*rd*	2	0.09
A-3-*pg*	1	0.09
A-3	0	0.06

Class	S	avg
A-1	**15**	12.25
A-2-*dist-to*	13	10.38
A-3-*pg-l-rd*	13	11.82
A-2-*dist*	12	5.31
A-3-*pg-l*	12	11.10
A-2-*pg*	10	**12.86**
A-2-*rd*	9	8.77
A-3-*all*	7	3.99
A-3-*true*	6	4.00
A-3-*false*	6	7.07
A-2-*false*	6	6.80
A-2-*all*	4	6.98
A-2-*true*	3	5.31
A-3-*rd*	2	6.43
A-3	2	4.28
A-3-*pg*	0	2.79

6 Experiments

In this section, we present experiments focusing on the *approximation* techniques of the *asprin* system for obtaining most dissimilar optimal solutions. While *enumeration* and *replication* provide exact results, they need to calculate and store a possibly exponential number of optimal models or deal with a large search space, respectively. Those techniques are therefore not effective for most practical applications. For Algorithm **A-2**, we considered the variations *rd*, *pg*, *true*, *false*, and *all* . In *dist*, we issued no timeout for the computation of the partial interpretation, while in *dist-to*, we set a timeout for this computation of half the total possible runtime. For Algorithm **A-3**, we consider the variations that include no extra ASP computation, namely, *rd*, *pg*, *true*, *false*, and *all* . We also evaluated a version without any heuristic modification (named simply **A-3**). Furthermore, following [18], we considered a variation of *pg*, viz. *pg-l*, where the atoms of the selected partial interpretation are given a higher priority, and *pg-l-rd*, extending *pg-l* by fixing initially a random sign to all atoms not appearing in the partial interpretation.

We gathered 186 instances from six different classes: *Design Space exploration (DSE)* from [3], *Timetabling (CTT)* from [4], *Crossing minimization* from the ASP competition 2013, *Metabolic network expansion* from [21], *Biological network repair* from [12] and *Circuit Diagnosis* from [23]. Since we required instances with multiple optimal solutions, we exclusively focused on Pareto optimality. DSE and CTT are inherently multi-objective and therefore we could naturally define a Pareto preference for them. For the other classes, we turned single-objective into multi-objective optimization problems by distributing their optimization statements. First, we split the atoms in the optimization statements into four or eight groups evenly. We chose for each group the same preference type, either cardinality or subset minimization, and aggregated them by means of Pareto preference. We calculated optimal solutions regarding these Pareto preferences. The same was done for CTT and DSE. An instance was selected if for some Pareto preference ten optimal solutions could be obtained within 600 seconds by *asprin*. This method generated 816 instances in total. We ran the benchmarks on a cluster of Linux machines with dual Xeon E5520 quad-core

2.26 GHz processors and 48 GB RAM. We restricted the runtime to 600 seconds and the memory usage to 20 GB RAM.

Since algorithms **A-1** and **A-2** involve querying programs over preferences, we started by evaluating the different query techniques. For that, we executed **A-1** with query methods **Q-1** to **Q-4** on all selected instances, stopping after the first *solveQuery* call was finished. The performance of query techniques **Q-2**, **Q-3**, and **Q-4** was similar regarding runtime and only **Q-1** was clearly worse. We selected **Q-4** for the remaining experiments due to its slightly lower runtime. For more detailed tables, we refer to [20].

Next, we approximated four most diverse optimal models with methods **A-1** to **A-3**. We measured runtime and two quality measures. The first, called diversification quality [18], gives the sum of the Hamming distances among all pairs of solutions normalized to values between zero and one. The second is the minimum distance among all pairs of solutions of a set in percent. The solution set size of four was chosen because [22] claims that three solutions is the optimal amount for a user, and considering one additional solution provides further insight into the different quality measures. For all algorithms that do not use heuristics for diversification, we instead enabled heuristics preferring a negative sign for the atoms appearing in preference statements. This was observed in [7] to improve performance.

Table 1(a) provides in column T the average runtime and in column TO the sum of timeouts. The different methods are ordered by the number of timeouts. The best results in a column are shown in bold. We see that **A-3** is by far the fastest with 70 timeouts, solving 91% of the instances. Heuristic variations of **A-3** perform the best after that. Less invasive heuristics achieve similar runtimes with 113-118 timeouts. More sophisticated heuristics perform worse at 349-443 timeouts. In a range from 618 to 773 timeouts, non-heuristic methods solve the least instances by a significant margin. The results are in tune with the nature of the methods. Heuristics modifying the solving process for diversity decrease the performance in comparison with solving heuristics aimed at performance, but not as much as more complex methods involving preferences over optimal models.

In particular, non-heuristic methods show many timeouts. If we tried to analyze the quality of the solutions by assuming worst possible values for the instances that timed out, the results would be dominated by these instances. To avoid that, we calculated a score independent of the runtime. We considered all possible parings of the different methods. For each pair, we compared only instances where both found a solution set. The method with better quality value for the majority of instances receives a point. Finally, we ordered the subsequent tables according to that score.

In Table 1(b), for each method we see the score in column S, and the average of the diversification quality (over the instances solved by the method) in column *avg*. This way, we can examine the quality a method has achieved compared to other methods, and also the individual average quality. **A-1** has the best quality with a score of 15, followed by **A-2**-*dist-to*, **A-2**-*pg*, **A-3**-*pg-l* and **A-3**-*pg-l-rd*. All of those techniques regard the whole previous solution set to calculate the next solution and guide the solving strictly to diversity. **A-2**-*pg*, **A-3**-*pg-l* and **A-3**-*pg-l-rd* are also the first, second and third place, respectively, for average diversification quality. Next, with scores ranging from 10-7, we see **A-2** methods that do not take into account the whole previous set, or that were simply unable to find many solutions at all, as in the case of **A-2**-*dist*. Finally, we observe that **A-3** variations only regarding the last solution or no previous information perform worst in score and average. In these cases, the heuristic does not seem to be strong enough to steer the solving to high quality solution sets, and **A-3** uses no heuristic or optimization techniques to ensure diverse solutions.

In analogy to Table 1(b), Table 1(c) provides information for the minimum distance among the solutions. The best methods considering score and average minimum distance, viz. **A**-1, **A**-2-*dist-to*, **A**-3-*pg-l-rd*, **A**-3-*pg-l*, **A**-2-*pg*, utilize information from the whole previous solution set and have strict diversification techniques.

Overall, plain heuristic methods perform better in regards to runtime while more complex methods, depending on all previous solutions, lead to better quality. Furthermore, **A**-3-*pg-l-rd* and **A**-3-*pg-l* provide the best trade-off between performance and quality. While **A**-1, **A**-2-*dist-to* and **A**-2-*pg* achieve higher quality, they could solve only 18%, 16% and 13% of the instances. On the other hand, **A**-3-*pg-l-rd* and **A**-3-*pg-l* provide good diversification quality and minimum distance while solving 46% of the instances.

7 Discussion

We presented a comprehensive framework for computing diverse (or similar) solutions to logic programs with generic preferences and implemented it in *asprin* 2, available at [1]. To this end, we introduced a spectrum of different methods, among them, generalizations of existing work to the case of programs with general preferences. Hence, certain fragments of our framework provide implementations of the proposals in [9, 25]. While the latter had to resort to solver wrappers or even internal solver modifications, *asprin* heavily relies upon multi-shot solving that allows for an easy yet fine-grained control of reasoning processes. Moreover, we provided several generic building blocks, such as *maxmin* (and *minmax*) preferences, query-answering for programs with preferences, preferences among optimal models, and an automated approach for the guess and check methodology of [11], all of which are also of interest beyond diversification. Finally, we took advantage of the uniform setting offered by *asprin* 2 to conduct a comparative empirical analysis of the various methods for diversification. Generally speaking, there is a clear trade-off between performance and diversification quality, which allows for selecting the most appropriate method depending on the hardness of the application at hand.

References

1 asprin. `http://www.cs.uni-potsdam.de/asprin`.

2 *Proceedings of the Twelfth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'13)*. Springer, 2013.

3 B. Andres, M. Gebser, M. Glaß, C. Haubelt, F. Reimann, and T. Schaub. Symbolic system synthesis using answer set programming. In *Proceedings of the Twelfth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'13)* [2], pages 79–91.

4 M. Banbara, T. Soh, N. Tamura, K. Inoue, and T. Schaub. Answer set programming as a modeling language for course timetabling. *Theory and Practice of Logic Programming*, 13(4-5):783–798, 2013.

5 C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.

6 G. Brewka, J. Delgrande, J. Romero, and T. Schaub. asprin: Customizing answer set preferences without a headache. In *Proceedings of the Twenty-Ninth National Conference on Artificial Intelligence (AAAI'15)*, pages 1467–1474. AAAI Press, 2015.

7 G. Brewka, J. Delgrande, J. Romero, and T. Schaub. Implementing preferences with asprin. In *Proceedings of the Thirteenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'15)*, pages 158–172. Springer, 2015.

8 G. Brewka, I. Niemelä, and M. Truszczyński. Answer set optimization. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI'03)*, pages 867–872. Morgan Kaufmann, 2003.

9 T. Eiter, E. Erdem, H. Erdogan, and M. Fink. Finding similar/diverse solutions in answer set programming. *Theory and Practice of Logic Programming*, 13(3):303–359, 2013.

10 T. Eiter and G. Gottlob. On the computational cost of disjunctive logic programming. *Annals of Mathematics and Artificial Intelligence*, 15(3-4):289–323, 1995.

11 T. Eiter and A. Polleres. Towards automated integration of guess and check programs in answer set programming: a meta-interpreter and applications. *Theory and Practice of Logic Programming*, 6(1-2):23–60, 2006.

12 M. Gebser, C. Guziolowski, M. Ivanchev, T. Schaub, A. Siegel, S. Thiele, and P. Veber. Repair and prediction (under inconsistency) in large biological networks with answer set programming. In *Proceedings of the Twelfth International Conference on Principles of Knowledge Representation and Reasoning (KR'10)*, pages 497–507. AAAI Press, 2010.

13 M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. *Clingo* = ASP + control: Preliminary report. In *Technical Communications of the Thirtieth International Conference on Logic Programming (ICLP'14)*, volume 14 of *Theory and Practice of Logic Programming*, 2014.

14 M. Gebser, R. Kaminski, and T. Schaub. Complex optimization in answer set programming. *Theory and Practice of Logic Programming*, 11(4-5):821–839, 2011.

15 M. Gebser, B. Kaufmann, R. Otero, J. Romero, T. Schaub, and P. Wanko. Domain-specific heuristics in answer set programming. In *Proceedings of the Twenty-Seventh National Conference on Artificial Intelligence (AAAI'13)*, pages 350–356. AAAI Press, 2013.

16 M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.

17 E. Hebrard, B. Hnich, B. O'Sullivan, and T. Walsh. Finding diverse and similar solutions in constraint programming. In *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI'05)*, pages 372–377. AAAI Press, 2005.

18 A. Nadel. Generating diverse solutions in SAT. In *Proceedings of the Fourteenth International Conference on Theory and Applications of Satisfiability Testing (SAT'11)*, pages 287–301. Springer, 2011.

19 V. Pareto. *Cours d'economie politique*. Librairie Droz, 1964.

20 J. Romero, T. Schaub, and P. Wanko. Computing diverse optimal stable models (extended version). Available at `http://www.cs.uni-potsdam.de/wv/publications/`, 2016.

21 T. Schaub and S. Thiele. Metabolic network expansion with ASP. In *Proceedings of the Twenty-fifth International Conference on Logic Programming (ICLP'09)*, pages 312–326. Springer, 2009.

22 H. Shimazu. Expertclerk: Navigating shoppers' buying process with the combination of asking and proposing. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI'01)*, pages 1443–1448. Morgan Kaufmann, 2001.

23 S. Siddiqi. Computing minimum-cardinality diagnoses by model relaxation. In *Proceedings of the Twenty-second International Joint Conference on Artificial Intelligence (IJCAI'11)*, pages 1087–1092. IJCAI/AAAI Press, 2011.

24 P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1-2):181–234, 2002.

25 Y. Zhu and M. Truszczyński. On optimal solutions of answer set optimization problems. In *Proceedings of the Twelfth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'13)* [2], pages 556–568.

Answer Set Programming for Qualitative Spatio-Temporal Reasoning: Methods and Experiments

Christopher Brenton[1], Wolfgang Faber[2], and Sotiris Batsakis[3]

1 **School of Computing and Engineering, University of Huddersfield, Huddersfield, United Kingdom**
`christopher.brenton@hud.ac.uk`
2 **School of Computing and Engineering, University of Huddersfield, Huddersfield, United Kingdom**
`w.faber@hud.ac.uk`
3 **School of Computing and Engineering, University of Huddersfield, Huddersfield, United Kingdom**
`s.batsakis@hud.ac.uk`

Abstract

We study the translation of reasoning problems involving qualitative spatio-temporal calculi into answer set programming (ASP). We present various alternative transformations and provide a qualitative comparison among them. An implementation of these transformations is provided by a tool that transforms problem instances specified in the language of the Generic Qualitative Reasoner (GQR) into ASP problems. Finally, we report on an experimental analysis of solving consistency problems for Allen's Interval Algebra and the Region Connection Calculus with eight base relations (RCC-8).

1998 ACM Subject Classification D.1.6 Logic Programming

Keywords and phrases answer set programming, qualitative spatio-temporal reasoning

Digital Object Identifier 10.4230/OASIcs.ICLP.2016.4

1 Introduction

In this paper, we study the translation of reasoning problems involving qualitative spatio-temporal calculi into answer set programming (ASP). Qualitative spatio-temporal calculi were developed in order to deal with situations in which precise time-points or coordinates are not known. They rather deal with spatio-temporal regions and relationships that hold among them. Perhaps the best known of these are Allen's Interval Algebra [1] and the family of Region Connection Calculi (RCC) [6]. More recently, quite many of these calculi have been defined and described in a uniform way that allows for calculus-independent reasoning systems such as GQR [14]. Qualitative spatio-temporal calculi have a number of applications, for instance in planning, but also in Semantic Web applications inside GeoSPARQL [3].

This work has been conducted to lay the foundations for a larger project, in which the aim is to support expressing and reasoning with preferences over spatio-temporal relations, and also query answering and expressing defaults. We envision that the methods developed in this paper can be extended to accommodate preferences using the system `asprin`[1] [5].

[1] `http://www.cs.uni-potsdam.de/asprin/`

Recently, there has been another approach to spatio-temporal reasoning using ASP in [13]; however, in that work the focus is on combining quantitative and qualitative reasoning and it uses ASPMT as an underlying mechanism. The latter would make the integration of preferences more difficult.

In a previous work, Li [9] proposed a transformation of reasoning problems over qualitative spatio-temporal calculi into ASP. Li's description is by example, using RCC-8 (RCC with eight base relations), and does not discuss many alternatives. Moreover, a supporting tool seems to have been lost (J. Li, personal correspondence, November 2014). In this paper we elaborate on Li's results and propose a generically described family of transformations. The transformations differ in what kinds of ASP constructs they use and what representational assumptions are taken. Our longer term perspective is to endow qualitative spatio-temporal calculi with language constructs that allow for reasoning with incomplete knowledge, default assumptions, and preferences, which makes ASP an attractive language for supporting these.

All of these transformations are implemented in the tool `GQRtoASPConverter`, which accepts consistency problems over qualitative spatio-temporal calculi specified in the language of GQR, and produces logic programs that conform to the ASP-Core-2 standard. We also provide a simple nomenclature for the various transformations, so that they are easy to remember and identify.

Finally, we have conducted an experimental analysis of the various transformations on consistency problems over Allen's Interval Algebra and the RCC-8 calculus. While this paper is based on [4], it has been substantially revised and expanded. This paper describes additional transformations, has more formal definitions of the transformations and provides a proof for the main correctness theorem. A bug that was identified just before preparing the camera-ready version of [4] has been fixed, which yields a somewhat different picture in the experimental results, which have also been considerably extended. Our findings show that a number of encodings perform persistently well, and that several of them also outperform the direct encoding presented by Li. Unfortunately, the performance of special-purpose tools such as GQR appears to be out of reach using the techniques in this and [9]. Even so, the ASP transformations allow for a range of reasoning problems, for instance query answering, rather than for solving just consistency problems. They will in particular prove useful as a basis of our larger project, which will involve preferences and defaults.

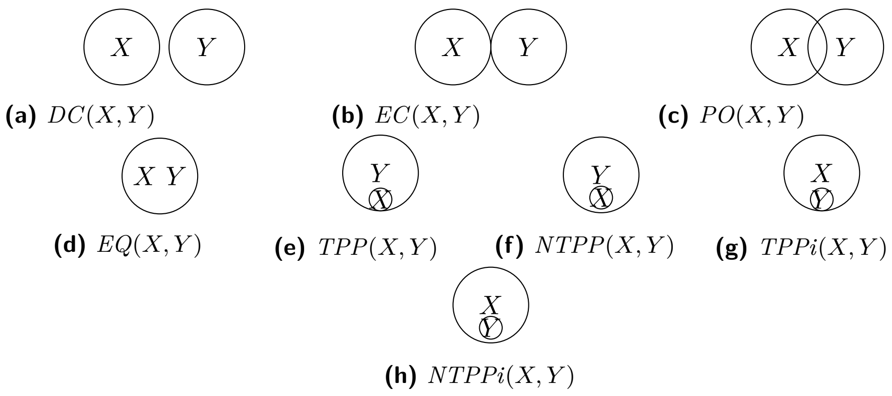
This work also provides an interesting set of new benchmark problems for ASP. In particular, some of the transformations create numerous disjunctive rules that can also be cyclic, which seems to trigger some suboptimal behaviour in current grounding algorithms.

2 Preliminaries

2.1 Qualitative Spatio-temporal Calculi

Temporal and spatial (e.g., topological) information often lacks precise values. For instance, in spatial reasoning, the exact location of an area might not be known. This calls for qualitative representations, which can be seen as abstractions of representations that involve precise values. Still, the relationships holding between such abstractly represented elements may be known. For example, the exact spatial location of "Europe" and "Italy" may not be known, while it is known that "Italy" is "inside" "Europe".

In temporal reasoning, the exact time frame in which an event occurs may not be known. Still, as with spatial reasoning, the relationships holding between events may be known. For example, it may not be known at what times breakfast and lunch were taken or at what time a newspaper was read, but it is known that breakfast occurred *before* lunch and may be the case that the newspaper was read *during* lunch.



(a) $DC(X, Y)$ **(b)** $EC(X, Y)$ **(c)** $PO(X, Y)$ **(d)** $EQ(X, Y)$ **(e)** $TPP(X, Y)$ **(f)** $NTPP(X, Y)$ **(g)** $TPPi(X, Y)$ **(h)** $NTPPi(X, Y)$

Figure 1 RCC-8 Base Relations.

More formally, a *qualitative (spatio-temporal) calculus* describes relations between elements of a set of elements $\mathcal{D}$ (possibly infinite). The set of base (or atomic) relations $\mathcal{B}$ is such that for each element in $\mathcal{D} \times \mathcal{D}$ exactly one base relation holds when complete information is available. Usually one is however confronted with a situation with incomplete or indefinite information, in which case for each element in $\mathcal{D} \times \mathcal{D}$ more than one definite base relation is known to possibly hold, but it is not known which one of these. In this case, we associate a set of base relations (those that possibly hold) to each pair of elements, formally one can do this by a labeling function $l : \mathcal{D}^2 \rightarrow 2^{\mathcal{B}}$. A set of base relations can be viewed as a disjunction of base relations, and the empty set represents inconsistency.

▶ **Definition 1.** Given a set of elements $\mathcal{D}$ (the domain) and a finite set of base relations $\mathcal{B}$, a (possibly partial) configuration is a labeling function $l : \mathcal{D}^2 \rightarrow 2^{\mathcal{B}}$. A configuration l is complete if $\forall(i, j) \in \mathcal{D}^2 : |l(i, j)| = 1$, in which case we can simplify the notation of the labeling function to $l : \mathcal{D}^2 \rightarrow \mathcal{B}$.

As an example, consider the Region Connection Calculus with eight base relations (RCC-8) [6], one of the main ways of representing topological relations. Figure 1 shows the intuitive meaning of the base relations (DC for disconnected, EC for externally connected, TPP for tangential proper part, NTPP for non-tangential proper part, PO for partially overlapping, EQ for equal, TPPi for tangential proper part inverse, and NTTPi for non-tangential proper part inverse). In our earlier example, the statement that Italy is inside Europe actually refers to a disjunction of base relations, as "inside" can refer to TPP or NTTP. So assuming $Italy \in \mathcal{D}$ and $Europe \in \mathcal{D}$, one would represent this as $Italy\{TPP, NTPP\}Europe$ or, in a more logic-oriented notation, $TPP(Italy, Europe) \vee NTTP(Italy, Europe)$.

As another example, consider Allen's Interval Algebra [1], one of the main ways of representing temporal information. Figure 2 shows an intuitive graphical representation of the base relations (b for before, m for meets, o for overlaps, d for during, s for starts, f for finishes, bi for before inverse, mi for meets inverse, oi for overlaps inverse, di for during inverse, si for starts inverse, fi for finishes inverse, and eq for equal). To continue with the previous temporal example stating that breakfast was taken before lunch and a newspaper was read during lunch, we can see that before refers to the b and during refers to d. If we assume for this example that $\{take\_breakfast, take\_lunch, read\_newspaper\} \subseteq \mathcal{D}$ then the relationships could be represented as $take\_breakfast\{b\}take\_lunch$ and $read\_newspaper\{d\}take\_lunch$ or, in a more logic-oriented notation, the two facts $b(take\_breakfast, take\_lunch)$. and $d(read\_newspaper, take\_lunch)$.

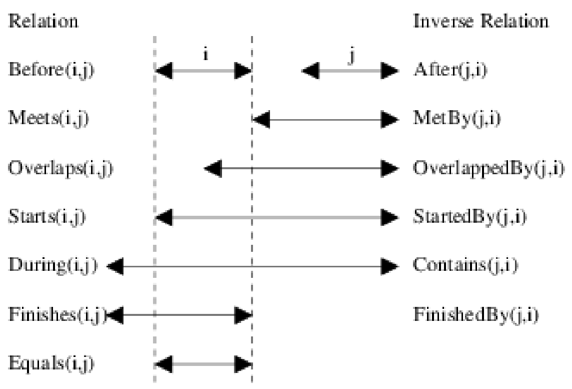


Figure 2 Allen's Interval Algebra Base Relations.

A qualitative (spatio-temporal) calculus will additionally identify which of the base relations is the equality relation (it is assumed to be present), which base relations are inverses of each other, and it will specify a composition table. The latter states for each pairs of base relations $\{R, S\} \subseteq \mathcal{B}$ and elements $\{X, Y, Z\} \subseteq \mathcal{D}$, if $R(X, Y)$ and $S(Y, Z)$ hold, which base relations possibly hold between X and Z. Formally, the composition table is a function $c : \mathcal{B}^2 \to 2^{\mathcal{B}}$.

▶ **Definition 2.** A qualitative (spatio-temporal) calculus (QSTC) is a tuple $\langle \mathcal{B}, e, i, c \rangle$ where B is a finite set of base relations, $e \in \mathcal{B}$ identifies the equality relation, $i : \mathcal{B} \to \mathcal{B}$ is a function that identifies the inverse for each base relation, and $c : \mathcal{B}^2 \to 2^{\mathcal{B}}$ is the composition table.

For RCC-8, EQ is the equality relation, EQ, DC, PO, and EC are inverses of themselves, while TPP is the inverse of TPPi, and NTTP is the inverse of NTTPi. For Allen's Interval Algebra, eq is the equality relation, and b, m, o, d, s, and f have inverse relations in bi, mi, oi, di, si, and fi respectively.

The most studied reasoning problem with qualitative calculi is the consistency problem, which asks whether a given configuration is consistent, that is, whether there is a complete subconfiguration (a *solution*) that is consistent with the composition table. This problem is known to be *NP-hard* in the general case, however tractable scenarios (i.e., solvable by polynomial time algorithms) have been identified [12]. There are also other, less studied, reasoning problems, such as asking whether a given relation holds between two given elements in some solution, or in all solutions.

▶ **Definition 3.** Given a QSTC $\mathcal{Q} = \langle \mathcal{B}, e, i, c \rangle$, a set of elements $\mathcal{D}$, and a configuration $l : \mathcal{D}^2 \to 2^{\mathcal{B}}$, a *solution* is a complete configuration $s : \mathcal{D}^2 \to \mathcal{B}$ such that $\forall (i, j) \in \mathcal{D}^2 : s(i, j) \in l(i, j)$, $\forall (i, j), (j, k) \in \mathcal{D}^2 : s(i, k) \in c(s(i, j), s(j, k))$, $\forall (i, j) \in \mathcal{D} : s(i, j) = i(s(j, i))$, and $\forall i \in \mathcal{D} : s(i, i) = e$. Let us denote the set of all solutions by $sol(\mathcal{Q}, \mathcal{D}, l)$. A configuration l over $\mathcal{D}$ is consistent with respect to a QSTC $\mathcal{Q}$ iff $sol(\mathcal{Q}, \mathcal{D}, l) \neq \emptyset$.

2.2 Answer Set Programming

The complete current ASP standard ASP-Core-2 is available at `https://www.mat.unical.it/aspcomp2013/ASPStandardization`. In the following, we present an overview of a subset of the ASP language used in the paper. For further background, we refer to [8, 2, 7]

A *predicate atom* is of the form $p(t_1, \ldots, t_n)$, where p is a *predicate name*, $t_1, \ldots, t_n$ are *terms* (constants or variables) and $n \geq 0$ is the *arity* of the predicate atom. A construct *not* a, where a is a predicate atom, is a *negation as failure (NAF) literal*. A *choice atom* is of the form $i\{a_1;\ldots;a_n\}j$ where $a_1,\ldots,a_n$ are predicate atoms and $n \geq 0$, $i \geq 0$, and $j \geq 0$. A literal is either a NAF literal or a choice atom. A *rule* is of the form

$$h_1 \mid \ldots \mid h_m \leftarrow b_1,\ldots,b_n.$$

where $h_1,\ldots,h_m$ are predicate or choice atoms (forming the rule's head) and $b_1,\ldots,b_n$ are literals (forming the rule's body) for $m \geq 0$ and $n \geq 0$. The rule is called an *integrity constraint* if $m = 0$, *fact* if $m = 1$ and $n = 0$, and *disjunctive fact* if $m > 1$ and $n = 0$. In facts and disjunctive facts, the $\leftarrow$ sign is usually removed for better readability. An ASP program is a set of rules.

Given a program P, the *Herbrand universe* of P consists of all constants that occur in P. The *Herbrand base* of P is the set of all predicate atoms that can be built by combining predicate names appearing in P with elements of the Herbrand universe of P. A (Herbrand) *interpretation* I for P is a subset of the Herbrand base of P, and contains all atoms interpreted as true by I. The *grounding* P^g of a program P is obtained by replacing the variables in each rule by all combinations of constants in the Herbrand universe and collecting all resulting rules. In the following, we will identify a program with its grounding. Given an interpretation I, a variable-free predicate atom a, $I \models a$ iff $a \in I$; for a NAF literal *not* a, $I \models$ *not* a iff $I \not\models a$; for a choice atom $i\{a_1,\ldots,a_n\}j$ iff $i \leq |\{a_k \mid I \models a_k, 0 \leq k \leq n\}| \leq j$. A rule is satisfied by I if for some head element h_i of the rule $I \models h$ whenever $I \models b_j$ for all body elements b_j. A program is satisfied by I iff all rules are satisfied by I. A satisfying interpretation is also called a model of the program. A model M of a program P is a minimal model, if no $N \subset M$ satisfies P. The *reduct* of a program with respect to an interpretation consists of those rules for which $I \models b_j$ for all body elements b_j. An interpretation I is an *answer set* of P if I is a minimal model of the reduct P^I. Let $AS(P)$ denote the set of all answer sets of program P.

3 Transformations of Qualitative Spatio-temporal Calculi to Answer-set Programming

Given the specification of a qualitative calculus, there are various ways to create an ASP program such that, together with a suitable representation of an input labeling, each answer set corresponds to one solution. Some first transformations of this kind were presented in [9]. In this section, we present a different and more systematic approach. Throughout the section we assume a domain $\mathcal{D}$, a configuration l and a QSTC $\langle \mathcal{B}, e, i, c\rangle$ to be given.

3.1 Representing Base Relations and Domain

To start with, each element of the domain will give rise to a fact.

▶ **Definition 4.** Given the domain $\mathcal{D}$, we will generate a fact

$$element(x). \qquad (1)$$

for each $x \in \mathcal{D}$.

For each base relation $r \in \mathcal{B}$ we will use a predicate of arity 2 for its ASP representation. This is different to [9], in which a single predicate *label* of arity 3 was used.

The simplest and most natural representation is to use one predicate for each base relation. For example, for RCC-8 we would consider eight predicates *dc*, *ec*, *po*, *eq*, *tpp*, *ntpp*, *tppi*, *ntppi*. The fact that two elements x and y are labeled by the base relation $TPPi$ would then be represented by the atom $tppi(x, y)$.

There is, however, a slight redundancy in this representation. For each pair of distinct inverse relations r and s, whenever $r(x, y)$ holds, it is clear that $s(y, x)$ also holds and $r(y, x)$ and $s(x, y)$ do not hold. We could use a single predicate for the pair of distinct inverse relations instead. For example for the inverse relations TPP and $TPPi$ of RCC-8, we could use the single predicate *tpp*, and the fact that two elements x and y are labeled by the base relation $TPPi$ would then be represented by the atom $tpp(y, x)$.

Since these two approaches differ in how they deal with pairs of distinct inverse relations, we refer to the first approach as *two-predicates-per-pair* and the second one as *one-predicate-per-pair*.

3.2 Representing the Search Space

The next issue to decide on is how to represent the search space (by representing all possible labelings). Let us assume that we use one of the representation methods described in Section 3.1, denoting by $\overline{r}(X, Y)$ the atom representing the fact that X and Y are labeled by r.

We will provide two encodings, which we will refer to as disjunctive and choice encodings. For the disjunctive encoding, we use a disjunctive rule together with a number of integrity constraints, ensuring that at most one of the base relations can hold between a pair of elements, and an auxiliary rule to handle the easy case of pairs of equal elements.

▶ **Definition 5** (Disjunctive Encoding). If $\mathcal{B} = \{r_1, \ldots, r_n\}$, the disjunctive encoding includes the disjunctive rule

$$\overline{r_1}(X, Y) \mid \ldots \mid \overline{r_n}(X, Y) \leftarrow element(X), element(Y), X \mathrel{!=} Y. \tag{2}$$

where $X \mathrel{!=} Y$ is a built-in predicate stating that X is distinct from Y. Moreover, for each pair of base relations $\{r, s\} \subseteq \mathcal{B}$ an integrity constraint

$$\leftarrow \overline{r}(X, Y), \overline{s}(X, Y). \tag{3}$$

is added.

Finally, a single rule

$$\overline{r_e}(X, X) \leftarrow element(X). \tag{4}$$

is added in order to deal with the equality relation e on equal elements (note that the equality relation can additionally also hold for two different elements.

For example, for RCC-8 and the one-predicate-per-pair approach, the disjunctive encoding results in

$$\begin{aligned}
&dc(X, Y) \mid ec(X, Y) \mid po(X, Y) \mid eq(X, Y) \mid tpp(X, Y) \mid ntpp(X, Y) \\
&\qquad \mid tppi(X, Y) \mid ntppi(X, Y) \leftarrow element(X), element(Y), X \mathrel{!=} Y. \\
&\leftarrow dc(X, Y), ec(X, Y). \quad \ldots \quad \leftarrow tppi(X, Y), ntppi(X, Y). \\
&eq(X, X) \leftarrow element(X).
\end{aligned}$$

There are 56 integrity constraints in this encoding.

Alternatively, one can equivalently state the same using a rule with a choice atom, arriving at the choice encoding.

▶ **Definition 6** (Choice Encoding). For $\mathcal{B} = \{r_1, \ldots, r_n\}$, the choice encoding contains the rule

$$1\{\overline{r_1}(X,Y); \ldots; \overline{r_n}(X,Y)\}1 \leftarrow element(X), element(Y), X\ != \ Y. \tag{5}$$

It also contains rule (4) for dealing with the equality relation e.

For example, for RCC-8 and the two-predicates-per-pair approach, the choice encoding results in

$$\begin{aligned}&1\{dc(X,Y); ec(X,Y); po(X,Y); eq(X,Y); tpp(X,Y); ntpp(X,Y);\\ &\quad tppi(X,Y); ntppi(X,Y)\}1 \leftarrow element(X), element(Y), X\ != \ Y.\end{aligned}$$

Only if the two-predicates-per-pair approach is taken, one can replace $X\ != \ Y$ by $X < Y$. We will refer to this as the *antisymmetric optimisation*. The idea is to avoid representing one inverse relation, for instance instead of having both $tpp(1,2)$ and $tppi(2,1)$ in the choice, this optimisation causes only $tpp(1,2)$ to be in the choice. However, the inverse relations still need to be derived, so other rules are needed to achieve this.

▶ **Definition 7** (Disjunctive Encoding with Antisymmetric Optimisation). For $\mathcal{B} = \{r_1, \ldots, r_n\}$, the disjunctive encoding with antisymmetric optimisation includes the disjunctive rule

$$\overline{r_1}(X,Y) \mid \ldots \mid \overline{r_n}(X,Y) \leftarrow element(X), element(Y), X < Y. \tag{6}$$

and for each pair of inverse relations ri and r (that is, $ri, r \in \mathcal{B} : i(r) = ri$)

$$\overline{ri}(X,Y) \leftarrow \overline{r}(Y,X), Y < X. \tag{7}$$

together with constraints (3) and rule (4).

▶ **Definition 8** (Choice Encoding with Antisymmetric Optimisation). For $\mathcal{B} = \{r_1, \ldots, r_n\}$, the choice encoding with antisymmetric optimisation contains the rule

$$1\{\overline{r_1}(X,Y); \ldots; \overline{r_n}(X,Y)\}1 \leftarrow element(X), element(Y), X < Y. \tag{8}$$

rules (7) and rule (4).

We would like to point out that the encodings in [9] have an analogue of the antisymmetric optimisation, but fail to include rules (7), resulting in correctness issues.

3.3 Representing the Composition Table

As described in Section 2.1, the function c represents the composition table of the calculus. For each pair of relations r, s in $\mathcal{B}$, we will create a number of constructs unless $c(r,s) = \mathcal{B}$. The constructs created will depend on the chosen approach, as described below.

The first approach, which we will refer to as the *rule encoding*, creates one disjunctive rule for each pair of relations, an immediate way of representing the composition table.

▶ **Definition 9** (Rule Encoding). For all $r, s \in \mathcal{B}$ such that $c(r,s) = \{r_1, \ldots, r_n\} \neq \mathcal{B}$, the rule encoding contains

$$\overline{r_1}(X,Z) \mid \ldots \mid \overline{r_n}(X,Z) \leftarrow \overline{r}(X,Y), \overline{s}(Y,Z). \tag{9}$$

For RCC-8, the composition of TPP and EC (resulting in DC or EC) is translated to the following rule encoding:

$$dc(X,Z) \mid ec(X,Z) \leftarrow tpp(X,Y), ec(Y,Z).$$

The second approach, which we will refer to as *integrity constraint encoding*, creates the rule (9) only if $n = 1$; in all other cases integrity constraints are created instead. This amounts to representing which relations must not hold in the composition.

▶ **Definition 10** (Integrity Constraint Encoding)**.** For all $r, s \in \mathcal{B}$ such that $\mathcal{B} \setminus c(r,s) = \{s_1, \ldots, s_k\}$ and $1 < |c(r,s)| < |\mathcal{B}|$, the integrity constraint encoding contains

$$\leftarrow \overline{s_1}(X,Z), \overline{r}(X,Y), \overline{s}(Y,Z). \quad \ldots \quad \leftarrow \overline{s_k}(X,Z), \overline{r}(X,Y), \overline{s}(Y,Z). \tag{10}$$

and if $c(r,s) = \{r_1\}$ then it contains

$$\overline{r_1}(X,Z) \leftarrow \overline{r}(X,Y), \overline{s}(Y,Z). \tag{11}$$

For RCC-8, the composition of TPP and EC (resulting in DC or EC) is translated to the following integrity constraint encoding (assuming the two-predicates-per-pair approach):

$$\begin{array}{ll}
\leftarrow po(X,Z), tpp(X,Y), ec(Y,Z). & \leftarrow eq(X,Z), tpp(X,Y), ec(Y,Z). \\
\leftarrow tpp(X,Z), tpp(X,Y), ec(Y,Z). & \leftarrow ntpp(X,Z), tpp(X,Y), ec(Y,Z). \\
\leftarrow tppi(X,Z), tpp(X,Y), ec(Y,Z). & \leftarrow ntppi(X,Z), tpp(X,Y), ec(Y,Z).
\end{array}$$

Rule and integrity constraints encodings can be mixed, we consider imposing a limit n for $|c(r,s)|$ up to which rules will be created, and beyond which integrity constraints will be created.

▶ **Definition 11** (Integrity Constraint Beyond n Encoding)**.** For a fixed $n < |\mathcal{B}|$ and all $r, s \in \mathcal{B}$ such that $\mathcal{B} \setminus c(r,s) = \{s_1, \ldots, s_k\}$ and $|c(r,s)| > n$, the integrity constraint beyond n encoding contains integrity constraints (10) and if $c(r,s) = \{r_1, \ldots, r_k\}$ with $k \leq n$ then it contains rule (9).

3.4 Representing the Input

As described in Definition 1 in Section 2.1, the input is a partial configuration (or labeling function) l over pairs of elements. Assuming $l(a,b) = \{r_1, \ldots, r_n\}$ for $\{a,b\} \subseteq \mathcal{D}$, we note that the signature of the labeling function is identical to the composition table function. Therefore, we follow the same approach as for representing the composition table, depending on whether the rule, integrity constraint, or integrity constraint beyond n encoding is employed. If $l(a,b) = \mathcal{B}$, nothing will be created in any of the approaches.

▶ **Definition 12** (Rule Input Encoding)**.** For all $a, b \in \mathcal{D}$ such that $l(a,b) = \{r_1, \ldots, r_n\} \neq \mathcal{B}$, the rule input encoding contains

$$\overline{r_1}(a,b) \mid \ldots \mid \overline{r_n}(a,b). \tag{12}$$

▶ **Definition 13** (Integrity Constraint Input Encoding)**.** For all $a, b \in \mathcal{D}$ such that $\mathcal{B} \setminus l(a,b) = \{s_1, \ldots, s_k\}$ and $1 < |l(a,b)| < |\mathcal{B}|$, the integrity constraint input encoding contains

$$\leftarrow \overline{s_1}(a,b). \quad \cdots \quad \leftarrow \overline{s_k}(a,b). \tag{13}$$

and if $l(a,b) = \{r_1\}$ then it contains

$$\overline{r_1}(a,b). \tag{14}$$

▶ **Definition 14** (Integrity Constraint Beyond n Input Encoding)**.** For all $a, b \in \mathcal{D}$, a fixed $n < |\mathcal{B}|$ and all $r, s \in \mathcal{B}$ such that $\mathcal{B} \backslash l(a,b) = \{s_1, \ldots, s_k\}$ and $|l(a,b)| > n$, the integrity constraint beyond n input encoding contains integrity constraints (10) and if $l(a,b) = \{r_1, \ldots, r_k\}$ with $k \leq n$ then it contains rule (9).

As an example, consider two regions *italy* and *europe* in the context of RCC-8, and assume that we know that TPP or $NTPP$ holds between *italy* and *europe* (i.e., $l(\mathit{italy}, \mathit{europe}) = \{TPP, NTPP\}$). The rule encoding will create one disjunctive fact

$$\mathit{tpp}(\mathit{italy}, \mathit{europe}) \mid \mathit{ntpp}(\mathit{italy}, \mathit{europe}).$$

whereas the integrity constraint encoding (assuming the one-predicate-per-pair approach) yields

$$\begin{array}{lll} \leftarrow dc(\mathit{italy}, \mathit{europe}). & \leftarrow ec(\mathit{italy}, \mathit{europe}). & \leftarrow po(\mathit{italy}, \mathit{europe}). \\ \leftarrow eq(\mathit{italy}, \mathit{europe}). & \leftarrow tppi(\mathit{italy}, \mathit{europe}). & \leftarrow ntppi(\mathit{italy}, \mathit{europe}). \end{array}$$

▶ **Theorem 15.** *Given a qualitative calculus $\mathcal{Q} = \langle \mathcal{B}, e, i, c \rangle$, a set of elements $\mathcal{D}$, and a configuration $l : \mathcal{D}^2 \to 2^{\mathcal{B}}$, let P be the ASP program generated by a transformation obtained by any admissible combination of options and optimisations presented in this section. There is a one-to-one correspondence between $sol(\mathcal{Q}, \mathcal{D}, l)$ and $AS(P)$.*

4 Proof of Theorem 15

Proof. We will first show that for each $s \in sol(\mathcal{Q}, \mathcal{D}, l)$, $At(s) = \{\mathit{element}(x) \mid x \in \mathcal{D}\} \cup \{b(i,j) \mid (i,j) \in \mathcal{D}^2, s(i,j) = b \in \mathcal{B}\} \in AS(P)$ if the two-predicate-per-pair approach is chosen, and $Ao(s) = \{\mathit{element}(x) \mid x \in \mathcal{D}\} \cup \{\overline{b}(i,j) \mid (i,j) \in \mathcal{D}^2, s(i,j) = b \in \mathcal{B}\} \in AS(P)$ if the one-predicate-per-pair approach is chosen and $\overline{b} = i(b)$ if $i(b)$ represents b in the encoding, and $\overline{b} = b$ otherwise.

Rules (1) are trivially satisfied by $\forall (i,j) \in \mathcal{D}^2 : s(i,j) \in l(i,j)$. Rules (2) are satisfied because s is a function, hence for each $(i,j) \in \mathcal{D}^2$ where $i \neq j$ the body of the corresponding ground rule is satisfied and exactly one of the head atoms is satisfied in $At(s)$ (resp. $Ao(s)$). For the same reason, constraints (3) are satisfied, too. Finally, rules (4) because $\forall i \in \mathcal{D} : s(i,i) = e$ holds. From the observation for (5) it immediately follows that (5) is satisfied as well. The ground instantiations of (6) with a true body are a subset of those of (5), and by the observation above are satisfied as well, similar for (5) and (8). Rules (7) are satisfied since $\forall (i,j) \in \mathcal{D} : s(i,j) = i(s(j,i))$ holds.

Next, rules (9) are satisfied by both $At(s)$ and $Ao(s)$ because $\forall (i,j), (j,k) \in \mathcal{D}^2 : s(i,k) \in c(s(i,j), s(j,k))$ and we observe that exactly one head atom is true whenever the body holds with respect to $At(s)$ (resp. $Ao(s)$), which also shows satisfaction of (11). Integrity constraints (10) hold because $\forall (i,j), (j,k) \in \mathcal{D}^2 : s(i,k) \in c(s(i,j), s(j,k))$ implies $\forall (i,j), (j,k) \in \mathcal{D}^2 : s(i,k) \notin \mathcal{B} \setminus c(s(i,j), s(j,k))$.

Finally, rules (12) are satisfied by $At(s)$ and $Ao(s)$ because $\forall (i,j) \in \mathcal{D}^2 : s(i,j) \in l(i,j)$ holds. Note that exactly one of the disjuncts is true. Constraints (13) are satisfied because $\forall (i,j) \in \mathcal{D}^2 : s(i,j) \in l(i,j)$ implies $\forall (i,j) \in \mathcal{D}^2 : s(i,j) \notin \mathcal{B} \setminus l(i,j)$.

If the antisymmetric optimisation is not employed, $At(s)$ is a minimal model of the reduct of P since any subset of $At(s)$ does not satisfy one of the rules (1) or one of the ground instances of (4) and either (2) (recall that $At(s)$ satisfies exactly one head atom for each of these) or (5), all of which are present in the reduct. The same reasoning shows that $Ao(s)$ is a minimal model of the reduct if the one-predicate-per-pair approach gave rise to P.

If the antisymmetric optimisation is employed, $At(s)$ is a minimal model of the reduct of P since any subset of $At(s)$ does not satisfy one of the rules (1) or one of the ground instances of (4) and either (6) (again, recall that $At(s)$ satisfies exactly one head atom for each of these), or (8), or (7), all of which are present in the reduct. The same reasoning shows that $Ao(s)$ is a minimal model of the reduct if the one-predicate-per-pair approach gave rise to P.

Now let us assume that $A \in AS(P)$. We will show that $sT(A) : \mathcal{D}^2 \rightarrow \mathcal{B}$ (for the two-predicate-per-pair approach) or $sO(A) : \mathcal{D}^2 \rightarrow \mathcal{B}$ (for the one-predicate-per-pair approach) are in $sol(\mathcal{Q}, \mathcal{D}, l)$, where the functions are defined as follows for all $(x, y) \in \mathcal{D}^2$: $sT(A)(x, y) = b$ if $b(x, y) \in A$; $sO(A)(x, y) = i(b)$ if $b(x, y) \in A$ and b was used to represent $i(b)$, $sO(A)(x, y) = b$ if $b(x, y) \in A$ otherwise.

First of all, we observe that the functions are well-defined because if the antisymmetric optimisation is not employed, the ground instances of (4) and (2) or (5) require at least one $b(x, y) \in A$ for some $b \in \mathcal{B}$ for each $(x, y) \in \mathcal{D}^2$. If the antisymmetric optimisation is employed, then the rules (4) and one of (6) or (8), and (7) also require at least one $b(x, y) \in A$ for some $b \in \mathcal{B}$ for each $(x, y) \in \mathcal{D}^2$. Moreover, for each $(x, y) \in \mathcal{D}^2$ $b(x, y) \in A$ holds for exactly one $b \in \mathcal{B}$ because of either (3), (5), or (6).

It holds that $\forall (i, j) \in \mathcal{D}^2 : sT(A)(i, j) \in l(i, j)$ (resp. $\forall (i, j) \in \mathcal{D}^2 : sO(A)(i, j) \in l(i, j)$ because rules (12) or integrity constraints (13) would otherwise not be satisfied by A.

We have $\forall (i, j), (j, k) \in \mathcal{D}^2 : sT(A)(i, k) \in c(sT(A)(i, j), sT(A)(j, k))$ (resp. $\forall (i, j), (j, k) \in \mathcal{D}^2 : sO(A)(i, k) \in c(sO(A)(i, j), sO(A)(j, k)))$ as otherwise rules (9) or integrity constraints (10) would not be satisfied by A.

Also, $\forall i \in \mathcal{D} : sT(A)(i, i) = e$ and $\forall i \in \mathcal{D} : sO(A)(i, i) = e$ trivially hold because of rules (4).

Finally, we can see $\forall (i, j) \in \mathcal{D} : sT(A)(i, j) = i(sT(A)(j, i))$ and $\forall (i, j) \in \mathcal{D} : sO(A)(i, j) = i(sO(A)(j, i))$ because the composition table needs to contain $c(sT(A)(i, j), e) = i(sT(A)(j, i))$. Then, because of the arguments in the previous two paragraphs, $\forall (i, j) \in \mathcal{D} : sT(A)(i, j) = i(sT(A)(j, i))$ holds. ◀

5 Implementation of Transformations

The transformation tool `GQRtoASPConverter`[2] is a command line tool implemented using Java 1.7 and JavaCC version 5.0. Its calculi and input definitions are in the syntax of GQR[3] [14].

The tool defines a grammar from which it is possible to construct a number of abstract syntax trees over the composition file and input file provided. Using the transformation specified, it is possible to parse these abstract syntax trees using the visitor option within JavaCC and rebuild them according to the techniques described employed within. It implements all transformations obtained by combining the various options described in Section 3. The transformations were designed in a modular fashion and we will refer to them using a three letter nomenclature. Each letter represents how each module was implemented.

The first module denotes how the search space is opened, either using the disjunctive encoding D or the choice encoding C. The second module denotes how to encode pairs of inverse base relations. T refers to the two-predicate-per-pair approach, while O refers to the one-predicate-per-pair approach. The third module denotes how composition tables and the

[2] Available at `https://github.com/ChrisBrenton/GQRtoASPConverter`.
[3] `http://sfbtr8.informatik.uni-freiburg.de/r4logospace/Tools/gqr.html`

input are represented. R is used to refer to the rule encoding, while I refers to the integrity constraint encoding.

As an example, *CTI* uses the choice encoding, the two-predicate-per-pair approach, and the integrity constraint encoding. In total, the following are available: *DTR*, *CTR*, *DOR*, *COR*, *DTI*, *CTI*, *DOI*, *COI*.

The modifier A is added to the end of the name if the antisymmetric optimisation mentioned at the end of Section 3.2 is employed. This optimisation is present in all encodings where the two-predicate-per-pair approach is employed as described in Section 3.1, thus the following are available: *CTIA*, *DTIA*, *CTRA*, *DTRA*.

Transformations that implement the integrity constraint encoding, as defined in definition 10 are extended to produce rules as defined in definition 11. These transformations are denoted by the presence of a number at the end of their transformation name. Currently, values of n between 1 and 7 inclusive are supported, where the lack of a number present in the name indicates $n = 1$. By example using the *CTI* family of transformations, a rule where 3 possible relations may hold would be transformed into integrity constraints by *CTI* and *CTI2*, and into rules with disjunctive heads by *CTI3*, *CTI4*, *CTI5*, *CTI6*, and *CTI7*. This is achieved by counting the number of child nodes present in an abstract syntax tree at the node representing the disjunction of possible relations, and producing a disjunctive rule or a number of integrity constraints accordingly.

The tool can also produce the "direct encoding" of [9], which is similar to *CTIA*, but uses a different encoding of base relations, and it is slightly different from the integrity constraint encoding, as it creates integrity constraints also if $|c(r, s)| = 1$ or $|l(a, b)| = 1$, while our approach would creates a single rule in these cases. We will in the following refer to this encoding as *LiDir*.

In total, this tool can produce 49 encodings.

`GQRtoASPConverter` can be used by running a command of the following structure:

```
java GqrCalculusParser [switch] [GQR spec] [GQR problem] [outputdir]
```

Here, the switch is the three/four letter abbreviation of the desired transformation in lower case characters, prefixed with a hyphen, such as -cti3 or -doi. The GQR spec file is a meta-file describing a calculus. It contains an identification of the equality relation, the size of the calculus, and references to two other files. These are one file that describes the composition table of a calculus as described in Section 2.1, and a converse file that describes the inverses of relations. The problem file is the file that should be translated by the tool, and should be in the format accepted by GQR. The outputdir is a folder in which the tool should put all translations.

By example, using -ctia4 as the switch, ~/Documents/GQR/gqr-1500/data/rcc8.spec as the GQR specification file, ~/Documents/GQR/gqr-1500/data/rcc8/csp/example-10x10.csp as the GQR problem file, and ~/Documents/RCC8/Example10x10 as the output directory will produce transformations according to the techniques employed within *CTIA4*.

6 Experimental Evaluation

Experiments involving consistency problems over the Region Connection Calculus with eight base relations (RCC-8) [6] and Allen's Interval Algebra [1] were carried out. Both calculi are widely used, and many of their properties have been investigated.

In the first set of experiments, `GQRtoASPConverter` was used with the problem files

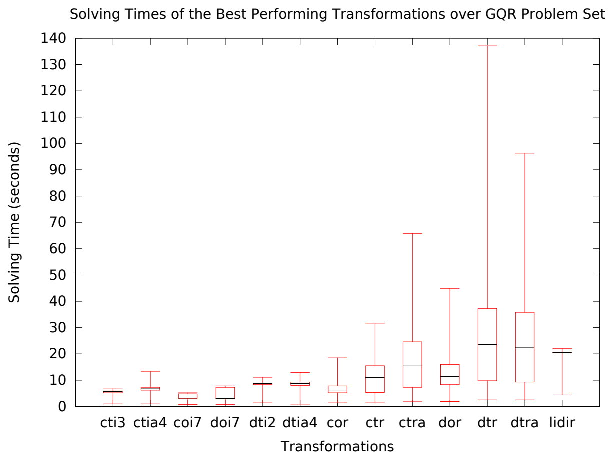


Figure 3 Results of the best performing transformation from each family on GQR RCC-8 problems.

provided with GQR version 1500 [4], the output of which were then solved using the ASP solver clingo [7] by means of the Pyrunner benchmarking tool[5]. Times given include the entire clingo process, from executing the command to receiving an output. Benchmarks were performed on an Intel® Core$^{\text{TM}}$ i7-4790 CPU @ 3.60GHz $\times$ 8 processor machine with a 300 second time out with 4GB memory available and using clingo version 4.4.0[6].

Results presented in this section will make use of box-and-whisker plots, where the whiskers represent maximum and minimum values for each transformation, and the boxes represent the interquartile range. The horizontal bar found within the boxes are used to represent the median time taken for each transformation.

Figure 3 shows the best performing transformation of each family of transformations over the RCC-8 problem set provided with GQR. Over this problem set, the *COI*, *DOI*, *CTI*, *CTIA*, *DTI*, and *DTIA* families of transformations hold the better performing transformations according to the maximum time taken to solve. A consistency in these families of transformations is that they all make use of the Integrity Constraint Beyond n Encoding as described in Definition 11. A more complete picture for all values of n is provided online[7], to show how transformations of the better performing families compare over the set of problems provided with GQR over RCC-8. This trend was also identified with the set of problems for Allen's Interval Algebra provided by GQR, the best performing of which are shown in figure 4, though variance exists on the value of n. Also, important to note is that no problem with a domain size greater than 20 was solved within the time and memory limits set for all transformations; all graphs provided only show problems that successfully solved.

For the second set of benchmarks, qualitative spatio-temporal constraint networks were randomly generated according to the algorithm described in [11]. Networks were generated with domain sizes ranging from 20 to 50 in increments of 10. In order to fall within the phase transition region for RCC-8, where all base relations are available, networks were generated with an average degree for each element varying between 8 and 10 in increments of

[4] `http://sfbtr8.informatik.uni-freiburg.de/r4logospace/Tools/gqr.html`
[5] `https://github.com/alviano/python`
[6] `http://sourceforge.net/projects/potassco/files/clingo/4.4.0/`
[7] `https://selene.hud.ac.uk/chrisbrenton/aspforqstr.php`

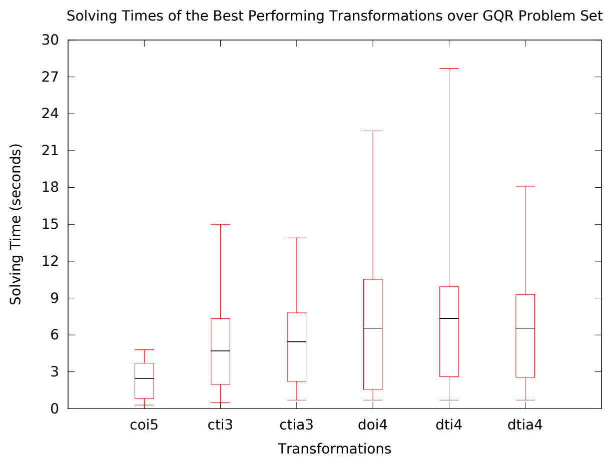


Figure 4 Results of the best performing transformations on GQR Allen problems.

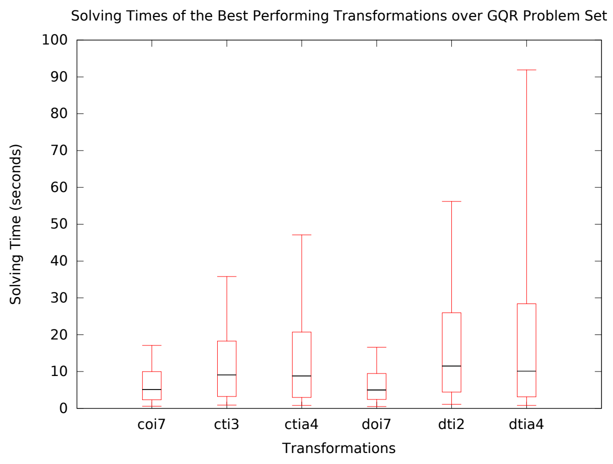


Figure 5 Results of the best performing transformations on randomly generated RCC-8 problems.

0.5 with an average label size of 4. For each combination of domain size and average degree, 20 networks were generated. Networks were also generated for Allen's Interval Algebra with domain sizes also ranging between 20 and 50 in increments of 10, with an average degree between 5 and 8 and an average label size of 6.5 in order to fall within the phase transition region for the calculus.

Figure 5 shows how the best performing transformations performed over the set of randomly generated networks for RCC-8. Notable is that the same family of transformations again prove to perform the most efficiently with respect to time taken to solve, though again the value of n does vary. All problems of all network sizes and degrees were solved within the set time and memory limits for RCC-8.

In the set of generated problems over Allen's Interval Algebra, all transformations solved all problems with a domain size of 20 within the set time and memory limits. In problems with domain size 30, *CTIA*, *DTI7*, and *DTRA* failed to solve one problem. *DOR* and *DTR* failed to solve 5 problems. *LiDir* failed to solve all problems within the set time and memory limits.

In the set of generated problems over Allen's Interval Algebra with domain size 40, only *COI5*, *COI6*, and *COI7* managed to solve all problems within the set time and memory limits.

In the set of generated problems over Allen's Interval Algebra with domain size 50, none of the transformations solved all problems, with *COI7* and *DOI7* solving the most at 45 out of 70.

GQR was also run over the set of generated problems for comparison; it is significantly faster than any of our approaches taking less than one second on all problems. However, we would like to note that while GQR is optimised for deciding consistency problems, it is limited to providing one solution. Answer set programming systems usually do not have this limitation, which will prove to beneficial in future work, and readily support query answering.

The generated problem sets for both RCC-8 and Allen's Interval Algebra are available online.[8]

In summary, we observed that the *COI7* encoding is the best performing one for the tested benchmark set. It also significantly outperforms the *LiDir* encoding. While not as performant as GQR, it provides the necessary flexibility for our future work that GQR does not offer.

7 Conclusion and Future Work

In this work a systematic approach to transforming qualitative spatio-temporal calculi and reasoning problems was developed. A number of options were identified that differ in representational issues and make use of different constructs of ASP. These were implemented in `GQRtoASPConverter`, which also supports a transformation previously suggested by Li [9]. An extensive set of benchmarks was run in order to identify the best-performing transformation or family of transformations. This turned out to be the *COI* family of transformations, particularly the *COI7* encoding.

While not discussed at length in this paper, also the transformations that turned out to be computationally inferior provided interesting insights. For instance, for many encodings involving numerous disjunctions, the grounders of the tested solvers (DLV and clingo) appear to create by far more ground rules than would be necessary. This can be observed in particular when creating problems that are easy (deterministic) to solve. One avenue for future work would be analysing whether grounding methods could be improved to deal with these kinds of programs (numerous disjunctions, possibly with cycles) in better ways.

There are also further options in the transformations that would be worth looking into. For instance, one could also translate the input into choice rules rather than disjunctive rules. Also completely different methods, such as using hybrid ASP and CSP solvers appear promising.

Also, we would like to enlarge the set of calculi considered for benchmarks, which in this paper were limited to RCC-8 and Allen's interval algebra. While these calculi appear to be the best-studied, also others, such as $\mathcal{OPRA}_m$ [10], could yield interesting benchmark problems.

Finally, at the moment only consistency problems were benchmarked. One of the potential advantages of using ASP in these domains is that also other problems such as query answering could be easily supported. Therefore, experimentally testing ASP on these problems would

[8] `https://selene.hud.ac.uk/chrisbrenton/solving-qstr.php`

be particularly interesting. As a further step, we would extend the methods developed in this paper and extend them to support preferences and defaults.

References

1 James F. Allen. An interval-based representation of temporal knowledge. In Patrick J. Hayes, editor, *Proceedings of the 7th International Joint Conference on Artificial Intelligence (IJCAI'81), Vancouver, BC, Canada, August 1981*, pages 221–226. William Kaufmann, 1981.

2 Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.

3 Robert Battle and Dave Kolas. Enabling the geospatial semantic web with parliament and geosparql. *Semantic Web*, 3(4):355–370, 2012.

4 Christopher Brenton, Wolfgang Faber, and Sotiris Batsakis. Solving qualitative spatio-temporal reasoning problems by means of answer set programming: Methods and experiments. In *Proceedings of the Eighth Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP 2015)*, Cork, Ireland, 2015.

5 Gerhard Brewka, James P. Delgrande, Javier Romero, and Torsten Schaub. Implementing preferences with asprin. In Francesco Calimeri, Giovambattista Ianni, and Miroslaw Truszczynski, editors, *Logic Programming and Nonmonotonic Reasoning – 13th International Conference (LPNMR 2015)*, volume 9345 of *Lecture Notes in Computer Science*, pages 158–172. Springer, 2015.

6 Anthony G. Cohn, Brandon Bennett, John Gooday, and Nicholas Mark Gotts. Qualitative spatial representation and reasoning with the region connection calculus. *GeoInformatica*, 1(3):275–316, 1997.

7 Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan and Claypool Publishers, 2012.

8 Michael Gelfond and Vladimir Lifschitz. The Stable Model Semantics for Logic Programming. In *Logic Programming: Proceedings Fifth Intl Conference and Symposium*, pages 1070–1080, Cambridge, Mass., 1988. MIT Press.

9 Jason Jingshi Li. Qualitative spatial and temporal reasoning with answer set programming. In *IEEE 24th International Conference on Tools with Artificial Intelligence, ICTAI 2012*, pages 603–609, 2012.

10 Till Mossakowski and Reinhard Moratz. Qualitative reasoning about relative direction of oriented points. *Artificial Intelligence*, 180–181:34–45, 2012.

11 Jochen Renz and Bernhard Nebel. Efficient methods for qualitative spatial reasoning. *Journal of Artificial Intelligence Research*, 15:289–318, 2001.

12 Jochen Renz and Bernhard Nebel. Qualitative spatial reasoning using constraint calculi. In Marco Aiello, Ian Pratt-Hartmann, and Johan van Benthem, editors, *Handbook of Spatial Logics*, pages 161–215. Springer, 2007.

13 Przemysław Andrzej Wałęga, Mehul Bhatt, and Carl P. L. Schultz. ASPMT(QS): non-monotonic spatial reasoning with answer set programming modulo theories. In Francesco Calimeri, Giovambattista Ianni, and Mirosław Truszczyński, editors, *Logic Programming and Nonmonotonic Reasoning – 13th International Conference (LPNMR 2015)*, volume 9345 of *Lecture Notes in Computer Science*, pages 488–501. Springer, 2015.

14 Matthias Westphal, Stefan Wölfl, and Zeno Gantner. GQR: a fast solver for binary qualitative constraint networks. In *Benchmarking of Qualitative Spatial and Temporal Reasoning Systems, Papers from the 2009 AAAI Spring Symposium, Technical Report SS-09-02, Stanford, California, USA, March 23-25, 2009*, pages 51–52, 2009.

Rewriting Optimization Statements in Answer-Set Programs[*]

Jori Bomanson[1], Martin Gebser[2], and Tomi Janhunen[3]

1 HIIT and Department of Computer Science, Aalto University, Espoo, Finland
jori.bomanson@aalto.fi
2 Department of Computer Science, University of Potsdam, Potsdam, Germany
gebser@cs.uni-potsdam.de
3 HIIT and Department of Computer Science, Aalto University, Espoo, Finland
tomi.janhunen@aalto.fi

Abstract

Constraints on *Pseudo-Boolean* (PB) expressions can be translated into Conjunctive Normal Form (CNF) using several known translations. In *Answer-Set Programming* (ASP), analogous expressions appear in weight rules and *optimization statements*. Previously, we have translated weight rules into normal rules, using normalizations designed in accord with existing CNF encodings. In this work, we rededicate such designs to *rewrite* optimization statements in ASP. In this context, a rewrite of an optimization statement is a replacement accompanied by a set of normal rules that together replicate the original meaning. The goal is partially the same as in translating PB constraints or weight rules: to introduce new meaningful auxiliary atoms that may help a solver in the search for (optimal) solutions. In addition to adapting previous translations, we present selective rewriting techniques in order to meet the above goal while using only a limited amount of new rules and atoms. We experimentally evaluate these methods in preprocessing ASP optimization statements and then searching for optimal answer sets. The results exhibit significant advances in terms of numbers of optimally solved instances, reductions in search conflicts, and shortened computation times. By appropriate choices of rewriting techniques, improvements are made on instances involving both small and large weights. In particular, we show that selective rewriting is paramount on benchmarks involving large weights.

1998 ACM Subject Classification I.2.3 Deduction and Theorem Proving

Keywords and phrases Answer-Set Programming, Pseudo-Boolean optimization, Translation methods

Digital Object Identifier 10.4230/OASIcs.ICLP.2016.5

1 Introduction

Answer-Set Programming (ASP) is a declarative programming paradigm suited to solving computationally challenging search problems [13] by encoding them as *answer-set programs*, commonly consisting of *normal*, *cardinality*, and *weight rules*, as well as *optimization statements* [31]. The latter three relate to linear *Pseudo-Boolean* (PB) constraints [21, 30] and PB optimization statements. Rules restrict the acceptable combinations of truth values for the atoms they contain. The role of a weight rule, or a PB constraint, is to check a bound on a weighted sum of literals, whereas an optimization statement aims at minimizing such a

[*] This work was funded by the Academy of Finland (251170), DFG (SCHA 550/9), as well as DAAD and the Academy of Finland (57071677/279121).

sum. Solutions to search problems can be cast to models, called *answer sets*, and computed by employing ASP *grounders* and *solvers* [4, 17, 22, 27, 31].

In both ASP and PB solving, support for non-standard rules or constraints can be implemented by translating them into simpler logical primitives prior to solving. The performance implications of such *translation-based approaches* in comparison to *native* solving techniques are mixed: both improvements and deteriorations have been observed [2, 10]. A promising direction in recent research [3, 2] is to translate only important constraints or parts of constraints during search. In previous work, we have evaluated the feasibility of translating cardinality rules [11] and weight rules [10] into normal rules, in a process that we call *normalization*. The explored techniques build on methods successfully applied in PB solving [21, 6, 5, 1]. The main observation with relevance to this work is that normalization commonly reduces the number of search conflicts at the cost of increased instance sizes and more time spent between conflicts.

In this paper, we introduce ways in which the primitives developed for cardinality and weight rule normalization can be used to rewrite optimization statements. Indeed, typical normalization or translation methods encode sums of input weights in some number system, such as unary or mixed-radix numbers, for which the comparison to a bound is straightforward. When dealing with optimization statements, the encodings of sums can be applied to the statements while forgoing comparisons. For illustration, suppose we intend to minimize the sum $a + b$ of two atoms. Then, we may equally well minimize the sum $c + d$ of two new atoms defined by sorting a and b via the rules "c :- a. c :- b. d :- a, b." This modification preserves the answer sets of a program regarding the original atoms and the respective optimization values. As in weight rule normalization, the purpose is to enhance solving performance by supplementing problem instances with structure in the form of auxiliary atoms. The atoms are defined in intuitively meaningful ways and provide new opportunities for ASP solvers to learn *no-goods* [22]. In addition, we develop selective rewriting techniques that partition input optimization statements and then rewrite some or all of the parts in separation. As a consequence, the size increase due to rewriting is mitigated, concerning the introduced auxiliary atoms and normal rules, and we study the tradeoff between the costs and benefits of rewriting in terms of solving performance.

The paper is organized as follows. Section 2 introduces basic notations, answer-set programs, and simple optimization rewriting techniques. More elaborate techniques to rewrite optimization statements are presented in Section 3. In Section 4, we experimentally study the performance implications of applying these techniques. Related work is discussed in Section 5, and Section 6 concludes the paper.

2 Preliminaries and Basic Techniques

The following subsections introduce matrix-based expressions used to represent optimization statements and simple techniques for rewriting them.

2.1 Pseudo-Boolean Expressions

A *positive literal* is a propositional *atom* a, and not a is a *negative literal*. We write $(n \times 1)$-matrices, or (column) vectors, as $\mathbf{v} = [v_1; \ldots; v_n]$ and refer to them by symbols like $\mathbf{b}, \mathbf{d}, \mathbf{w}, \boldsymbol{\pi}$ when the elements v_i are nonnegative integers, by $\mathbf{1}$ when $v_1 = \cdots = v_n = 1$, and by $\boldsymbol{h}, \boldsymbol{l}, \boldsymbol{p}, \boldsymbol{q}, \boldsymbol{r}, \boldsymbol{s}, \boldsymbol{t}$ when $v_1, \ldots, v_n$ are literals. The *concatenation* of vectors $\mathbf{v}_1$ and $\mathbf{v}_2$ is $[\mathbf{v}_1; \mathbf{v}_2]$. We also use $(m \times n)$-matrices W of nonnegative integers w_{ij}, where i is the row and j the column. For the result W = AB of multiplying $(m \times k)$- and $(k \times n)$-matrices with elements a_{ij} and b_{ij}, respectively, we have $w_{ij} = \sum_{l=1}^{k} a_{il} b_{lj}$. In the *transpose* W^{T}, each w_{ij} is swapped with w_{ji}. We define a *Pseudo-Boolean* (PB) *expression* e to be a linear combination of nonnegative *weights* $\mathbf{w} = [w_1; \ldots; w_n]$ and literals $\boldsymbol{l} = [l_1; \ldots; l_n]$:

$$e = \mathbf{w}^{\mathrm{T}}\boldsymbol{l} = [w_1 \cdots w_n] \begin{bmatrix} l_1 \\ \vdots \\ l_n \end{bmatrix} = w_1 l_1 + \cdots + w_n l_n.$$

Let $\mathrm{A}(e) = \mathrm{A}(\boldsymbol{l}) = \{a \mid 1 \leq i \leq n, l_i = a \text{ or } l_i = \text{not } a\}$ denote the set of atoms in e and $\boldsymbol{l}$. An *interpretation* I is a set of atoms distinguishing true atoms $a \in I$ and false atoms $a \notin I$. A vector like $\boldsymbol{l}$ *evaluates* to $\boldsymbol{l}(I) = [b_1; \ldots; b_n]$ at I, where for $1 \leq i \leq n$, we have $b_i = 1$ iff $l_i = a$ and $a \in I$, or $l_i = \text{not } a$ and $a \notin I$, and $b_i = 0$ otherwise. An expression like e *evaluates* to $e(I) = (\mathbf{w}^{\mathrm{T}}\boldsymbol{l})(I) = \mathbf{w}^{\mathrm{T}}(\boldsymbol{l}(I))$ at I. We extend this notation to sums of expressions by letting $(e_1 + \cdots + e_m)(I) = e_1(I) + \cdots + e_m(I)$. Two expressions e_1 and e_2 are *equivalent*, denoted by $e_1 \equiv e_2$, iff $e_1(I) = e_2(I)$ for each $I \subseteq \mathrm{A}(e_1) \cup \mathrm{A}(e_2)$.

2.2 Answer-Set Programs

We consider (ground) answer-set *programs* P, defined as sets of (normal) *rules*, which are triples $r = (a, B, C)$ of a *head* atom a and sets of *positive body* atoms B and *negative body* atoms C. An *optimization program* O is a program-expression pair (P, e), written in the *ASP-Core-2* input language format [14] using the forms

$$a \text{ :- } b_1, \ldots, b_k, \text{not } c_1, \ldots, \text{not } c_m. \tag{1}$$

$$:\sim l_1.\ [w_1, 1] \quad \ldots \quad :\sim l_n.\ [w_n, n] \tag{2}$$

$$\#\text{minimize } \{w_1, 1 : l_1; \ldots; w_n, n : l_n\}. \tag{3}$$

for rules $(a, \{b_1, \ldots, b_k\}, \{c_1, \ldots, c_m\})$ in (1), and *weak constraints* in (2) or *optimization statements* in (3) for the expression $e = w_1 l_1 + \cdots + w_n l_n$.

Let $\mathrm{A}(P) = \bigcup_{(a,B,C)\in P}(\{a\} \cup B \cup C)$, $\mathrm{H}(O) = \mathrm{H}(P) = \{a \mid (a, B, C) \in P\}$, and $\mathrm{A}(O) = \mathrm{A}(P) \cup \mathrm{A}(e)$ denote the sets of atoms occurring in P, as heads in P, or in $O = (P, e)$, respectively. An interpretation I *satisfies* a rule $r = (a, B, C)$ iff $B \subseteq I$ and $C \cap I = \emptyset$ imply $a \in I$. The *reduct* of P with respect to I is $P^I = \{(a, B, \emptyset) \mid (a, B, C) \in P, C \cap I = \emptyset\}$. The set $\mathrm{SM}(P)$ of *stable models* of P, also called *answer sets* of P, is the set of all interpretations $M \subseteq \mathrm{A}(P)$ that are subset-minimal among the interpretations satisfying every rule $r \in P^M$. A stable model M of P is *optimal* iff $e(M) = \min\{e(N) \mid N \in \mathrm{SM}(P)\}$.

Our goal is to rewrite optimization statements while preserving the stable models of a program and associated optimization values. To this end, we utilize notions for comparing the joint parts of optimization programs [26]. Two sets S_1 and S_2 of interpretations are *visibly equal* with respect to a set V of *visible* atoms iff there is a bijection $f : S_1 \to S_2$ such that, for each $I \in S_1$, we have $I \cap V = f(I) \cap V$.

For two sets V_1 and V_2 of atoms, a program P *realizes* a function $f : 2^{V_1} \to 2^{V_2}$ iff for each $I \subseteq V_1$, there is exactly one $M \in \mathrm{SM}(P \cup \{a. \mid a \in I\})$ and $f(I) = M \cap V_2$. An optimization program $O = (P, e')$ is an *optimization rewrite* of an expression e with respect to a set V of visible atoms iff P realizes a function $f : 2^V \to 2^{\mathrm{A}(e')}$ such that, for each $I \subseteq V$, we have $e(I) = e'(f(I))$. In this case, we also say that e is *rewritable* as (P, e') with respect to V.

To decompose optimization rewrites, we say that a set V of atoms and a sequence $P_1, \ldots, P_m$ of programs *fit* iff $(V \cup \bigcup_{j=1}^{i-1} \mathrm{A}(P_j)) \cap \mathrm{H}(P_i) = \emptyset$ for each $1 \leq i \leq m$. Programs that fit preserve the definitions of atoms in V and the programs preceding them.

▶ **Proposition 1.** *Let $O = (P, e)$ be an optimization program, and e be rewritable as (P', e') with respect to $\mathrm{A}(O)$ such that $\mathrm{A}(O)$ and P' fit. Then, there is a bijection $f : \mathrm{SM}(P) \to \mathrm{SM}(P \cup P')$ such that*

1. $\mathrm{SM}(P)$ *and* $\mathrm{SM}(P \cup P')$ *are visibly equal with respect to* $\mathrm{A}(O)$ *via* f*, and*
2. $e(M) = e'(f(M))$ *for each* $M \in \mathrm{SM}(P)$.

2.3 Optimization Rewrites for Small Weights

In this subsection, we examine simple, yet effective rewriting techniques applicable to optimization statements with small weights. To begin with, we define building blocks for sorting operations on vectors of literals. Intuitively, a vector $\boldsymbol{s}$ of literals encodes the value $(\mathbf{1}^{\mathrm{T}}\boldsymbol{s})(I)$ at an interpretation I. When $\boldsymbol{s}$ is sorted, it represents this value as a *unary* number. To obtain such numbers, vectors of literals can be recursively sorted and added up via merging. These operations permute truth values, and hence preserve the encoded values.

▶ **Definition 2.** A vector $\boldsymbol{s}$ of literals is *sorted at* an interpretation I iff the weights in $\boldsymbol{s}(I)$ are monotonically decreasing, and *sorted under* a set S of interpretations iff $\boldsymbol{s}$ is sorted at each $I \in S$.

▶ **Definition 3.** Let $\boldsymbol{t} = [\boldsymbol{h}_1; \boldsymbol{h}_2]$ and $\boldsymbol{s}$ be vectors of literals having the same length, and P be a program realizing a function $f : 2^{\mathrm{A}(\boldsymbol{t})} \to 2^{\mathrm{A}(\boldsymbol{s})}$. Then, P is

1. a *sorting program* with input $\boldsymbol{t}$ and output $\boldsymbol{s}$ iff for each $I \subseteq \mathrm{A}(\boldsymbol{t})$, $\boldsymbol{s}$ is sorted at $f(I)$ and $(\mathbf{1}^{\mathrm{T}}\boldsymbol{t})(I) = (\mathbf{1}^{\mathrm{T}}\boldsymbol{s})(f(I))$;
2. a *merging program* with inputs $\boldsymbol{h}_1, \boldsymbol{h}_2$ and output $\boldsymbol{s}$ iff for each $I \subseteq \mathrm{A}(\boldsymbol{t})$ at which $\boldsymbol{h}_1$ and $\boldsymbol{h}_2$ are sorted, $\boldsymbol{s}$ is sorted at $f(I)$ and $(\mathbf{1}^{\mathrm{T}}[\boldsymbol{h}_1; \boldsymbol{h}_2])(I) = (\mathbf{1}^{\mathrm{T}}\boldsymbol{s})(f(I))$.

Moreover, for any program $P' \supseteq P$, we assume that $(\mathrm{A}(P) \cup \mathrm{A}(\boldsymbol{s})) \cap \mathrm{H}(P' \setminus P) \subseteq \mathrm{A}(\boldsymbol{t})$.

▶ **Example 4.** A sorting program P with input $\boldsymbol{t} = [t_1; t_2]$ and output $\boldsymbol{s} = [s_1; s_2]$ such that $\mathrm{A}(\boldsymbol{t}) \cap \mathrm{A}(\boldsymbol{s}) = \{t_1, t_2\} \cap \{s_1, s_2\} = \emptyset$, which yields an optimization rewrite $(P, \mathbf{1}^{\mathrm{T}}\boldsymbol{s})$ of $\mathbf{1}^{\mathrm{T}}\boldsymbol{t}$ with respect to $\mathrm{A}(\boldsymbol{t}) = \{t_1, t_2\}$, is given by

$$s_1 \text{ :- } t_1. \quad s_1 \text{ :- } t_2. \quad s_2 \text{ :- } t_1,\, t_2.$$

This program can serve as a base case in recursive constructions of larger sorting programs, such as the following. Given vectors $\boldsymbol{h}_1$, $\boldsymbol{h}_2$, $\boldsymbol{s}_1$, $\boldsymbol{s}_2$ and $\boldsymbol{s}$ of literals having appropriate lengths, a sorting program with input $\boldsymbol{t} = [\boldsymbol{h}_1; \boldsymbol{h}_2]$ and output $\boldsymbol{s}$ is recursively obtained as the union of (i) a sorting program P_1 with input $\boldsymbol{h}_1$ and output $\boldsymbol{s}_1$, (ii) a sorting program P_2 with input $\boldsymbol{h}_2$ and output $\boldsymbol{s}_2$, and (iii) a merging program P_3 with inputs $\boldsymbol{s}_1, \boldsymbol{s}_2$ and output $\boldsymbol{s}$, assuming that $\mathrm{A}(\boldsymbol{t})$ and P_1, P_2, P_3 fit.

Note that, by definition, sorting programs are merging programs, and both lend themselves to rewriting expressions with unit weights. Moreover, such *optimization rewriting* can be applied to an arbitrary expression $\mathbf{w}^{\mathrm{T}}\boldsymbol{l}$ after *flattening* it into the form $\mathbf{1}^{\mathrm{T}}\boldsymbol{t}$, where $\boldsymbol{t}$ is any vector such that $\mathbf{1}^{\mathrm{T}}\boldsymbol{t} \equiv \mathbf{w}^{\mathrm{T}}\boldsymbol{l}$. For example, $e = 2a + 4b + 3c + 3d + e + 4f$ is reproduced by picking $\boldsymbol{t} = [a; a; b; b; b; b; c; c; c; d; d; d; e; f; f; f; f]$. Rewrites based on flattening can, however, become impractically large when there are literals with large non-unit weights. To alleviate this problem, we consider selective rewriting techniques in Section 3.3, and alternative ways to handle non-unit weights in Section 3.4.

3 More Elaborate Rewriting Techniques

In this section, we present techniques for rewriting optimization statements that build on those presented in Section 2.3. They are based on a process in which we rewrite optimization statements in parts using simple techniques and then form new substitute optimization statements from the outputs. Before going into the details, let us illustrate the basic idea.

▶ **Example 5.** Consider the minimization statement

$$\#\text{minimize}\ \{5, 1 : a;\ 10, 2 : b;\ 15, 3 : c\}.$$

The statement encodes the expression $5a + 10b + 15c$. To deal with the non-unit weights, we can flatten it into $5a+5b+5b+5c+5c+5c$ and rewrite it using a single sorting program with input $[a;\ b;\ b;\ c;\ c;\ c]$. On the other hand, we obtain a more concise rewrite by modifying the expression into $5a + 5c + 10b + 10c$, sorting the parts $[a;\ c]$ and $[b;\ c]$ into vectors of auxiliary atoms $[d;\ e]$ and $[f;\ g]$, and minimizing the expression $5d + 5e + 10f + 10g$:

$$d \text{ :- } a. \quad d \text{ :- } c. \quad e \text{ :- } a, c.$$
$$f \text{ :- } b. \quad f \text{ :- } c. \quad g \text{ :- } b, c.$$
$$\#\text{minimize}\ \{5, 1 : d;\ 5, 2 : e;\ 10, 3 : f;\ 10, 4 : g\}.$$

3.1 Mixed-radix Bases and Decomposition

We define a *mixed-radix base* to be any pair $(\mathbf{b}, \boldsymbol{\pi})$ of *radices* $\mathbf{b} = [b_1;\ \dots;\ b_k]$ and *place values* $\boldsymbol{\pi} = [\pi_1;\ \dots;\ \pi_k]$ such that $b_k = \infty$ and, for each $1 \leq i \leq k$, we have $\pi_i = \prod_{j=1}^{i-1} b_i$. Examples of mixed-radix bases include the usual base for counting seconds, minutes, hours, and days, $([60;\ 60;\ 24;\ \infty], [1;\ 60;\ 3600;\ 86400])$, and the finite-length binary base $([2;\ \dots;\ 2;\ \infty], [1;\ 2;\ 4;\ \dots;\ 2^{k-1}])$ for any $k \geq 1$. In a given base $(\mathbf{b}, \boldsymbol{\pi})$, every nonnegative integer d has a *mixed-radix decomposition* with *digits* $\mathbf{d} = [d_1;\ \dots;\ d_k]$ such that $\mathbf{d}^{\mathrm{T}}\boldsymbol{\pi} = d$, and exactly one decomposition $\mathbf{d}$ of d satisfies $d_i < b_i$ for all $1 \leq i < k$. We say that d_i is the ith *least* or the $(k + 1 - i)$th *most significant* digit of d in base $(\mathbf{b}, \boldsymbol{\pi})$. More generally, the mixed-radix decomposition of an $(n \times 1)$-vector $\mathbf{w}$ of weights is a $(k \times n)$-matrix W such that $\mathrm{W}^{\mathrm{T}}\boldsymbol{\pi} = \mathbf{w}$. By these definitions, the ith row gives the ith least significant digits of all weights, and the jth column gives the digits of the jth weight.

▶ **Example 6.** In base $(\mathbf{b}, \boldsymbol{\pi}) = ([3;\ 2;\ 2;\ \infty],\ [1;\ 3;\ 6;\ 12])$, we may decompose weights $\mathbf{w} = \begin{bmatrix} 21 \\ 1 \\ 3 \\ 5 \end{bmatrix}$ into a matrix $\mathrm{W} = \begin{bmatrix} 0&1&0&2 \\ 1&0&1&1 \\ 1&0&0&0 \\ 1&0&0&0 \end{bmatrix}$ such that $\mathrm{W}^{\mathrm{T}}\boldsymbol{\pi} = \begin{bmatrix} 0&1&1&1 \\ 1&0&0&0 \\ 0&1&0&0 \\ 2&1&0&0 \end{bmatrix} \begin{bmatrix} 1 \\ 3 \\ 6 \\ 12 \end{bmatrix} = \begin{bmatrix} 21 \\ 1 \\ 3 \\ 5 \end{bmatrix} = \mathbf{w}$.

3.2 Selecting Mixed-radix Bases

Given a vector $\mathbf{w} = [w_1;\ \dots;\ w_n]$ of weights, our goal is to pick a base $(\mathbf{b}, \boldsymbol{\pi})$ with some number k of radices that yields a decomposition matrix W with small digits w_{ij} such that $\mathrm{W}^{\mathrm{T}}\boldsymbol{\pi} = \mathbf{w}$. In view of sorting programs, introduced in Section 2.3, whose sizes are of order $\mathrm{c}_{\mathrm{sort}}(n) = n(\log n)^2$, the aim is to minimize the size needed for sorting every row:

$$\mathrm{c}(\mathbf{b}, \mathbf{w}) = \sum_{i=1}^{k} \mathrm{c}_{\mathrm{sort}} \left(\sum_{j=1}^{n} w_{ij} \right).$$

To this end, we use a greedy heuristic algorithm: for each $i = 1, 2, \dots$, let $\pi_i = \prod_{j=1}^{i-1} b_j$ and pick radix b_i as the product of the least prime p that minimizes $\mathrm{c}([p;\ 2;\ \dots;\ 2],$

$[\lfloor w_1/\pi_i \rfloor; \ldots; \lfloor w_n/\pi_i \rfloor])$ and the greatest common divisor of $\{\lfloor w_j/(\pi_i p) \rfloor \mid 1 \leq j \leq n\}$, defined here to be infinite for $\{0\}$, and stop at $b_i = \infty$. We note that complete optimization procedures were proposed for finding optimal bases in translating PB constraints [18].

3.3 Selective Optimization Rewriting

In the following, we define ways to carry out partial optimization rewriting. The goal is to reduce the needed size while retaining as much of the benefits of rewriting as possible.

By additivity, sums of expressions can be rewritten term-by-term. Recall that, given an expression $e = \mathbf{w}^{\mathrm{T}}\boldsymbol{l}$, we may decompose $\mathbf{w}$ in any mixed-radix base to obtain a matrix W. Then, $\mathbf{w}^{\mathrm{T}} = (\mathrm{W}^{\mathrm{T}}\boldsymbol{\pi})^{\mathrm{T}} = \boldsymbol{\pi}^{\mathrm{T}}\mathrm{W}$ yields $e = \boldsymbol{\pi}^{\mathrm{T}}\mathrm{W}\boldsymbol{l}$, and any sum such that $\mathrm{W}_1 + \cdots + \mathrm{W}_m = \mathrm{W}$ carries over to a sum reproducing the expression $e = \boldsymbol{\pi}^{\mathrm{T}}(\sum_{i=1}^{m} \mathrm{W}_i)\boldsymbol{l} = \sum_{i=1}^{m} \boldsymbol{\pi}^{\mathrm{T}}\mathrm{W}_i\boldsymbol{l}$.

▶ **Lemma 7.** *Let* $\mathrm{W} = \mathrm{W}_1 + \cdots + \mathrm{W}_m$ *be a mixed-radix decomposition such that* $\mathrm{W}^{\mathrm{T}}\boldsymbol{\pi} = \mathbf{w}$ *for an expression* $e = \mathbf{w}^{\mathrm{T}}\boldsymbol{l}$*. For* $1 \leq i \leq m$*, let* $\boldsymbol{\pi}^{\mathrm{T}}\mathrm{W}_i\boldsymbol{l}$ *be rewritable as* $O_i = (P_i, e_i)$ *with respect to* $\mathrm{A}(e)$ *such that* $\mathrm{A}(e)$ *and* $O_1, \ldots, O_{i-1}, O_{i+1}, \ldots, O_m, O_i$ *fit. Then, e is rewritable as* $(\bigcup_{i=1}^{m} P_i, \sum_{i=1}^{m} e_i)$ *with respect to* $\mathrm{A}(e)$*.*

This opens up selective rewriting strategies. To this end, suppose there are k radices in a base $(\mathbf{b}, \boldsymbol{\pi})$, and let $\mathbf{w} = [w_1; \ldots; w_n]$, so that W is a $(k \times n)$-matrix. For any $m \in \{k, n\}$ and $S \subseteq \{1, \ldots, m\}$, let I_S denote the symmetric $(m \times m)$-*selection* matrix having the value 1 in row i and column i iff $i \in S$, and 0 otherwise. Intuitively, when fixing some $S \subseteq \{1, \ldots, n\}$, the $(k \times n)$-matrix WI_S selects weights w_i for all $i \in S$, while columns $j \notin S$ are set to zero. Similarly, when we fix some $S \subseteq \{1, \ldots, k\}$, the $(k \times n)$-matrix $\mathrm{I}_S\mathrm{W}$ captures the ith significant digits for all $i \in S$, and rows $j \notin S$ are set to zero.

▶ **Example 8.** Given $\mathrm{W} = \begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}$ and $S = \{1, 2\}$, we have $\mathrm{I}_S = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$, $\mathrm{WI}_S = \begin{bmatrix} 0 & 1 & 0 \\ 3 & 4 & 0 \\ 6 & 7 & 0 \end{bmatrix}$, and $\mathrm{I}_S\mathrm{W} = \begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 0 & 0 & 0 \end{bmatrix}$.

By means of selection matrices, we can conveniently partition W in either dimension, along its columns or rows, respectively. Namely, given a partition $S_1, \ldots, S_l$ of $\{1, \ldots, m\}$ for $m \in \{k, n\}$, we have that $\sum_{i=1}^{l} \mathrm{I}_{S_i}$ is the identity matrix. In view of Lemma 7, $\sum_{i=1}^{l} \mathrm{WI}_{S_i} = \mathrm{W}$, if $m = n$, and $\sum_{i=1}^{l} \mathrm{I}_{S_i}\mathrm{W} = \mathrm{W}$, if $m = k$, thus yield literal-wise or significance-wise optimization rewrites, respectively. For example, lines (a) in Figure 1 draw the partition $\{1, 2, 3\}, \{4, 5, 6\}, \{7, 8, 9\}$ applied to $m = n = 9$ literals, limiting the size of parts to $t = 3$. We below focus on particular cases of such *matrix partition rewrites*.

To begin with, we may rewrite some *bounded* number t of weighted literals in *equal-weight chunks* by forming a partition $S_1, \ldots, S_l$ based on (maximal) sets S_i such that $|S_i| \leq t$ and $|\{w_i \mid i \in S_i\}| = 1$, so that each chunk consists of up to t literals of the same weight. Second, we may let $l = k$ and pick $S_i = \{i\}$ for each $1 \leq i \leq k$ to rewrite an expression in *digit-wise* layers of the form $\boldsymbol{\pi}^{\mathrm{T}}(\mathrm{I}_{S_i}\mathrm{W})\boldsymbol{l}$. Third, we may drop a number t of least significant digits from each weight, such as those above line (b) in Figure 1, and rewrite the *globally* most significant digits only. This amounts to using quotients due to division by π_{t+1}, and can be formalized by applying the previous technique to a base $([\pi_{t+1}; \infty], [1; \pi_{t+1}])$, where only the quotient weights $\lfloor w_i/\pi_{t+1} \rfloor$ are rewritten for $1 \leq i \leq n$.

Matrix partition rewrites not a priori referring to particular digits include the *literal-wise* approach indicated by lines (a) in Figure 1. Another strategy is to pick, for each weight, a number t of its *locally* most significant digits starting from the most significant nonzero digit. To express this as a matrix partition rewrite, let W be a $(k \times n)$-matrix as before. Then, define an equally-sized matrix W_2 by mapping the elements $w_1, \ldots, w_k$

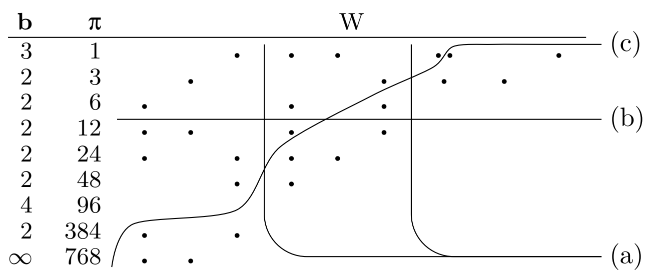


Figure 1 A mixed-radix decomposition matrix expressed using dots for digit 1 and pairs of dots for 2. The lines represent partitions for matrix partition rewrites. Columns separated by lines (a) denote a literal-wise partition, using the parameter value $t = 3$. Digits below lines (b) and (c) represent the globally or locally most significant digits to be rewritten, based on $t = 3$ or $t = 2$, respectively.

of each column in W to elements $v_1, \dots, v_k$ of a respective column in W_2 such that $v_i = \max\{0, \min\{w_i, t - \sum_{j=i+1}^{l} u_j\}\}$ for each $1 \le i \le k$, where $[u_1; \dots; u_l] = [b_1; \dots; b_{l-1}; w_l]$, $w_l \ne 0$, and $w_{l+1} = \cdots = w_k = 0$. This yields $\mathrm{W} = \mathrm{W}_1 + \mathrm{W}_2$ for $\mathrm{W}_1 = \mathrm{W} - \mathrm{W}_2$, so that, by Lemma 7, $\boldsymbol{\pi}^{\mathrm{T}}\mathrm{W}_1\boldsymbol{l}$ and $\boldsymbol{\pi}^{\mathrm{T}}\mathrm{W}_2\boldsymbol{l}$ can be rewritten separately. Similar to rewriting the globally most significant digits only, we may rewrite the locally more significant part W_2, as located below line (c) in Figure 1 for $t = 2$, but not the rest.

3.4 Optimization Rewrites for Large Weights

We build on sorting and merging programs to devise optimization rewrites applicable to expressions containing large non-unit weights. To this end, we begin by defining an abstract class of rewrites that encode unary numbers as sorted vectors of literals. This allows us to compose rewrites from building blocks that rely on sorted inputs to produce sorted outputs.

▶ **Definition 9.** Let $O = (P, d + \alpha_1 \mathbf{1}^{\mathrm{T}}\boldsymbol{s}_1 + \cdots + \alpha_m \mathbf{1}^{\mathrm{T}}\boldsymbol{s}_m)$ be an optimization rewrite of an expression e such that P realizes $f : 2^{\mathrm{A}(e)} \to 2^{\mathrm{A}(d + \alpha_1 \mathbf{1}^{\mathrm{T}}\boldsymbol{s}_1 + \cdots + \alpha_m \mathbf{1}^{\mathrm{T}}\boldsymbol{s}_m)}$. Then, O is a *multi-unary rewrite* of e with the set $\{\boldsymbol{s}_1, \dots, \boldsymbol{s}_m\}$ of vectors as its output iff $\boldsymbol{s}_i$ is sorted under the image of f for all $1 \le i \le m$.

Recall the strategy from Section 3.3 to rewrite an expression in digit-wise layers based on a mixed-radix decomposition. Such a *digit-wise* rewrite can be realized by flattening along with sorting programs applied to the layers in parallel. In this process, the radix b_i for a layer i limits the number of (flattened) inputs to a corresponding sorting program.

▶ **Proposition 10.** *Let* W *be a mixed-radix decomposition in a base* $(\mathbf{b}, \boldsymbol{\pi})$ *with place values* $\boldsymbol{\pi} = [\pi_1; \dots; \pi_k]$ *such that* $\mathrm{W}^{\mathrm{T}}\boldsymbol{\pi} = \mathbf{w}$ *for an expression* $e = \mathbf{w}^{\mathrm{T}}\boldsymbol{l}$. *For* $1 \le i \le k$, *let* $\mathbf{w}_i$ *be the ith row of* W, $\boldsymbol{t}_i$ *be some vector such that* $\mathbf{1}^{\mathrm{T}}\boldsymbol{t}_i \equiv \mathbf{w}_i^{\mathrm{T}}\boldsymbol{l}$, *and* P_i *be a sorting program with input* $\boldsymbol{t}_i$ *and output* $\boldsymbol{s}_i$ *such that* $\mathrm{A}(e)$ *and*

$$P_1, \dots, P_{i-1}, P_{i+1}, \dots, P_k, P_i \cup \{(a, \emptyset, \emptyset) \mid a \in (\mathrm{A}(P_i) \cup \mathrm{A}(\boldsymbol{s}_i)) \cap (\mathrm{A}(e) \setminus \mathrm{A}(\boldsymbol{t}_i))\}$$

fit. Then, $(\bigcup_{i=1}^{k} P_i, \sum_{i=1}^{k} \pi_i \mathbf{1}^{\mathrm{T}}\boldsymbol{s}_i)$ *is a multi-unary rewrite of* e *with output* $\{\boldsymbol{s}_1, \dots, \boldsymbol{s}_k\}$.

The above conditions that $\mathrm{A}(e)$ and sorting programs fit (in any order) as well as that atoms from $\mathrm{A}(e)$ are used as inputs in $\boldsymbol{t}_i$ only make sure that the sorting programs define disjoint atoms and otherwise evaluate nothing but their inputs. While such conditions are easy to establish, in practice, different sorting programs may share common substructures based on the same inputs. In fact, a scheme for optimizing the layout of sorting programs towards structure sharing is given in [10].

Digit-wise rewrites based on sorting programs yield non-unique mixed-radix decompositions of the sum of input weights. For example, given $\mathbf{b} = [6; \infty]$, $\boldsymbol{\pi} = [1; 6]$, and $e = 5a + 5b + 10c + d$, the sorting programs from Proposition 10 realize a function $f : 2^{\{a,b,c,d\}} \to 2^{\{s_{1,1},\dots,s_{1,15},s_{2,1}\}}$ leading to an output expression e' of the form $s_{1,1} + \cdots + s_{1,15} + 6s_{2,1}$. Then, the sum 10, associated with both $I_1 = \{a, b\}$ and $I_2 = \{c\}$, is mapped to $f(I_1) = \{s_{1,1}, \dots, s_{1,10}\}$ or $f(I_2) = \{s_{1,1}, \dots, s_{1,4}, s_{2,1}\}$, respectively, where $e'(f(I_1)) = e'(f(I_2)) = 10$. In terms of ASP solving, this means that a bound on the optimization value is not captured by a single no-good. Instead, several representations of the same value may be produced during search. An encoding of a *unique* mixed-radix decomposition can be built by combining sorting programs with *deferred carry propagation*, utilizing merging programs, as introduced in Section 2.3, to express addition along with division of unary numbers by constants. To begin with, we formalize the role of a merging program in this context.

▶ **Lemma 11.** *Let $O = (P, d + \alpha\mathbf{1}^{\mathrm{T}}\boldsymbol{h}_1 + \alpha\mathbf{1}^{\mathrm{T}}\boldsymbol{h}_2)$ be a multi-unary rewrite of an expression e with output $\{\boldsymbol{h}_1, \boldsymbol{h}_2\}$, and P' be a merging program with inputs $\boldsymbol{h}_1, \boldsymbol{h}_2$ and output $\boldsymbol{s}$ such that $\mathrm{A}(O)$ and $P' \cup \{(a, \emptyset, \emptyset) \mid a \in (\mathrm{A}(P') \cup \mathrm{A}(\boldsymbol{s})) \cap (\mathrm{A}(e) \setminus \mathrm{A}([\boldsymbol{h}_1; \boldsymbol{h}_2]))\}$ fit. Then, $(P \cup P',$ $d + \alpha\mathbf{1}^{\mathrm{T}}\boldsymbol{s})$ is a multi-unary rewrite of e with output $\{\boldsymbol{s}\}$.*

The purpose of merging programs is to map the sum of digits in one layer and a corresponding carry from less significant layers to a unary number, which can in turn provide a carry to the next layer. To obtain such carries, we make use of division. Namely, given a vector $\boldsymbol{s} = [s_1; \dots; s_n]$ and a positive integer m, we define the *quotient* of $\boldsymbol{s}$ divided by m as $[s_m; s_{2m}; \dots; s_{\lfloor n/m \rfloor m}]$, which contains every mth literal of $\boldsymbol{s}$. A respective residue, also represented as a unary number, is produced by a program as follows.

▶ **Definition 12.** Let $\boldsymbol{s} = [s_1; \dots; s_n]$ and $\boldsymbol{r} = [r_1; \dots; r_k]$ be vectors of literals such that $k = \min\{n, m-1\}$ for some positive integer m. A program P is a *residue program* modulo m with input $\boldsymbol{s}$ and output $\boldsymbol{r}$ iff P realizes a function $f : 2^{\mathrm{A}(\boldsymbol{s})} \to 2^{\mathrm{A}(\boldsymbol{r})}$ such that, for each $I \subseteq \mathrm{A}(\boldsymbol{s})$ at which $\boldsymbol{s}$ is sorted, $\boldsymbol{r}$ is sorted at $f(I)$ and $(\mathbf{1}^{\mathrm{T}}\boldsymbol{r})(f(I)) = (\mathbf{1}^{\mathrm{T}}\boldsymbol{s})(I) \bmod m$. Moreover, for any program $P' \supseteq P$, we assume that $(\mathrm{A}(P) \cup \mathrm{A}(\boldsymbol{r})) \cap \mathrm{H}(P' \setminus P) \subseteq \mathrm{A}(\boldsymbol{s})$.

▶ **Example 13.** A residue program modulo m with input $\boldsymbol{s} = [s_1; \dots; s_n]$ and output $\boldsymbol{r} = [r_1; \dots; r_k]$ for $k = \min\{n, m-1\}$, generalizing the design displayed on the left of Figure 2, is given by

$$\begin{array}{lll} r_j \text{ :- } s_{qm+j}, \text{ not } s_{(q+1)m}. & \text{for } 0 \le q < \lfloor n/m \rfloor & \text{and } 1 \le j < m, \\ r_j \text{ :- } s_{qm+j}. & \text{for } q = \lfloor n/m \rfloor & \text{and } 1 \le j \le n - qm. \end{array}$$

▶ **Lemma 14.** *Let $O = (P, d + \alpha\mathbf{1}^{\mathrm{T}}\boldsymbol{s})$ be a multi-unary rewrite of an expression e with output $\{\boldsymbol{s}\}$, and P' be a residue program modulo m with input $\boldsymbol{s}$ and output $\boldsymbol{r}$ such that $\mathrm{A}(O)$ and $P' \cup \{(a, \emptyset, \emptyset) \mid a \in (\mathrm{A}(P') \cup \mathrm{A}(\boldsymbol{r})) \cap (\mathrm{A}(e) \setminus \mathrm{A}(\boldsymbol{s}))\}$ fit. Then, $(P \cup P', d + m\alpha\mathbf{1}^{\mathrm{T}}\boldsymbol{q} + \alpha\mathbf{1}^{\mathrm{T}}\boldsymbol{r})$ is a multi-unary rewrite of e with output $\{\boldsymbol{q}, \boldsymbol{r}\}$, where $\boldsymbol{q}$ is the quotient of $\boldsymbol{s}$ divided by m.*

We are now ready to augment digit-wise rewrites according to Proposition 10 with carry propagation, and call the resulting scheme (unique) *mixed-radix rewrite*. Such a rewrite resembles (parts of) the CNF encoding of PB expressions given in [21], when built from Batcher's odd-even sorting and merging networks [8].

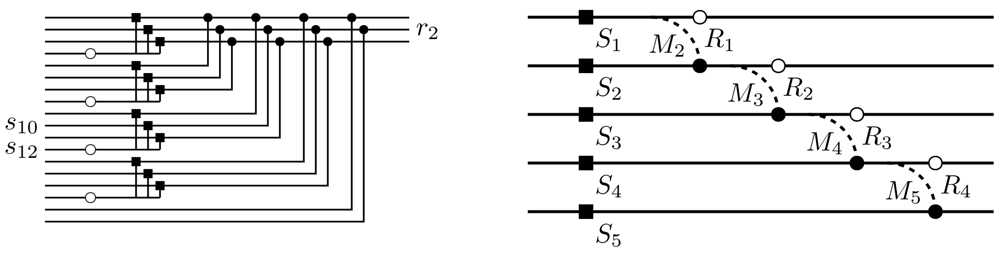


Figure 2 The residue program from Example 13 for $n = 18$ and $m = 4$ (left), with $\circ$, $\blacksquare$, and $\bullet$ standing for logical *not*s, *and*s, and *or*s, and the layout of a mixed-radix rewrite as in Proposition 15 in a base of length $k = 5$ (right), with $\circ$, $\blacksquare$, and $\bullet$ standing for R_i, S_i, and M_i, and dashed lines for quotients $\boldsymbol{q}_i$.

▶ **Proposition 15.** *Let* W *be a mixed-radix decomposition in a base* $(\mathbf{b}, \boldsymbol{\pi})$ *with radices* $\mathbf{b} = [b_1; \ldots; b_k]$ *and place values* $\boldsymbol{\pi} = [\pi_1; \ldots; \pi_k]$ *such that* $\mathrm{W}^{\mathrm{T}}\boldsymbol{\pi} = \mathbf{w}$ *for an expression* $e = \mathbf{w}^{\mathrm{T}}\boldsymbol{l}$*. For* $1 \leq i \leq k$*, let* $\mathbf{w}_i$ *be the ith row of* W*,* $\boldsymbol{t}_i$ *be some vector such that* $\mathbf{1}^{\mathrm{T}}\boldsymbol{t}_i \equiv \mathbf{w}_i^{\mathrm{T}}\boldsymbol{l}$*, and*

1. S_i *be a sorting program with input* $\boldsymbol{t}_i$ *and output* $\boldsymbol{h}_i$*,*
2. M_i *be a merging program with inputs* $\boldsymbol{h}_i, \boldsymbol{q}_i$ *and output* $\boldsymbol{s}_i$*, where* $\boldsymbol{q}_i$ *is the quotient of* $\boldsymbol{s}_{i-1}$ *divided by* b_{i-1}*, provided that* $1 < i$*,*
3. R_i *be a residue program modulo* b_i *with input* $\boldsymbol{s}_i$ *and output* $\boldsymbol{r}_i$*, provided that* $i < k$*,*

such that $\mathrm{A}(e)$ *and* $S'_1, \ldots, S'_{i-1}, S'_{i+1}, \ldots, S'_k, S'_i, M'_1, R'_1, \ldots, M'_k, R'_k$ *fit, where* $M_1 = R_k = \emptyset$*,* $\boldsymbol{s}_1 = \boldsymbol{h}_1$*,* $\boldsymbol{r}_k = \boldsymbol{s}_k$*, and* $P' = P \cup \{(a, \emptyset, \emptyset) \mid a \in (\mathrm{A}(P) \cup V_2) \cap (\mathrm{A}(e) \setminus V_1)\}$ *for any program* P *with atoms* V_1 *and* V_2 *in its input(s) or output, respectively. Then,* $(\bigcup_{i=1}^{k}(S_i \cup M_i \cup R_i), \sum_{i=1}^{k} \pi_i \mathbf{1}^{\mathrm{T}}\boldsymbol{r}_i)$ *is a multi-unary rewrite of* e *with output* $\{\boldsymbol{r}_1, \ldots, \boldsymbol{r}_k\}$*.*

The principal design of such a mixed-radix rewrite is visualized on the right of Figure 2. Using sorting and merging programs according to [8], $\mathrm{c}_{\mathrm{sort}}(n) = n(\log n)^2$ and $\mathrm{c}_{\mathrm{merge}}(n) = n \log n$ limit the size of each sorting or merging program, respectively, needed to rewrite a mixed-radix decomposition of $\mathbf{w} = [w_1; \ldots; w_n]$ in a binary base, while each residue program (modulo 2) requires linear size. As $k = \lceil \log \max\{w_1, \ldots, w_n\} \rceil$ bounds the number of programs, the resulting size is of the order $kn(\log n)^2$, which matches the CNF encoding of PB expressions given in [21].

4 Experiments

For evaluating the effect of optimization rewriting, we implemented the rewriting strategies described above in the tool LP2NORMAL (2.27),[1] and ran the ASP solver CLASP (3.1.4) [22], using its "trendy" configuration for a single thread per run, on a cluster of Linux machines equipped with Intel Xeon E5-4650 2.70GHz processors. All of the applied optimization rewrites are primarily based on sorting programs, built from (normal) ASP encodings of Batcher's odd-even merging networks [8, 11], or alternatively from merging programs that do not introduce auxiliary atoms whenever the sum of required atoms and rules is reduced in this way. Moreover, each merging program is enhanced by (redundant) integrity constraints asserting the implication from a consecutive output atom to its predecessor, groups of sorting programs are compressed by means of structure sharing [10], and rewritings are pruned by

[1] Available with benchmarks at `http://research.ics.aalto.fi/software/asp`.

Table 1 Impact of optimization rewriting on solving performance.

Connected Still-Life

	cons	time	conf	22	28	55	15
–	3.6	3.4	7.9	O	S	S	S
64	4.1	**1.0**	**5.3**	**O**	**O**	**O**	S
so	4.2	1.2	5.4	O	O	S	S

Crossing Minimization

	cons	time	conf	50	21	1	5	8
–	3.8	2.9	7.6	O	S	S	S	S
64	4.5	**0.8**	**4.8**	**O**	**O**	S	**O**	S
so	4.6	1.4	5.2	O	O	O	S	S

Maximal Clique

	cons	time	conf	51	92	10	33
–	5.9	2.9	6.6	O	S	S	S
64	6.0	**1.3**	**5.2**	**O**	**O**	**O**	S
so	6.1	1.7	**5.2**	O	O	S	S

Timetabling

	cons	time	conf	24	12	1	2	1	12	1	2	2
–	5.0	2.1	6.1	O	S	O	O	O	S	S	S	S
64	6.2	**1.9**	**5.0**	**O**	**O**	**O**	**O**	S	S	S	S	M
so	6.9	2.8	5.4	O	O	S	T	S	S	T	M	M

Bayes Alarm

	cons	time	conf	5	3	1	1	22
–	4.3	1.0	5.7	O	O	O	S	S
l1	5.3	0.6	4.7	O	O	O	S	S
l2	5.5	**0.3**	4.4	**O**	**O**	**O**	**O**	S
l3	5.6	**0.3**	**4.1**	**O**	**O**	**O**	**O**	S
g7	6.5	2.2	5.1	O	O	S	S	S
mr	6.8	2.5	5.2	O	S	S	S	S

Timetabling

	cons	time	conf	24	1	12	1	1	1	11	1	1	4
–	5.0	**2.6**	6.6	O	O	S	O	O	O	S	S	S	S
l1	6.5	2.9	**6.1**	O	S	O	S	O	O	S	S	S	M
l2	6.6	3.1	6.2	**O**	**O**	**O**	**O**	**O**	T	S	S	T	M
l3	6.7	3.1	6.4	O	O	O	O	S	T	S	S	T	M
g7	5.4	2.7	6.7	O	O	S	O	O	O	S	S	S	S
mr	6.9	2.9	6.4	O	O	O	S	T	T	S	T	T	M

Bayes Water

	cons	time	conf	15	3	1	4	4
–	3.0	2.8	7.4	O	S	S	S	S
l1	4.6	2.0	6.2	O	O	S	S	S
l2	4.8	1.8	5.8	O	O	O	S	S
l3	4.9	**1.6**	5.5	**O**	**O**	**O**	**O**	S
g7	5.5	2.1	**5.2**	O	O	O	S	S
mr	5.8	2.3	**5.2**	O	O	O	S	S

Markov Network

	cons	time	conf	19	4	2	1	1	1	2	42
–	4.1	2.0	6.6	**O**	**O**	**O**	S	S	S	**O**	S
l1	4.9	2.0	6.1	O	O	O	O	S	S	S	S
l2	5.0	**1.8**	**5.8**	**O**	**O**	**O**	**O**	S	**O**	S	S
l3	5.2	2.0	**5.8**	**O**	**O**	**O**	**O**	**O**	S	S	S
g7	5.7	2.7	6.1	O	O	S	S	S	S	S	S
mr	6.1	3.1	6.2	O	S	S	S	S	S	S	S

Bayes Hailfinder

	cons	time	conf	31	1	3	10	2	1	1	2
–	4.0	**1.4**	5.9	O	O	O	S	S	S	S	S
l1	5.3	1.6	**5.0**	O	O	O	O	S	S	S	S
l2	5.4	1.7	**5.0**	**O**	**O**	**O**	**O**	**O**	S	**O**	S
l3	5.5	1.9	5.1	**O**	**O**	**O**	**O**	**O**	**O**	S	S
g7	6.2	3.2	5.4	O	S	S	S	S	S	S	S
mr	6.5	3.0	5.3	O	O	S	S	S	S	S	S

dropping rules not needed to represent optimization values below or near the value of the first stable model found by CLASP in a (short) trial run skipping optimization.

Table 1 provides experimental results for six benchmark classes. Columns headed by numbers partition the instances of a class based on solving performance: an entry "O" expresses that optima were found and proven for the respective number of instances within 10,800s time and 16GB memory limit per run; "S" means that some solutions have been obtained, yet not proven optimal; "T" marks that no solution was reported in time; and "M" indicates aborts due to memory excess. For each class, columns "cons", "time", and "conf" give the decimal logarithms of the averages of numbers of constraints, seconds of CPU time, and conflicts reported by CLASP with respect to rewriting strategies indicated in rows. The constraints are averaged over instances on which no rewriting strategy under consideration aborted due to memory excess, and the time and conflicts are averaged over instances solved optimally with respect to all strategies. That is, accumulated runtimes and conflicts refer to the instances in a corresponding column consisting of “O” entries only, while runs without proven optima are not included. Smallest CPU times and numbers of conflicts are highlighted in boldface, and likewise the “O” entries of rewriting strategies leading to most optimally solved instances for a class.

The four benchmark classes in the upper part of Table 1, Connected Still-Life, Crossing Minimization, and Maximal Clique from the ASP Competition [15, 23] along with Curriculum-based Course Timetabling [7, 12], involve optimization statements with unit weights $w_i = 1$ or few groups of non-unit weights, respectively, in case of Timetabling. Hence, these classes lend themselves to sorting inputs in digit-wise layers, as described in Sections 2.3 and 3.3, where unit weights yield a single layer, so that sorting programs produce a unique representation of their sum as a unary number. This strategy is denoted by “so”, its bounded application to equal-weight chunks of up to $t = 64$ literals by “64”, and no rewriting at all by “–”.

Comparing these three approaches, we observe that the bounded strategy dominates in terms of optimally solved instances as well as runtimes and conflicts over the subsets of instances solved optimally with respect to all three strategies. The edge over plain CLASP without rewriting is particularly remarkable and amounts to 83 more optimally solved instances for Connected Still-Life, 26 for Crossing Minimization, 102 for Maximal Clique, and still 11 for Timetabling, considering that mixed-radix decompositions obtained via digit-wise sorting (without carry propagation) are not necessarily unique for the latter class. While the unbounded strategy also yields substantial improvements relative to plain CLASP, it incurs a significantly larger size increase that does not pay off by reducing search conflicts any further than the bounded approach, and thus does not lead to more optimally solved instances either. That is, the introduction of sorting programs and atoms for bounded unary numbers, capturing parts of optimization statements in separation, already suffices to counteract combinatorial explosion due to optimization statements to the extent feasible.

The five benchmark suites in the lower part of Table 1 stem from three classes, Bayesian Network Learning [20, 24] with samples from three data sets, Markov Network Learning [19, 25], and Curriculum-based Course Timetabling again. The corresponding instances feature non-unit weights amenable to mixed-radix rewrites, as presented in Section 3.4, which yield a unique mixed-radix decomposition of the sum of input weights. We denote this approach by “mr”, and in addition consider selective strategies based on matrix partitioning according to Section 3.3: limiting mixed-radix rewrites to a number t of locally most significant digits per weight, indicated by “l1” for $t = 1$, “l2” for $t = 2$, and “l3” for $t = 3$, as well as “g7” dropping the $t = 7$ least significant digits to rewrite the globally most significant digits of each weight only. The baseline of plain CLASP without rewriting is again marked by “–”.

Regarding these approaches, we observe that the strategies denoted by “l2” and “l3”, focusing on locally most significant digits, constitute a good tradeoff between size increase and reduction of conflicts, which in turn leads to more optimally instances solved than other rewriting strategies and plain CLASP. In fact, full rewriting “mr” blows up size more than (additionally) facilitating search, the global strategy “g7” is not flexible enough to encompass all diverging non-unit weights in an optimization statement, and plain CLASP cannot draw on rewrites to learn more effective no-goods. This becomes particularly apparent on the Bayes Hailfinder instances, where “l2” and “l3” yield 13 more optimally solved instances than plain CLASP, 16 more than “mr”, and 17 more than “g7”. For the other classes, Markov Network and Timetabling, the distinction is less clear and amounts to singular instances separating the local strategies “l2” and “l3” from each other as well as plain CLASP or full mixed-radix rewriting “mr”, respectively.

In comparison to the bounded digit-wise sorting approach denoted by “64”, also applied to Timetabling in the upper part of Table 1, the best-performing strategy “l2” based on mixed-radix rewrites leads to the same number of optimally solved instances, yet incurring two timeouts more. The latter observation indicates that the search of CLASP does not truly benefit from the unique representation of a sum of weights in relation to just producing unary numbers capturing sums of digits. We conjecture that this is due to the non-monotonic character of residue programs, while sorting and merging programs map inputs to outputs in a monotonic fashion. This suggests trying alternative rewriting approaches that avoid residue programs while dealing with diverging non-unit weights, and such methods are future work.

5 Related Work

In previous work, we have addressed the normalization of cardinality rules [11] and weight rules [10], and this paper extends the investigation of rewriting techniques to optimization statements. While normalization allows for completely eliminating cardinality and weight rules, optimization rewriting maps one expression to another, where the introduced atoms and rules add structure that provides new opportunities for ASP solvers to learn no-goods, which may benefit solving performance.

Mixed-radix rewrites, as presented in Section 3.4, resemble the CNF encoding of PB expressions given in [21]. More recent translation methods [6, 10, 29] use so-called tares to simplify bound checking for mixed-radix decompositions of PB constraints, while tares are not meaningful for optimization statements that lack fixed bounds. The hybrid CNF encoding of cardinality constraints in [9] compensates weak propagation by (small) partial building blocks, which is comparable to the selective rewriting techniques in Section 3.3. Dynamic approaches to limit the size of CNF encodings of PB expressions, complementing selective rewriting strategies, include conflict-directed lazy decomposition [3], where digit-wise rewriting is performed selectively during search. A related strategy [2] consists of fully rewriting expressions deemed relevant in PB solving. Moreover, CNF encodings of PB optimization statements can be simplified when creating them incrementally during search [28]. In contrast to such dynamic approaches, we aim at preprocessing ASP optimization statements prior to solving. As a consequence, our rewriting techniques can be flexibly combined with different optimization strategies [22, 4].

6 Conclusions

Our work extends the scope of normalization methods, as originally devised for cardinality and weight rules, by developing rewriting techniques for optimization statements as well. In this context, sorting programs serve as basic building blocks for representing sums of unit weights or digits as unary numbers. When dealing with non-unit weights, merging unary numbers amounts to carry propagation, so that residues yield a unique mixed-radix decomposition of the sum of input weights. Our rewriting strategies can be applied selectively based on partitioning a vector of weights or a corresponding matrix, respectively. Such partial optimization rewrites allow for reducing the size needed to augment answer-set programs with additional structure in order to enhance the performance of ASP solvers.

Experiments with the ASP solver CLASP showed substantially improved robustness of its model-guided optimization strategy, used by default, due to optimization rewriting. This particularly applies to benchmarks involving unit weights or moderately many groups of non-unit weights, respectively. Sorting here effectively counteracts the combinatorial explosion faced without rewriting, as no-goods over unary numbers capture much larger classes of interpretations than those stemming from optimization statements over comparably specific atoms. Mixed-radix decompositions and carry propagation helped to improve solving performance for benchmarks with non-unit weights as well, yet not by the same amount as sorting equal weights, provided equal weights exist. We conjecture that this observation is related to the non-monotonic character of residue programs, and investigating alternative approaches avoiding them is part of future work. The latter also includes in-depth experiments with core-guided optimization strategies, which in preliminary tests seemed unimpaired by rewriting, neither positively nor negatively. Finally, our experiments indicated that selective rewriting techniques require significantly less size than full rewriting to reduce search conflicts equally well, so that predefining effective adaptive selection strategies is of interest.

References

1 Ignasi Abío, Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Valentin Mayer-Eichberger. A new look at BDDs for Pseudo-Boolean constraints. *Journal of Artificial Intelligence Research*, 45:443–480, 2012. `doi:10.1613/jair.3653`.

2 Ignasi Abío, Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Peter J. Stuckey. To encode or to propagate? The best choice for each constraint in SAT. In *Proceedings of CP 2013*, volume 8124 of *LNCS*, pages 97–106. Springer, 2013. `doi:10.1007/978-3-642-40627-0_10`.

3 Ignasi Abío and Peter J. Stuckey. Conflict directed lazy decomposition. In *Proceedings of CP 2012*, volume 7514 of *LNCS*, pages 70–85. Springer, 2012. `doi:10.1007/978-3-642-33558-7_8`.

4 Mario Alviano, Carmine Dodaro, Nicola Leone, and Francesco Ricca. Advances in WASP. In Calimeri et al. [16], pages 40–54. `doi:10.1007/978-3-319-23264-5_5`.

5 Roberto Asín, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. Cardinality networks: A theoretical and empirical study. *Constraints*, 16(2):195–221, 2011. `doi:10.1007/s10601-010-9105-0`.

6 Olivier Bailleux, Yacine Boufkhad, and Olivier Roussel. New encodings of Pseudo-Boolean constraints into CNF. In *Proceedings of SAT 2009*, volume 5584 of *LNCS*, pages 181–194. Springer, 2009. `doi:10.1007/978-3-642-02777-2_19`.

7 Mutsunori Banbara, Takehide Soh, Naoyuki Tamura, Katsumi Inoue, and Torsten Schaub. Answer set programming as a modeling language for course timetabling. *Theory and Practice of Logic Programming*, 13(4-5):783–798, 2013. `doi:10.1017/S1471068413000495`.

8 Kenneth E. Batcher. Sorting networks and their applications. In *Proceedings of AFIPS 1968*, pages 307–314. ACM, 1968. `doi:10.1145/1468075.1468121`.

9 Yael Ben-Haim, Alexander Ivrii, Oded Margalit, and Arie Matsliah. Perfect hashing and CNF encodings of cardinality constraints. In *Proceedings of SAT 2012*, volume 7317 of *LNCS*, pages 397–409. Springer, 2012. `doi:10.1007/978-3-642-31612-8_30`.

10 Jori Bomanson, Martin Gebser, and Tomi Janhunen. Improving the normalization of weight rules in answer set programs. In *Proceedings of JELIA 2014*, volume 8761 of *LNCS*, pages 166–180. Springer, 2014. `doi:10.1007/978-3-319-11558-0_12`.

11 Jori Bomanson and Tomi Janhunen. Normalizing cardinality rules using merging and sorting constructions. In *Proceedings of LPNMR 2013*, volume 8148 of *LNCS*, pages 187–199. Springer, 2013. `doi:10.1007/978-3-642-40564-8_19`.

12 Alex Bonutti, Fabio De Cesco, Luca Di Gaspero, and Andrea Schaerf. Benchmarking curriculum-based course timetabling: Formulations, data formats, instances, validation, visualization, and results. *Annals of Operations Research*, 194(1):59–70, 2012. `doi:10.1007/s10479-010-0707-0`.

13 Gerhard Brewka, Thomas Eiter, and Mirosław Truszczyński. Answer set programming at a glance. *Communications of the ACM*, 54(12):92–103, 2011. doi:10.1145/2043174.2043195.

14 Francesco Calimeri, Wolfgang Faber, Martin Gebser, Giovambattista Ianni, Roland Kaminski, Thomas Krennwallner, Nicola Leone, Francesco Ricca, and Torsten Schaub. ASP-Core-2: Input language format. Available at https://www.mat.unical.it/aspcomp2013/ASPStandardization/, 2012.

15 Francesco Calimeri, Martin Gebser, Marco Maratea, and Francesco Ricca. Design and results of the fifth answer set programming competition. *Artificial Intelligence*, 231:151–181, 2016. doi:10.1016/j.artint.2015.09.008.

16 Francesco Calimeri, Giovambattista Ianni, and Miroslaw Truszczynski, editors. *Proceedings of LPNMR 2015*, volume 9345 of *LNCS*. Springer, 2015. doi:10.1007/978-3-319-23264-5.

17 Broes De Cat, Bart Bogaerts, Maurice Bruynooghe, Gerda Janssens, and Marc Denecker. Predicate logic as a modelling language: The IDP system. Available at https://arxiv.org/abs/1401.6312, 2016.

18 Michael Codish, Yoav Fekete, Carsten Fuhs, and Peter Schneider-Kamp. Optimal base encodings for Pseudo-Boolean constraints. In *Proceedings of TACAS 2011*, volume 6605 of *LNCS*, pages 189–204. Springer, 2011. doi:10.1007/978-3-642-19835-9_16.

19 Jukka Corander, Tomi Janhunen, Jussi Rintanen, Henrik J. Nyman, and Johan Pensar. Learning chordal Markov networks by constraint satisfaction. In *Proceedings of NIPS 2014*, pages 1349–1357. NIPS Foundation, 2013.

20 James Cussens. Bayesian network learning with cutting planes. In *Proceedings of UAI 2011*, pages 153–160. AUAI, 2011.

21 Niklas Eén and Niklas Sörensson. Translating Pseudo-Boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2(1-4):1–26, 2006.

22 Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence*, 187-188:52–89, 2012. doi:10.1016/j.artint.2012.04.001.

23 Martin Gebser, Marco Maratea, and Francesco Ricca. The design of the sixth answer set programming competition. In Calimeri et al. [16], pages 531–544. doi:10.1007/978-3-319-23264-5_44.

24 Tommi S. Jaakkola, David A. Sontag, Amir Globerson, and Marina Meila. Learning Bayesian network structure using LP relaxations. In *Proceedings of AISTATS 2010*, pages 358–365. JMLR Proceedings, 2010.

25 Tomi Janhunen, Martin Gebser, Jussi Rintanen, Henrik J. Nyman, Johan Pensar, and Jukka Corander. Learning discrete decomposable graphical models via constraint optimization. *Statistics and Computing*, online access, 2015. doi:10.1007/s11222-015-9611-4.

26 Tomi Janhunen and Ilkka Niemelä. Applying visible strong equivalence in answer-set program transformations. In *Essays on Logic-Based AI in Honour of Vladimir Lifschitz*, volume 7265 of *LNCS*, pages 363–379. Springer, 2012. doi:10.1007/978-3-642-30743-0_24.

27 Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, 7(3):499–562, 2006. doi:10.1145/1149114.1149117.

28 Panagiotis Manolios and Vasilis Papavasileiou. Pseudo-Boolean solving by incremental translation to SAT. In *Proceedings of FMCAD 2011*, pages 41–45. FMCAD Inc., 2011.

29 Norbert Manthey, Tobias Philipp, and Peter Steinke. A more compact translation of Pseudo-Boolean constraints into CNF such that generalized arc consistency is maintained. In

Proceedings of KI 2014, volume 8736 of *LNCS*, pages 123–134. Springer, 2014. doi:10.1007/978-3-319-11206-0_13.

30 Olivier Roussel and Vasco M. Manquinho. Pseudo-Boolean and cardinality constraints. In *Handbook of Satisfiability*, pages 695–733. IOS, 2009. doi:10.3233/978-1-58603-929-5-695.

31 Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1-2):181–234, 2002. doi:10.1016/S0004-3702(02)00187-X.

Justifications and Blocking Sets in a Rule-Based Answer Set Computation*

Christopher Béatrix[1], Claire Lefèvre[2], Laurent Garcia[3], and Igor Stéphan[4]

1 **LERIA, University of Angers, Angers, France**
`beatrix@info.univ-angers.fr`
2 **LERIA, University of Angers, Angers, France**
`claire@info.univ-angers.fr`
3 **LERIA, University of Angers, Angers, France**
`garcia@info.univ-angers.fr`
4 **LERIA, University of Angers, Angers, France**
`stephan@info.univ-angers.fr`

Abstract

Notions of justifications for logic programs under answer set semantics have been recently studied for atom-based approaches or argumentation approaches. The paper addresses the question in a rule-based answer set computation: the search algorithm does not guess on the truth or falsity of an atom but on the application or non application of a non monotonic rule. In this view, justifications are sets of ground rules with particular properties. Properties of these justifications are established; in particular the notion of blocking set (a reason incompatible with an answer set) is defined, that permits to explain computation failures. Backjumping, learning, debugging and explanations are possible applications.

1998 ACM Subject Classification D.1.6 Logic Programming, I.2.3 Logic Programming

Keywords and phrases Answer Set Programming, Justification, Rule-based Computation

Digital Object Identifier 10.4230/OASIcs.ICLP.2016.6

1 Introduction

Answer Set Programming (ASP) is a very convenient paradigm to represent knowledge in Artificial Intelligence and to encode Constraint Satisfaction Problems. It is also a very interesting way to practically solve them since some efficient solvers are available [18, 12, 5]. Usually, knowledge representation in ASP is done by means of first-order rules. But, most of the ASP solvers are propositional and they begin by an instantiation phase in order to obtain a propositional program from the first-order one. Furthermore, most of the ASP solvers are based on search algorithms where a choice point is on whether an atom is or is not in a model. But some other solvers like `Gasp` [15], `ASPeRiX` [10, 11] and `OMiGA` [3] are based on principles which do not need this preliminary instantiation of the first-order program: the rule guided approach. The choice point of the search algorithm is on the application or the non application of an on-the-fly instantiated rule.

Justifications in logic programs are intended to provide information about the reason why some property is true, in general why an atom is or is not part of an answer set. The

* This work was supported by ANR (National Research Agency), project ASPIQ under the reference ANR-12-BS02-0003.

main applications are to help users in understanding the program behavior and debugging it. Indeed, in diagnosis or decision systems, it can be important to understand why a decision is made or why a potential decision is not reached; or, in a debugging perspective, explain why an unintended solution is reached or why a given interpretation is not an answer set. Each related work addresses the problem from a specific viewpoint: for example, an atom-based approach using well-founded semantics for [16] and an argumentation framework with attacks and supports for [17]. [16] distinguishes *off-line justification* which is a reason for the truth value of an atom w.r.t. a given answer set (a complete interpretation) from *on-line justification* which is a reason for the truth value of an atom during the computation of an answer set and thus w.r.t. an incomplete interpretation.

The present work deals with on-line justification from a rule-based perspective: a justification is a set of rules with specific status, and truth values of atoms can remain undefined until the end of the computation. In on-line justifications, an interesting question is that of the failure of a computation. In practice, to explain the failures allows to help guide the search and can have direct applications in backjumping and learning. But justifications are interesting by themselves to explain (partial) results of a computation and to debug logic programs.

A rule based computation is a forward chaining process that builds a 3-valued interpretation $\langle IN, OUT\rangle$ in which each atom can be true (belongs to IN), false (belongs to OUT) or undefined (belongs neither to IN nor to OUT). At each revision step, a ground rule is chosen to be applied or to be blocked. During this process, some "reasons" (sets of ground rules, each of them having some properties – "status" – w.r.t. the interpretation under construction) can be associated to atoms justifying their adding to the interpretation. From these first reasons, we are able to compute why an atom is undefined or why the computation fails. A blocking set is defined as a reason that justifies the failure of a computation: it is composed of the applied and blocked rules responsible of the failure. In practice, the blocking sets allow to prune the search tree.

There exist several works on justification [16, 17, 1] and debugging [6, 4, 2]. Papers about justification focus on the reason why some interpretation is an answer set (explanation of the truth values of atoms in an interpretation). To our knowledge, the two closest works are [16] and [17]. [16] encodes an explanation by a graph where the nodes are atoms (annotated "true" or "false"). Their justification is based on the well-founded semantics which determines negative atoms that can be "assumed" to be false (as opposed to atoms which are always false). This corresponds to the `Smodels` solving process. The approach of [17] is in argumentative terms. The ASP program is translated into a theory of argumentation. This allows the construction of arguments and attack relation on these arguments from which justifications can be computed. Justifications are also encoded by graphs: an attack tree of an argument is a graph where the nodes are arguments and the edges are attacks or supports between arguments. An argument-based justification is defined as a flattened version of the preceding tree: it is a set of support and attack relations between atoms. [1] proposes a construction of propositional formulas that encode provenance information for the logic program; justifications can then be extracted from these formulas.

On the other hand, the goal of the debugging systems is to explain why some interpretation is not an answer set, or why an interpretation, expected not to be an answer set, is an answer set and, eventually, to propose repairs of the program. [4] characterize inconsistency in terms of bridge rules: rules which need to be altered for restoring consistency, or combination of rules which causes inconsistency. [6] uses meta-programming technique in order to find semantic errors of programs, based on some types of errors. [2] also uses meta-programming method to generate propositional formulas that encode provenance information and suggest repairs.

The paper is organized as follows. Section 2 gives some background about ASP. Section 3 presents the concept of *computation*: a constructive characterization of answer sets. In Section 4, notions of justifications and blocking sets are defined, and some of their properties are established. Section 5 concludes by some perspectives.

2 Answer Set Programming

A *normal logic program* is a set of rules like

$$c \leftarrow a_1, \dots, a_n, not\ b_1, \dots, not\ b_m. \quad (n \geq 0, m \geq 0)$$

where $c, a_1, \dots, a_n, b_1, \dots, b_m$ are atoms built from predicate symbols, constants, variables and function symbols. For a rule r (or by extension for a rule set), we note $head(r) = c$ its *head*, $body^+(r) = \{a_1, \dots, a_n\}$ its *positive body*, $body^-(r) = \{b_1, \dots, b_m\}$ its *negative body* and $body(r) = body^+(r) \cup body^-(r)$. When the negative body of a rule is not empty we say that this rule is *non-monotonic*. A *ground substitution* is a mapping from the set of variables to the set of the ground terms (terms without any variable). If t is a term (resp. a an atom) and σ a ground substitution, $\sigma(t)$ (resp. $\sigma(a)$) is a *ground instance* of t (resp. a). A program P can be seen as an intensional version of the propositional program $ground(P) = \bigcup_{r \in P} ground(r)$ where $ground(r)$ is the set of all fully instantiated rules that can be obtained by substituting every variable in r by every constant of the Herbrand universe of P. The set of *generating rules* [8] of an atom set X for a program P, $GR_P(X)$, is defined as $GR_P(X) = \{\sigma(r) \mid r \in P, \sigma$ is a ground substitution s.t. $\sigma(body^+(r)) \subseteq X$ and $\sigma(body^-(r)) \cap X = \emptyset\}$. A set of ground rules R is *grounded* if there exists an enumeration $\langle r_1 \dots r_n \rangle$ of the rules of R such that $\forall i \in [1..n]$, $body^+(r_i) \subseteq head\{r_j \mid j < i\}$. Then, X is an answer set of P (originally called a *stable model* [7]) if and only if $X = head(GR_P(X))$ and $GR_P(X)$ is grounded.

3 Rule-based Answer Set Computation

In this section, a constructive characterization of answer sets for normal logic programs, based on a concept of ***ASPeRiX** computation* [9], is presented. This concept is itself based on an abstract notion of *computation* for ground programs proposed in [13]. The only syntactic restriction required is that every rule of a program must be *safe*. That is, all variables occurring in the rule occur also in its positive body. Moreover, every constraint (i.e. headless rule) is considered given with the particular head $\perp$ and is also safe.

An ***ASPeRiX** computation* for a program P is defined as a process on a computation state based on a *partial interpretation* which is a pair $\langle IN, OUT \rangle$ of disjoint atom sets included in the Herbrand base of P. Intuitively, all atoms in IN belong to a search answer set and all atoms in OUT do not. The notion of partial interpretation defines different status for rules. If r is a rule, σ is a ground substitution and $I = \langle IN, OUT \rangle$ is a partial interpretation: $\sigma(r)$ is *supported* w.r.t. I when $body^+(\sigma(r)) \subseteq IN$, $\sigma(r)$ is *blocked* w.r.t. I when $body^-(\sigma(r)) \cap IN \neq \emptyset$, $\sigma(r)$ is *unblocked* w.r.t. I when $body^-(\sigma(r)) \subseteq OUT$, and r is *applicable* with σ w.r.t. I when $\sigma(r)$ is supported and not blocked.[1]

[1] The negation of blocked, *not blocked*, is different from *unblocked*.

An ASPeRiX computation is a forward chaining process that instantiates and fires one unique rule at each iteration according to two kinds of inference: a monotonic step of *propagation* and a nonmonotonic step of *choice*. To fire a rule means to add the head of the rule in the set IN. If P is a set of first order rules, I is a partial interpretation and R is a set of ground rules: $\Delta_{pro}(P, I, R) = \{(r, \sigma) \mid r \in P, \sigma$ is a ground substitution s.t. $\sigma(r)$ is supported and unblocked w.r.t. I, and $\sigma(r) \notin R\}$, $\Delta_{cho}(P, I, R) = \{(r, \sigma) \mid r \in P, \sigma$ is a ground substitution s.t. $\sigma(r)$ is applicable w.r.t. I and $\sigma(r) \notin R\}$. These sets contain pairs (r, σ) but, for simplicity, we sometimes consider they contain ground rules $\sigma(r)$. They are used in the following definition of an ASPeRiX computation. The specific case of constraints (rules with $\perp$ as head) is treated by adding $\perp$ to OUT set. By this way, if a constraint is fired (violated), $\perp$ should be added to IN and thus, $\langle IN, OUT\rangle$ would not be a partial interpretation. The sets R^{app} and R^{excl} represent the ground rules that are respectively fired and excluded during the computation.

▶ **Definition 1** (ASPeRiX Computation)**.** Let P be a first order normal logic program. An *ASPeRiX computation* for P is a sequence $\langle R_i, K_i, I_i\rangle_{i=0}^{\infty}$ of ground rule sets pairs $R_i = \langle R_i^{app}, R_i^{excl}\rangle$, ground rule sets K_i and partial interpretations $I_i = \langle IN_i, OUT_i\rangle$ that satisfies the following conditions:

- $R_0 = \langle\emptyset, \emptyset\rangle$, $K_0 = \emptyset$ and $I_0 = \langle\emptyset, \{\perp\}\rangle$,
- (Revision) 4 possible cases:

  (Propagation) $r_i = \sigma(r)$ for $(r, \sigma) \in \Delta_{pro}(P, I_{i-1}, R_{i-1}^{app})$,
  $R_i = \langle R_{i-1}^{app} \cup \{r_i\}, R_{i-1}^{excl}\rangle$, $K_i = K_{i-1}$
  and $I_i = \langle IN_{i-1} \cup \{head(r_i)\}, OUT_{i-1}\rangle$

  or (Rule choice) $\Delta_{pro}(P \cup K_{i-1}, I_{i-1}, R_{i-1}^{app}) = \emptyset$,
  $r_i = \sigma(r)$ for $(r, \sigma) \in \Delta_{cho}(P, I_{i-1}, R_{i-1}^{app} \cup R_{i-1}^{excl})$,
  $R_i = \langle R_{i-1}^{app} \cup \{r_i\}, R_{i-1}^{excl}\rangle$, $K_i = K_{i-1}$
  and $I_i = \langle IN_{i-1} \cup \{head(r_i)\}, OUT_{i-1} \cup body^-(r_i)\rangle$

  or (Rule exclusion) $\Delta_{pro}(P \cup K_{i-1}, I_{i-1}, R_{i-1}^{app}) = \emptyset$,
  $r_i = \sigma(r)$ for $(r, \sigma) \in \Delta_{cho}(P, I_{i-1}, R_{i-1}^{app} \cup R_{i-1}^{excl})$,
  $R_i = \langle R_{i-1}^{app}, R_{i-1}^{excl} \cup \{r_i\}\rangle$, $K_i = K_{i-1} \cup \{\perp \leftarrow \bigcup_{b \in body^-(r_i)} not\ b.\}$
  and $I_i = I_{i-1}$

  or (Stability) $R_i = R_{i-1}$, $K_i = K_{i-1}$ and $I_i = I_{i-1}$,

If $\exists i \geq 0$, $\Delta_{cho}(P \cup K_i, I_i, R_i^{app} \cup R_i^{excl}) = \emptyset$, then the computation is said to *converge*[2] to the set $IN_\infty = \bigcup_{i=0}^{\infty} IN_i$.

Revision by (Rule exclusion) is not necessary to characterize answer sets. It adds the possibility to block a rule from Δ_{cho} instead of firing it. To block a rule is to add a constraint with the negative atoms of the rule body. This possibility restricts rule choice in Δ_{cho} and thus forbids some computations: if a ground rule r is blocked, the computation can only converge to an answer set whose generating rules do not contain r. It is only useful for having a correspondence between theoretical computations and practical search trees.

▶ **Example 2.** Let P_2 be the following program:

$R_1 : v(1).\quad R_2 : v(2).\quad R_3 : v(3).\quad R_4 : green(4).\quad R_5 : edge(1,3).\quad R_6 : edge(3,4).$
$R_7 : red(X) \leftarrow v(X), not\ green(X).\quad R_9 : \perp \leftarrow edge(X,Y), red(X), red(Y).$
$R_8 : green(X) \leftarrow v(X), not\ red(X).\quad R_{10} : \perp \leftarrow edge(X,Y), green(X), green(Y).$

[2] Convergence is not always ensured due to function symbols. The problem can be fixed by limiting the nesting of function symbols.

The following sequence is an ASPeRiX computation for P_2:

$I_0 \quad = \langle \emptyset, \{\bot\} \rangle$
(Propagation)
$r_1 \quad = v(1).$ with $(R_1, \emptyset) \in \Delta_{pro}(P_2, I_0, \emptyset)$
$I_1 \quad = \langle \{\mathbf{v(1)}\}, \{\bot\} \rangle$
Steps 2 to 6 are similar: facts of the program are added to IN set by propagation.
$I_6 \quad = \langle \{v(1), v(2), v(3), green(4), edge(1,3), edge(3,4)\}, \{\bot\} \rangle$
(Rule choice) $\Delta_{pro}(P_2, I_6, \{r_1, \cdots, r_6\}) = \emptyset$
$r_7 \quad = red(1) \leftarrow v(1), not\ green(1).$ with $(R_7, \{X \leftarrow 1\}) \in \Delta_{cho}(P_2, I_6, \{r_1, \cdots, r_6\})$
$I_7 \quad = \langle \{v(1), v(2), v(3), \mathbf{red(1)}, green(4), edge(1,3), edge(3,4)\}, \{\bot, \mathbf{green(1)}\} \rangle$
(Rule choice) $\Delta_{pro}(P_2, I_7, \{r_1, \cdots, r_7\}) = \emptyset$
$r_8 \quad = red(2) \leftarrow v(2), not\ green(2).$ with $(R_7, \{X \leftarrow 2\}) \in \Delta_{cho}(P_2, I_7, \{r_1, \cdots, r_7\})$
$I_8 \quad = \langle \{v(1), v(2), v(3), red(1), \mathbf{red(2)}, green(4), edge(1,3), edge(3,4)\},$
$\{\bot, green(1), \mathbf{green(2)}\} \rangle$
(Rule choice) $\Delta_{pro}(P_2, I_8, \{r_1, \cdots, r_8\}) = \emptyset$
$r_9 \quad = red(3) \leftarrow v(3), not\ green(3).$ with $(R_7, \{X \leftarrow 3\}) \in \Delta_{cho}(P_2, I_8, \{r_1, \cdots, r_8\})$
$I_9 \quad = \langle \{v(1), v(2), v(3), red(1), red(2), \mathbf{red(3)}, green(4), edge(1,3), edge(3,4)\},$
$\{\bot, green(1), green(2), \mathbf{green(3)}\} \rangle$

$(R_9, \{X \leftarrow 1, Y \leftarrow 3\}) \in \Delta_{pro}(P_2, I_9, \{r_1, \cdots, r_9\})$ but $r_{10} = (\bot \leftarrow edge(1,3), red(1), red(3).)$ cannot be applied by (Propagation) because its head, $\bot$, is already into the OUT set. The computation does not converge, it is "blocked": the only possible revision is stability.

If the rule $r_7 = (green(1) \leftarrow v(1), not\ red(1).)$ instead of $(red(1) \leftarrow v(1), not\ green(1).)$ with $(R_8, \{X \leftarrow 1\}) \in \Delta_{cho}(P_2, I_6, \{r_1, \cdots, r_6\})$ has been chosen at step 7, other steps being the same, then

$I_9 \quad = \langle \{v(1), v(2), v(3), green(1), red(2), red(3), green(4), edge(1,3), edge(3,4), red(1)\},$
$\{\bot, green(2), green(3)\} \rangle$

$\Delta_{pro}(P_2, I_9, \{r_1, \cdots, r_9\}) = \emptyset, \ \Delta_{cho}(P_2, I_9, \{r_1, \cdots, r_9\}) = \emptyset$
$I_{10} \quad = I_9$

This last ASPeRiX computation converges to the set $\{v(1), v(2), v(3), green(1), red(2), red(3), green(4), edge(1,3), edge(3,4)\}$ which is an answer set for P_2.

The following theorem establishes a connection between the results of any ASPeRiX computation which converges and the answer sets of a normal logic program.

▶ **Theorem 3.** *[9] Let P be a normal logic program and X be an atom set. Then, X is an answer set of P if and only if there is an ASPeRiX computation $S = \langle R_i, K_i, I_i \rangle_{i=0}^{\infty}$, $I_i = \langle IN_i, OUT_i \rangle$, for P such that S converges and $IN_\infty = X$.*

Let us note that in order to respect the revision principle of an ASPeRiX computation each sequence of partial interpretations must be generated by using the propagation inference based on rules from Δ_{pro} as long as possible before using the choice based on Δ_{cho} in order to fire a nonmonotonic rule. Then, because of the non determinism of the selection of rules from Δ_{cho}, the natural implementation of this approach leads to a usual search tree where, at each node, one has to decide whether or not to fire a rule chosen in Δ_{cho}. Persistence of applicability of the nonmonotonic rule chosen by (Rule choice) to be fired is ensured by adding to the OUT set all ground atoms from its negative body. On the other branch, where the rule is not fired (Rule exclusion), the translation of its negative body into a new constraint ensures that it becomes impossible to find later an answer set in which this rule is not blocked.

4 Justifications and Blocking Sets

In an `ASPeRiX` computation, now simply called *computation*, a *reason* is a justification of some property. For instance, the property could be that some atom a belongs to IN (resp. OUT), or that a is undetermined (neither into the IN nor into the OUT sets), or that a constraint c belongs to the K set, or that the computation does not converge. A *reason* is defined as a set of numbered ground rules (numbered rules used for (Revision) in a computation). These rules are those responsible of the satisfaction of the property.

4.1 Reasons of atoms and rules

4.1.1 Reasons of the atoms in IN or OUT sets and of the constraints

We define in this section how reasons of atoms and rules are calculated in a computation: reasons why an atom is added to the IN or OUT sets and reasons why a rule is added to a program. In practice, only constraints are added during the search (to the K set) but it is easier to define reasons for all ground rules (included those issued from the initial program). Reasons are defined as follows in a computation $S = \langle R_i, K_i, I_i \rangle_{i=0}^{n}$ for a program P where $I_i = \langle IN_i, OUT_i \rangle$.

Rules from P. To each instance of rule and constraint of the initial program reason $\{r_0\}$ is associated where r_0 is a new constant with number 0. For every rule instance r of the initial program, $reason(r, S) = \{r_0\}$.

(Propagation) step. During the (Propagation) step, the head of a ground rule r_i is added to the IN_i set because the rule is supported and unblocked: all the atoms of the positive body belong to IN_{i-1} and all the atoms of the negative body belong to OUT_{i-1}. The reason of the adding of $head(r_i)$ to the IN_i set is the set of reasons why the atoms of the body are in the partial interpretation plus the rule itself: $reason(head(r_i), S) = \bigcup_{a \in body(r_i)} reason(a, S) \cup \{r_i\}$.

(Rule choice) step. During a (Rule Choice), a rule r_i is chosen to be unblocked and the atoms of the negative body of the rule are added to the OUT_i set with the only justification being that r_i has been arbitrary unblocked: $reason(a, S) = \{r_i\}$, for all $a \in body^-(r_i) \setminus OUT_{i-1}$.

During the same step, the head of r_i is added to the IN_i set (since r_i is henceforth supported and unblocked). The justification is the same as that in (Propagation) step (except that the rule is only non blocked at step $i-1$ and becomes unblocked only at step i): $reason(head(r_i), S) = \bigcup_{a \in body(r_i)} reason(a, S) \cup \{r_i\}$.

(Rule exclusion) step. During a (Rule exclusion) step, where a rule r_i is chosen to be blocked, a constraint is added to the K_i set. To such a constraint, generated to block the chosen rule r_i, is associated the reason $\{r_i\}$ since this arbitrary choice is only justified by itself: $reason(\perp \leftarrow body^-(r_i)., S) = \{r_i\}$.

▶ **Example 4** (Example 2 continued). Reason of instances of rules R_1 to R_{10} is $\{r_0\}$. At propagation step 1, $v(1)$ is added to IN with reason $\{r_1\}$. Then at choice step 7, when the rule $r_7 = (red(1) \leftarrow v(1), not\ green(1).)$ is chosen to be unblocked, $green(1)$ is added to OUT with reason $\{r_7\}$ and $red(1)$ is added to IN with reason $\{r_1, r_7\}$. If the rule r_7 was excluded instead of being chosen, the constraint $(\perp \leftarrow not\ green(1).)$ would be added to the K set with the reason $\{r_7\}$ (cf. Figure 1).

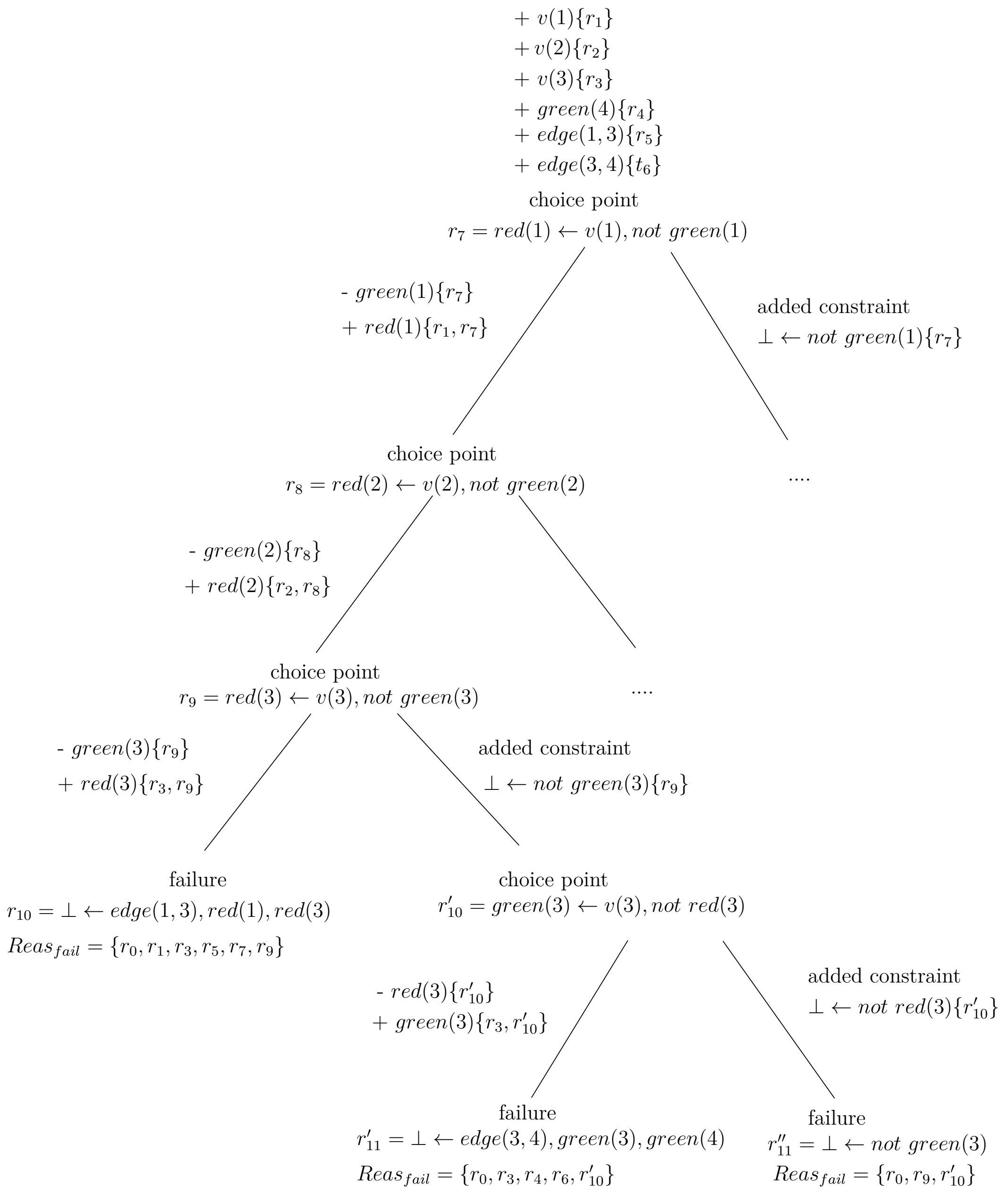

$+\ v(1)\{r_1\}$
$+\ v(2)\{r_2\}$
$+\ v(3)\{r_3\}$
$+\ green(4)\{r_4\}$
$+\ edge(1,3)\{r_5\}$
$+\ edge(3,4)\{t_6\}$
choice point
$r_7 = red(1) \leftarrow v(1), not\ green(1)$
$-\ green(1)\{r_7\}$
$+\ red(1)\{r_1, r_7\}$
added constraint
$\perp \leftarrow not\ green(1)\{r_7\}$
choice point
$r_8 = red(2) \leftarrow v(2), not\ green(2)$
....
$-\ green(2)\{r_8\}$
$+\ red(2)\{r_2, r_8\}$
choice point
$r_9 = red(3) \leftarrow v(3), not\ green(3)$
....
$-\ green(3)\{r_9\}$
$+\ red(3)\{r_3, r_9\}$
added constraint
$\perp \leftarrow not\ green(3)\{r_9\}$
failure
$r_{10} = \perp \leftarrow edge(1,3), red(1), red(3)$
$Reas_{fail} = \{r_0, r_1, r_3, r_5, r_7, r_9\}$
choice point
$r'_{10} = green(3) \leftarrow v(3), not\ red(3)$
$-\ red(3)\{r'_{10}\}$
$+\ green(3)\{r_3, r'_{10}\}$
added constraint
$\perp \leftarrow not\ red(3)\{r'_{10}\}$
failure
$r'_{11} = \perp \leftarrow edge(3,4), green(3), green(4)$
$Reas_{fail} = \{r_0, r_3, r_4, r_6, r'_{10}\}$
failure
$r''_{11} = \perp \leftarrow not\ green(3)$
$Reas_{fail} = \{r_0, r_9, r'_{10}\}$

Figure 1 Part of the search tree for the program P_2 of Example 2. At each node, left branch is (Rule choice) and right branch is (Rule exclusion). Each branch corresponds to a computation. Adding atom a to IN set with reason R is symbolized by $+\ a\ R$, and adding a to OUT set with reason R is symbolized by $-\ a\ R$.

4.1.2 Reasons of the undetermined atoms

In a computation, the interpretation $\langle IN, OUT\rangle$ may remain partial until the end of the sequence. If an atom a_0 is undetermined (i.e., not in the IN nor in the OUT sets), it is only known that a_0 cannot be proven. Intuitively, if a_0 cannot be proven it is because no ground rule concluding a_0 can be fired. Then, it has to be determined why a rule has never been fired along the sequence of a computation.

Let $S = \langle R_i, K_i, I_i\rangle_{i=0}^{k}$ be a computation prefix with $I_i = \langle IN_i, OUT_i\rangle$ and $R_i = \langle R_i^{app}, R_i^{excl}\rangle$, and r be a ground rule which has not been fired during the computation: $r \notin R_k^{app}$. The reason why r has not been fired may be: (i) There is an atom a from its positive body which is in the OUT_k set and then prevents the rule from being supported; or (ii) there is an atom a from its negative body which is in the IN_k set and then blocks the rule. For such an atom a, a reason why the rule r is not applicable is the reason why a belongs to the IN_k set (resp. OUT_k).

Another possible reason why r has not been fired may be that (iii) there is an atom a from its positive body which is undetermined, i.e., it belongs neither to the IN_k set nor to the OUT_k set and, again, prevents the rule from being supported. In this case, a reason why r is not applicable is the reason why a is undetermined.

Finally, if r has not been fired despite it was applicable, it means that the rule has been chosen for (Rule exclusion), and then blocked by adding a constraint to the K set. In this case, the reason why the rule r is not applicable is simply the reason of this constraint, i.e., this arbitrary choice to exclude r.

▶ **Example 5.** Let $I_k = \langle\{x\}, \{c, d\}\rangle$, a be a non provable atom, and $r_1 = (a \leftarrow y,\ not\ c.)$ and $r_2 = (a \leftarrow x,\ not\ b,\ not\ d.)$ be the only two ground rules concluding a.

Atom a is not provable because, firstly, r_1 is not supported (due to undetermined atom y) and thus cannot be fired, and, secondly, r_2 has not been fired despite it is applicable. Then r_2 has necessarily been chosen for (Rule exclusion) and thus blocked by adding the constraint $(\perp \leftarrow not\ b,\ not\ d.)$ to the K set. Finally, a has failed to be proven because the undetermined atom y prevents r_1 from being supported, and $(\perp \leftarrow not\ b,\ not\ d.)$ blocks r_2. The reason of undetermined atom a will be the union of the reason of undetermined atom y in S and $reason(\perp \leftarrow not\ b,\ not\ d., S)$.

In the following definition, a reason of an undetermined atom a is defined with respect to a sequence $T = \langle R_i, Atoms_i, Reas_i\rangle_{i=0}^{\infty}$. The idea is the following. For each i, $Atoms_i$ is the set of undetermined atoms for which a reason has to be defined, it is then initialized with $\{a\}$. For each ground rule r whose head is in $Atoms_i$, we have to determine a reason for which r has not been fired. At each step i, such a rule r_i is chosen, and a reason for which it has not been fired is determined and added to the $Reas_i$ set. If this reason involves another undetermined atom b, b is added to the $Atoms_i$ set. R_i is the set of ground rules already treated at step i, thus the sequence converges when all ground rules whose head is in $Atoms_i$ are treated.

If P is a program and a is an atom, $hrule(a, P) = \{r \in ground(P) \mid head(r) = a\}$.

▶ **Definition 6** (Reason of undetermined atoms)**.** Let P be a program, $S{=}\langle R_i, K_i, I_i\rangle_{i=0}^{n}$ be a computation prefix with $I_n = \langle IN_n, OUT_n\rangle$, and a be an undetermined atom: $a \notin IN_n \cup OUT_n$. A *reason of undetermined atom a*, denoted $reason_{und}(a, S)$, is defined with respect to a sequence $T = \langle R_i, Atoms_i, Reas_i\rangle_{i=0}^{\infty}$ where for each i, R_i is a set of ground rules, $Atoms_i$ is an atomset, and $Reas_i$ is a set of ground rules, that satisfies the following conditions:

- $R_0 = \emptyset$, $Atoms_0 = \{a\}$, $Reas_0 = \{r_0\}$
- $\forall i > 0$, $r_i \in \bigcup_{at \in Atoms_{i-1}} hrule(at, P) \setminus R_{i-1}$ and satisfies one of the following conditions:

  (i) $\exists l \in (body^+(r_i) \cap OUT_n)$,

  or (ii) $\exists l \in (body^-(r_i) \cap IN_n)$

  Then, $R_i = R_{i-1} \cup \{r_i\}$, $Atoms_i = Atoms_{i-1}$,
  $Reas_i = Reas_{i-1} \cup reason(l, S)$

  or (iii) $\exists l \in (body^+(r_i) \setminus (IN_n \cup OUT_n))$,

  Then, $R_i = R_{i-1} \cup \{r_i\}$, $Atoms_i = Atoms_{i-1} \cup \{l\}$, $Reas_i = Reas_{i-1}$

  or (iv) $\exists const \in K$ such that $const = (\bot \leftarrow \cup_{b \in body^-(r_i)} not\ b.)$
  and $reason(const, S) = \{r_i\}$

  Then, $R_i = R_{i-1} \cup \{r_i\}$, $Atoms_i = Atoms_{i-1}$,
  $Reas_i = Reas_{i-1} \cup reason(const, S)$

  or (v) $R_i = R_{i-1}$, $Atoms_i = Atoms_{i-1}$, $Reas_i = Reas_{i-1}$
- (Convergence) $\exists i \geq 0,\ R_i = \bigcup_{at \in Atoms_{i-1}} hrule(at, P)$

The sequence $T = \langle R_i, Atoms_i, Reas_i \rangle_{i=0}^{\infty}$ is said to converge with $Reas_\infty = \bigcup_{i=0}^{\infty} Reas_i$. Then, $reason_{und}(a, S) = Reas_\infty$.

4.2 Blocking sets

Each rule r from a reason *Reason* can be of three types according to what justifies r to belong to the reason: it can be into the reason in order to justify the truth of its head ($Reason_S^{decl}$), in this case r has been fired at a (Propagation) or (Rule choice) step and $r \in reason(head(r), S) \subseteq Reason$; or r can be into the reason in order to justify the falsity of an atom from its negative body ($Reason_S^{nblock}$), in this case r has been unblocked at a (Rule choice) step and, for $l \in body^-(r)$, $\{r\} = reason(l, S) \subseteq Reason$; or r can be into the reason in order to justify an undetermined atom ($Reason_S^{block}$), in this case r has been blocked at a (Rule exclusion) step by adding a constraint c and $\{r\} = reason(c, S) \subseteq Reason$.

▶ **Definition 7** (Reason types)**.** Let P be a program, $S = \langle K_i, R_i, I_i \rangle_{i=0}^{n}$ be a computation prefix for P with $R_i = \langle R_i^{app}, R_i^{excl} \rangle$, and *Reason* be a set of numbered rules.

- $Reason_S^{decl} = \{r_i \in Reason \cap R_n^{app} \mid reason(head(r_i), I_i) \subseteq Reason\}$
- $Reason_S^{nblock} = \{r_i \in Reason \cap R_n^{app} \mid reason(head(r_i), I_i) \nsubseteq Reason\}$
- $Reason_S^{block} = Reason \cap R_n^{excl}$

▶ **Example 8** (Example 4 continued)**.** Consider $Reason = \{r_7\}$, the reason of $green(1)$. $r_7 \in R^{app}$ and $r_7 \in R_S^{nblock}$ because $reason(head(r_7)) = reason(red(1)) = \{r_1, r_7\} \nsubseteq Reason$. But if we consider $Reason = \{r_1, r_7\}$, the reason of $red(1)$, $r_7 \in R^{app}$ and $r_7 \in R_S^{decl}$ because $reason(head(r_7)) \subseteq Reason$. In the first case, r_7 is the reason why the rule is unblocked while in the second case r_7 belongs to the reason because it is fired.

In a computation S which converges to an answer set $X = IN_\infty$, the set of fired rules R^{app} coincides with the generating rules of X, $GR_P(X)$, and $X = head(GR_P(X))$. A reason is a subset of the fired and excluded rules in a computation. For an answer set to be compatible with a reason it must be established that (1) the rules from R^{app} that belong to the reason because they are fired in the computation are generating rules of X, (2) the rules from R^{app} that belong to the reason because they are not blocked in the computation are not blocked w.r.t. X (regardless of whether or not supported w.r.t. X), (3) the rules from R^{excl} that belong to the reason are blocked w.r.t. X (again, regardless of whether or not supported).

▶ **Definition 9** (Compatible). Let P be a program, $S = \langle K_i, R_i, I_i\rangle_{i=0}^{k}$ be a computation prefix for P with $R_k = \langle R_k^{app}, R_k^{excl}\rangle$, and $Reas$ be a set of ground rules such that $Reas \subseteq R_k^{app} \cup R_k^{excl} \cup \{r_0\}$.

An atom set X is *compatible* with $Reas$ if $\begin{cases} Reas_S^{decl} \subseteq GR_P(X) \\ \forall r \in Reas_S^{nblock}, body^-(r) \cap X = \emptyset \\ \forall r \in Reas_S^{block}, body^-(r) \cap X \neq \emptyset \end{cases}$

The rules belonging to a reason are those responsible of the satisfaction of some property. For instance, the property could be that some atom a belongs to IN (resp. OUT), or that a is undetermined, or that the computation does not converge. For a reason to be a real reason of a computation property, each answer set X compatible with the reason must satisfy this property. For instance, Theorem 14 of Section 4.3.1 says that each answer set X compatible with the reason of an undetermined atom a (Def. 6) verifies that $a \notin X$. If there is no answer set compatible with some reason $Reas$, then we say that $Reas$ is a blocking set.

▶ **Definition 10** (Blocking set). Let P be a program, $S = \langle K_i, R_i, I_i\rangle_{i=0}^{k}$ be a computation prefix for P with $R_k = \langle R_k^{app}, R_k^{excl}\rangle$ and $Reas$ be a set of ground rules such that $Reas \subseteq R_k^{app} \cup R_k^{excl} \cup \{r_0\}$. $Reas$ is a *blocking set* if there is no answer set X for P compatible with $Reas$.

▶ **Example 11** (Example 8 continued). Consider the sequence $S = \langle K_i, R_i, I_i\rangle_{i=0}^{9}$ with $R_9^{app} = \{r_1, \ldots, r_9\}$. $Reas = \{r_1, r_3, r_5, r_7, r_9\}$ is a blocking set: $Reas \subseteq Reas_S^{decl}$ and there is no answer set X such that $Reas \subseteq GR_P(X)$ since if r_7 and r_9 are generating rules, vertices 1 and 3 are red and the constraint ($\bot \leftarrow edge(1,3), red(1), red(3).$) is a generating rule too. Note that r_0 has no impact on blocking sets and thus $Reas \cup \{r_0\}$ is a blocking set too.

4.3 Failures

In a concrete calculation of answer sets, the search process can be represented by a search tree where the nodes are "guesses" about supported rules to be unblocked (Rule choice) or blocked (Rule exclusion). In such a tree, each branch corresponds to a computation prefix which converges (success branch) or not (failure branch). In case of failure, some backtrack must be done in order to explore another branch. Figure 1 illustrates a part of the search tree of Example 2.

The failures presented here correspond to the computations (branches) that do not converge. The first case, *blocked prefix*, corresponds to branches where no revision is available. The second case, *failure combination*, corresponds to a node where the two branches fail.

4.3.1 Blocked computations

A computation prefix is *blocked* if the computation does not converge and the only possible revision is stability. A computation can be blocked in two cases: either a contradiction is detected by the (Propagation) step (there is a rule r_i which is supported and unblocked but it cannot be fired because its head is already into the OUT set[3]), either there is no more applicable rule but there is at least a non satisfied constraint (i.e. supported and not blocked). Note that all other applicable rules can be chosen by (Rule choice) or, if the head is already in the OUT set, by (Rule exclusion). Thus, a computation cannot be blocked due to an applicable rule other than a constraint.

[3] For the computation to be blocked, we impose in Definition 12 that all rules from Δ_{pro} cannot be fired. But in practice it suffices than one rule from Δ_{pro} cannot be fired in order to ensure the failure.

▶ **Definition 12** (Blocked Prefix and Failure Reason)**.** Let P be a program, $S = \langle K_i, R_i, I_i\rangle_{i=0}^{k}$ be a prefix of a computation for P with $R_i = \langle R_i^{app}, R_i^{excl}\rangle$ and $I_i = \langle IN_i, OUT_i\rangle$. S is said to be *blocked* if S satisfies one of the following conditions. In each case, a *failure reason* due to a ground rule rf, noted $reason_{fail}(S, rf)$, is defined.

(Propagation failure) $\Delta_{pro}(P \cup K_k, I_k, R_k^{app}) \neq \emptyset$
$\forall r \in \Delta_{pro}(P \cup K_k, I_k, R_k^{app}), head(r) \in OUT_k$
$reason_{fail}(S, rf) = \bigcup_{a \in body(rf)} reason(a, S) \ \cup \ reason(rf, S) \ \cup \ reason(head(rf), S)$
with $rf \in \Delta_{pro}(P \cup K_k, I_k, R_k^{app})$

or (Non satisfied constraint) $\Delta_{pro}(P \cup K_k, I_k, R_k^{app}) = \emptyset$,
$\Delta_{cho}(P, I_k, R_k^{app} \cup R_k^{excl}) = \emptyset$, $\Delta_{cho}(K_k, I_k, R_k^{app} \cup R_k^{excl}) \neq \emptyset$
$reason_{fail}(S, rf) = reason(rf, S) \cup$
$\bigcup_{a \in (body^-(rf) \cap OUT_k)} reason(a, S) \ \cup \ \bigcup_{a \in (body^-(rf) \setminus OUT_k)} reason_{und}(a, S)$
with $rf \in \Delta_{cho}(K_k, I_k, R_k^{app} \cup R_k^{excl})$

In the first case, there exists at least a rule $rf \in \Delta_{pro}(P \cup K_k, I_k, R_k^{app})$ with $head(r) \in OUT_k$, the reason of the contradiction is the reason why rf is supported and unblocked, the reason of the rule itself (which is $\{r_0\}$ if it is not a constraint added by (Rule Exclusion)) and the reason why $head(rf)$ is in the OUT_k set.

In the second case, there exists at least a non satisfied constraint c, the reason of this failure is the set of reasons which make the constraint not blocked (such a constraint from K set has an empty positive body). Note that the constraint is not blocked ($body^-(c) \cap IN_k = \emptyset$) but not unblocked ($body^-(c) \not\subseteq OUT_k$) otherwise it would have been fired during the propagation step. Hence, there is at least an atom in the negative body whose status remained undetermined.

▶ **Example 13** (Example 2 continued)**.** The sequence $S = \langle K_i, R_i, I_i\rangle_{i=0}^{9}$ is a blocked prefix (see the left most branch of the tree of Figure 1): $(R_9, \{X \leftarrow 1, Y \leftarrow 3\}) \in \Delta_{pro}(P_2, I_9, \{r_1, \cdots, r_9\})$ and $r_{10} = (\bot \leftarrow edge(1,3), red(1), red(3).)$ but $head(r_{10}) = \bot \in OUT$. $reason_{fail}(S, r_{10}) = reason(edge(1,3), S) \cup reason(red(1), S) \cup reason(red(3), S) \cup reason(r_{10}, S) \cup reason(\bot, S) = \{r_0, r_1, r_3, r_5, r_7, r_9\}$.

Properties of a blocked computation prefix can then be established. The following theorem says that a reason RU of an undetermined atom a (see Def. 6) is a real justification of the non provability of a: a cannot belong to an answer set compatible with RU.

▶ **Theorem 14.** *Let P be a program, $S = \langle K_i, R_i, I_i\rangle_{i=0}^{k}$ be a blocked computation prefix for P, a be an atom such that $a \notin (IN_k \cup OUT_k)$, and RU be a reason of the undetermined atom a. For all answer set X compatible with RU, $a \notin X$.*

▶ **Example 15.** Consider the branch leading to the third leaf of the tree of Figure 2. $green(3)$ is undetermined because of r'_{10} (Rule exclusion), and $reason_{und}(green(3)) = \{r_0, r'_{10}\}$. Theorem 14 guarantees that for all answer set X such that r'_{10} is blocked, $green(3) \notin X$.

The failure reason RF of a blocked computation prefix is also a real justification of failure: there is no answer set compatible with RF.

▶ **Theorem 16.** *Let P be a program, S be a blocked prefix of a computation and RF be a failure reason for S. RF is a blocking set.*

▶ **Example 17** (Example 13 continued)**.** $reason_{fail}(S, r_{10}) = \{r_0, r_1, r_3, r_5, r_7, r_9\}$ is a blocking set (see Example 11). This means that the corresponding steps of the computation are those responsible of the failure. Other revision steps, for instance the (Propagation) $r_4 = (green(4).)$ or the (Rule choice) $r_8 = (red(2) \leftarrow v(2), not\ green(2).)$ have nothing to do with this failure.

4.3.2 Failure combination

We call *choice points* the steps of a computation S where (Rule choice) or (Rule exclusion) are used. In practice, they correspond to nodes in a search tree. If r_i and r_j are numbered rules, $r_i < r_j$ iff $i < j$. If R is a set of numbered rules, $max(R) = r$ iff for all $r_i \in R, r_i \leq r$, and $R_{<r_i} = \{r \in R \mid r < r_i\}$. If P is a program, $S = \langle K_i, R_i, I_i\rangle_{i=0}^{n}$ is a computation prefix for P with $R_i = \langle R_i^{app}, R_i^{excl}\rangle$, and $Reason$ is a set of numbered rules, $choicePoints(S) = \{r_i \in R_i^{app} \cup R_i^{excl} \mid i \in [1 \ldots n], \Delta_{pro}(P \cup K_{i-1}, I_{i-1}, R_{i-1}^{app}) = \emptyset\}$ are the ground rules r_i used for choice points in S, $choicePoints(Reason, S) = Reason \cap choicePoints(S)$ is the restriction of the preceding set to the rules belonging to some reason $Reason$ and $lastChoicePoint(Reason, S) = max(choicePoints(Reason, S))$ is the last rule (the rule with the greatest number) from $Reason$ used for a choice point in S.

Suppose two computations that do not converge and that are the same up to a choice point lc. At this step, the computations differ: one uses rule r_{lc} for (Rule choice) and the other use the same rule for (Rule exclusion). If r_{lc} is the last choice point involved in the two failure reasons, then we can conclude that this rule r_{lc} is not implicated in the failure: the rule can be applied or excluded (thus, to be a generating rule or not), the computation fails in both cases. So, the failure exists prior to this choice point lc. A new failure reason can be defined by joining the two failure reasons and restricting them to the rules involved in the computation at a step preceding lc. If the greatest numbered rule of this new failure reason is r_k, then we can conclude that the computation prefix ending at step k will fail too: it is a failure prefix.

▶ **Definition 18** (Failure Prefix and Failure Reason)**.** Let P be a program. A *failure prefix* of a computation prefix for P and a failure reason for this prefix are defined as follows:

1. A blocked prefix is a failure prefix and its failure reason is defined as in Definition 12.
2. Let $S_1 = \langle K_{1i}, R_{1i}, I_{1i}\rangle_{i=0}^{n}$ and $S_2 = \langle K_{2i}, R_{2i}, I_{2i}\rangle_{i=0}^{m}$ be two failure prefix of computation for P with $R_{1i} = \langle R_{1i}^{app}, R_{1i}^{excl}\rangle$ and $R_{2i} = \langle R_{2i}^{app}, R_{2i}^{excl}\rangle$ and let RF_1 and RF_2 be two failure reasons for, respectively, S_1 and S_2 such that:
   - $lastChoicePoint(RF_1, S_1) = lastChoicePoint(RF_2, S_2) = r_{lc}$
   - $\langle S_1\rangle_{i=0}^{lc-1} = \langle S_2\rangle_{i=0}^{lc-1}$
   - $r_{lc} \in R_{1lc}^{app}$ and $r_{lc} \in R_{2lc}^{excl}$

   Let $reasonFail = RF_{1<r_{lc}} \cup RF_{2<r_{lc}}$ and $r_k = max(reasonFail)$.
   $\langle S_1\rangle_{i=0}^{k} = \langle S_2\rangle_{i=0}^{k}$ is a failure prefix and $reasonFail$ is a *failure reason* for this prefix.

The following theorem establishes that a failure reason of a failure prefix is a real justification of failure: no answer set is compatible with it. A corollary is that a failure prefix cannot be extended to a computation which converges.

▶ **Theorem 19.** *Let P be a program, S be a failure prefix of a computation and RF be a failure reason for S. Then, RF is a blocking set.*

▶ **Theorem 20.** *Let P be a program and S be a failure prefix of a computation for P. For any computation S' with prefix S, S' does not converge.*

▶ **Example 21** (Example 2 continued)**.** Suppose that at step 9 (cf. Fig. 1), the rule r_9 is excluded instead of being chosen. Then the constraint $(\bot \leftarrow not\ green(3).)$ is added to K_9 with the reason $\{r_9\}$. Step 10 can be the choice of the rule $r'_{10} = (green(3) \leftarrow v(3), not\ red(3).)$; in this case $r'_{11} = (\bot \leftarrow edge(3,4), green(3), green(4).) \in \Delta_{pro}(P_2 \cup K_{10}, I_{10}, \{r_1, \cdots, r_{10}\})$ and $reason_{fail}(S', r'_{11}) = \{r_0, r_3, r_4, r_6, r'_{10}\}$. If r'_{10} is excluded at step 10, the constraint $(\bot \leftarrow not\ red(3).)$ is added to K_{10} with the reason $\{r'_{10}\}$. There are non satisfied constraints, for instance $r''_{11} = (\bot \leftarrow not\ green(3).)$. $green(3)$ is undefined with reason $\{r_0, r'_{10}\}$ and $reason_{fail}(S'', r''_{11}) = \{r_0, r_9, r'_{10}\}$. The combination of the two failures leads to $reason_{fail}(S', r'_{10}) = \{r_0, r_3, r_4, r_6, r_9\}$. At the preceding choice point (r_9), a new combination of failures leads to $reason_{fail}(S, r_9) = \{r_0, r_1, r_3, r_4, r_5, r_6, r_7\}$.

4.3.3 Application to Backjumping

Our approach has already been used for backjumping in the solver `ASPeRiX`: failure reasons are computed during the solving process and permit to jump to the last choice point related to the failure instead of a simple chronological backtrack.

The nodes of the search tree correspond to choice points where an instantiated rule is chosen to be applied (left branch) or to be blocked (right branch). In case of failure, a reason of the failure is computed in order to know the nodes implicated in the failure and to avoid visiting sub-trees where the same failure will necessarily occur again. A failure on a leaf of the tree correspond to a blocked prefix of a computation (see Definition 12 and Theorem 16). When the left and right branches of a node both fail, their failure reasons can be combined to determine the reason of the failure of the sub-tree. This corresponds to a failure prefix of a computation (see Definition 18 and Theorem 19). When the combination of failures permits backjumping, Theorem 20 guarantees that the jumped sub-trees cannot lead to an answer set. For instance at step 9 of Figure 1, $reason_{fail}(S, r_9) = \{r_0, r_1, r_3, r_4, r_5, r_6, r_7\}$ (see Example 21), the choice point 8 is not responsible of the failure and the right branch of this node can be safely jumped.

In practice the reason defined above, constituted by the ground rules responsible of the failure, are too detailed. To each rule is then associated the node (choice point) in the tree where it is used. So the computed reasons are only sets of choice points (instead of sets of rules), and suffice to know the last choice responsible of the failure. For instance, the above failure reason will become $\{0, 7\}$ where 0 represents all revisions done before the first choice point.

The implementation is only a prototype that uses Prolog to build the ground rules necessary to compute the reason of undetermined atoms. Indeed, the set of ground rules whose head is a given atom cannot be easily computed in our approach where the rules are first order ones. The backward chaining of Prolog can solve the problem, although not very effectively. But it shows a drastic reduction of the number of choice points for some programs. This will be developed in another paper.

5 Conclusion

Notions of justifications and blocking sets for rule-based answer set computations have been presented with theorems establishing that justifications are “true” ones. These justifications meet those of other related approaches. Each of them views justifications in a particular perspective. Our viewpoint is that of a rule-based computation where a reason is a set of ground rules with particular properties. Our justifications are comparable to on-line justifications of [16]: they can be defined during the computation process. A difference is that we define justifications for undetermined atoms; in atom-based approaches, all atoms are determined at the end of a computation.

These justifications have already been used for backjumping in the solver `ASPeRiX` Other direct applications are learning, interactive debugging and explanation of answers in diagnosis systems. A preliminary approach on learning in a rule-based system is proposed in [19]. An important problem is that learned rules are generally constraints and that constraints are not used for propagation in rule-based solvers. So progress must be done for better treating constraints and/or learning rules that are not constraints. For debugging, the rule-based approach seems interesting because it allows stepping the search of answer sets. For instance, [14] proposes such an approach of stepping with a notion of computation closed to ours. Blocking sets could also explain absence of solution and help to propose repairs.

References

1 C. V. Damásio, A. Analyti, and G. Antoniou. Justifications for logic programming. In *LPNMR 2013,*, pages 530–542, 2013.

2 C. V. Damásio, J. Moura, and A. Analyti. Unifying justifications and debugging for answer-set programs. In *ICLP 2015*, 2015.

3 M. Dao-Tran, T. Eiter, M. Fink, G. Weidinger, and A. Weinzierl. OMiGA: An open minded grounding on-the-fly answer set solver. In *JELIA 2012*, pages 480–483, 2012.

4 T. Eiter, M. Fink, P. Schüller, and A. Weinzierl. Finding explanations of inconsistency in multi-context systems. In *KR 2010*, 2010.

5 M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Conflict-driven answer set solving. In *IJCAI 2007*, pages 386–392, 2007.

6 M. Gebser, J. Pührer, T. Schaub, and H. Tompits. A meta-programming technique for debugging answer-set programs. In *AAAI 2008*, pages 448–453, 2008.

7 M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Logic Programming, Proceedings of the Fifth International Conference and Symposium*, pages 1070–1080, 1988.

8 K. Konczak, T. Linke, and T. Schaub. Graphs and colorings for answer set programming. *Theory and Practice of Logic Programming*, 6:61–106, 1 2006. `doi:10.1017/S1471068405002528`.

9 C. Lefèvre, C. Béatrix, I. Stéphan, and L. Garcia. Asperix, a first order forward chaining approach for answer set computing. *CoRR*, abs/1503.07717:(to appear in TPLP), 2015. URL: `http://arxiv.org/abs/1503.07717`.

10 C. Lefèvre and P. Nicolas. A first order forward chaining approach for answer set computing. In *LPNMR 2009*, pages 196–208, 2009.

11 C. Lefèvre and P. Nicolas. The first version of a new ASP solver : `ASPeRiX`. In *LPNMR 2009*, pages 522–527, 2009.

12 N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, 7(3):499–562, 2006. `doi:10.1145/1149114.1149117`.

13 L. Liu, E. Pontelli, T. C. Son, and M. Truszczynski. Logic programs with abstract constraint atoms: The role of computations. *Artificial Intelligence*, 174(3-4):295–315, 2010. `doi:10.1016/j.artint.2009.11.016`.

14 J. Oetsch, J. Pührer, and H. Tompits. Stepping through an answer-set program. In *LPNMR 2011*, pages 134–147, 2011.

15 A. Dal Palù, A. Dovier, E. Pontelli, and G. Rossi. Answer set programming with constraints using lazy grounding. In *ICLP 2009*, 2009.

16 E. Pontelli, T. C. Son, and O. El-Khatib. Justifications for logic programs under answer set semantics. *Theory and Practice of Logic Programming*, 9(1):1–56, 2009. `doi:10.1017/S1471068408003633`.

17 C. Schulz and F. Toni. Justifying answer sets using argumentation. *Theory and Practice of Logic Programming*, 16(1):59–110, 2016. `doi:10.1017/S1471068414000702`.

18 P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1-2):181–234, 2002. `doi:10.1016/S0004-3702(02)00187-X`.

19 A. Weinzierl. Learning non-ground rules for answer-set solving. In *2nd Workshop on Grounding and Transformations for Theories with Variables, GTTV 2013*, 2013.

Intelligent Instantiation and Supersafe Rules*

Vladimir Lifschitz

Department of Computer Science, University of Texas at Austin, Austin, TX, USA
`vl@cs.utexas.edu`

Abstract

In the input languages of most answer set solvers, a rule with variables has, conceptually, infinitely many instances. The primary role of the process of intelligent instillation is to identify a finite set of ground instances of rules of the given program that are "essential" for generating its stable models. This process can be launched only when all rules of the program are safe. If a program contains arithmetic operations or comparisons then its rules are expected to satisfy conditions that are even stronger than safety. This paper is an attempt to make the idea of an essential instance and the need for "supersafety" in the process of intelligent instantiation mathematically precise.

1998 ACM Subject Classification D.3.1 Formal Definitions and Theory

Keywords and phrases answer set programming

Digital Object Identifier 10.4230/OASIcs.ICLP.2016.7

1 Introduction

The input languages of most answer set solvers are not typed. When a program in such a language is grounded, the variables occurring in it can be replaced by arbitrary ground terms not containing arithmetic operations, and that includes arbitrary integers.[1] Thus the set of ground instances of any non-ground rule is infinite. The primary role of the process of intelligent instantiation is to identify a finite set of ground instances of the rules of the program that are "essential" for generating its stable models.

The possibility of intelligent instantiation is predicated on the assumption that every rule of the given program is safe – that each variable occurring in the rule appears also nonnegated in its body. If an unsafe rule is found in the program then the solver produces an error message and stops. For example, generating the stable models of a program will not be attempted if it contains the rule

$$p(X, Y) \leftarrow q(X),$$

because the variable Y occurs in its head but not in the body.

The safety assumption does not guarantee that the set of essential instances is finite. For example, all rules of the program

$$\begin{array}{l} p(a), \\ p(b), \\ p(f(f(X))) \leftarrow p(X) \end{array} \qquad (1)$$

* This work was supported in part by the National Science Foundation under Grant IIS-1422455.

[1] The language SPARC [1] is a notable exception. In a SPARC program, finite sorts are assigned to arguments of all predicates, and the range of integers allowed in the process of grounding is finite.

are safe, but its third rule has infinitely many instances essential for constructing the stable model:

$$\begin{array}{ll} p(f(f(a))) \leftarrow p(a), & p(f(f(b))) \leftarrow p(b), \\ p(f(f(f(f(a))))) \leftarrow p(f(f(a))), & p(f(f(f(f(b))))) \leftarrow p(f(f(b))), \\ \multicolumn{2}{c}{\dots} \end{array}$$

The rules in the first line are essential because their bodies $p(a)$ and $p(b)$ are facts from (1). The rules in the second line are essential because their bodies are identical to the heads of the rules in the first line, and so on. An attempt to form all essential instances will add a finite set of rules at each step, but it will not terminate. In the terminology of Calimeri et al. [4], program (1) is not finitely ground. The safety of all rules does guarantee, however – for programs containing neither arithmetic operations nor comparisons – that all essential instances can be found in a stepwise manner, as in the example above, with finitely many instances added at every step.

In the presence of arithmetic operations and comparisons, on the other hand, the possibility of launching the process of accumulating essential instances is not ensured by the safety of all rules. For example, each of the rules

$$p(X,Y) \leftarrow q(X+Y), \tag{2}$$
$$p(X,Y) \leftarrow X < Y, \tag{3}$$
$$p(X,Y) \leftarrow X = Y \tag{4}$$

is safe in the sense that both variables occurring in it appear nonnegated in the body. But the presence of any of these rules in a program causes the grounder GRINGO to stop execution with the same error message as in the presence of an unsafe rule. On the other hand, GRINGO does not object against the rules

$$p(X,Y) \leftarrow X = Y,\, q(X) \tag{5}$$

and

$$p(X) \leftarrow X + 3 = 4. \tag{6}$$

The discussion of safety in Version 2.0 of the Potassco User Guide (`http://sourceforge.net/projects/potassco/files/guide/`) shows that the conditions under which GRINGO treats a rule as safe are quite complicated.[2] Such conditions have to be imposed because safety in the traditional sense does not guarantee the possibility of calculating essential instances in the step-by-step manner; the rules of the program must be "supersafe."

This paper shows how our informal discussion of essential instances, of the role of intelligent instantiation, and of the need for supersafety can be made mathematically precise. In the next section we define which elements of a set Γ of ground rules are essential and

[2] According to that document, occurrences of variables in the scope of arithmetic functions can only justify safety for "simple arithmetic terms" – terms containing a single occurrence of a variable and no arithmetic operations other than addition, subtraction, and multiplication. This explains why GRINGO does not accept rule (2) as safe: the term $X+Y$ is not simple. Moreover, if multiplication is used, then the constant factor must not evaluate to 0 for the variable occurrence to justify safety. Furthermore, according to the User Guide, safety is not justified by occurrences of variables in inequalities; hence (3) is not accepted. This restriction does not apply to equalities. "However, this only works when unification can be made directionally, i.e., it must be possible to instantiate one side without knowing the values of variables on the other side." This explains why GRINGO considers rule (4) unsafe but accepts (5) and (6).

prove that the set $E(\Gamma)$ of essential rules has the same stable models as the whole Γ. The set $E(\Gamma)$ is defined as the union of a monotone sequence of subsets $E_k(\Gamma)$, representing the stepwise process of accumulating essential rules. After describing a class of logic programs with variables and arithmetic in Section 3, we define and study the concept of a supersafe rule (Section 4). The main result of this paper, proved in Section 5, shows that if Γ is the propositional image of a program consisting of supersafe rules then each of the sets $E_k(\Gamma)$ is finite. This theorem clarifies the role of the additional conditions that GRINGO imposes on safe rules.

2 Essential Rules

2.1 Propositional Programs

We start by describing the class of programs without variables for which the concept of an essential rule will be defined.[3] Consider a fixed propositional signature – a set of symbols called *atoms*. (In applications to the study of logic programs with variables and arithmetic, the signature will consist of the ground atoms not containing arithmetic operations.) A *(propositional) rule* is an expression of the form $H \leftarrow B$, where the *head* H and the *body* B are propositional formulas formed from atoms and the symbols $\top$ (true) and $\bot$ (false) using the connectives $\wedge$, $\vee$, $\neg$.

A *(propositional) program* is a set of rules.

A set M of atoms will be identified with the truth assignment that maps all elements of M to *true* and all other atoms to *false*. The *reduct* R^M of a rule R relative to M is the rule obtained by replacing, in the head and in the body of R, each subformula F that begins with negation and is not in the scope of negation with $\top$ if M satisfies F, and with $\bot$ otherwise. The reduct Γ^M of a program Γ is defined as the set of the reducts R^M of all rules R of Γ. We say that M is a *stable model* of a program Γ if M is minimal among the sets satisfying Γ^M.

2.2 Essential Part of a Propositional Program

Consider a propositional program Γ such that the body of every rule of Γ is a conjunction of formulas of three types: (a) symbols $\top$ and $\bot$; (b) atoms; (c) formulas beginning with negation. In the definition of the essential part of Γ below, the following terminology is used. A *nonnegated atom* of a propositional formula F is an atom A such that at least one occurrence of A in F is not in the scope of negation. A rule from Γ is *trivial* if at least one of the conjunctive terms of its body is $\bot$.

The subsets $E_0(\Gamma)$, $E_1(\Gamma), \dots$ of Γ are defined as follows:

- $E_0(\Gamma) = \emptyset$,
- $E_{k+1}(\Gamma)$ is the set of all nontrivial rules R of Γ such that every nonnegated atom of the body of R is also a nonnegated atom of the head of some rule from $E_k(\Gamma)$.

It is clear that every member of the sequence $E_0(\Gamma), E_1(\Gamma), \dots$ is a subset of the one that follows (by induction). It is clear also that if $E_{k+1}(\Gamma) = E_k(\Gamma)$ then $E_l(\Gamma) = E_k(\Gamma)$ for all l that are greater than k.

The set of *essential rules* of Γ, denoted by $E(\Gamma)$, is defined as the union $\bigcup_{k\geq 0} E_k(\Gamma)$. The *degree* of an essential rule R is the smallest k such that $R \in E_k(\Gamma)$.

[3] Programs considered here are programs with nested expressions [7] without classical negation, with the usual symbols for propositional connectives used instead of the comma, the semicolon, and "*not*" in the original publication.

▶ **Theorem 1.** *Programs Γ and $E(\Gamma)$ have the same stable models.*

▶ **Example 2.** If the rules of Γ are

$$\begin{aligned}&a_1 \vee a_2 \leftarrow \neg a_0,\\ &\qquad b_n \leftarrow a_n \wedge a_{n+1} \qquad (n \geq 0)\end{aligned}$$

then

$$\begin{aligned}&E_0(\Gamma) = \emptyset,\\ &E_1(\Gamma) = \{a_1 \vee a_2 \leftarrow \neg a_0\},\\ &E_2(\Gamma) = \{a_1 \vee a_2 \leftarrow \neg a_0,\ b_1 \leftarrow a_1 \wedge a_2\},\end{aligned}$$

and $E_3(\Gamma) = E_2(\Gamma)$. It follows that Γ has two essential rules: rule $a_1 \vee a_2 \leftarrow \neg a_0$ of degree 1 and rule $b_1 \leftarrow a_1 \wedge a_2$ of degree 2. The program consisting of these two rules has the same stable models as Γ: $\{a_1\}$ and $\{a_2\}$.

▶ **Example 3.** If the rules of Γ are

$$\begin{aligned}&a_1 \leftarrow \top,\\ &a_{2n} \leftarrow a_n \qquad (n \geq 0)\end{aligned}$$

then

$$\begin{aligned}&E_0(\Gamma) = \emptyset,\\ &E_1(\Gamma) = \{a_1 \leftarrow \top\},\\ &E_2(\Gamma) = \{a_1 \leftarrow \top,\ a_2 \leftarrow a_1\},\\ &E_3(\Gamma) = \{a_1 \leftarrow \top,\ a_2 \leftarrow a_1,\ a_4 \leftarrow a_2\},\\ &\quad \dots,\end{aligned}$$

so that

$$E(\Gamma) = \{a_1 \leftarrow \top\} \cup \{a_{2^{k+1}} \leftarrow a_{2^k} \ : \ k \geq 0\}.$$

The set of essential rules in this case is infinite, but for every positive k the program has only one essential rule of degree k. The set $E(\Gamma)$ has the same stable model as Γ: $\{a_1, a_2, a_4, \dots\}$.

▶ **Example 4.** If the rules of Γ are

$$\begin{aligned}&a_0 \leftarrow \top,\\ &b_{m,n} \leftarrow a_{n-m} \qquad (n \geq m \geq 0)\end{aligned}$$

then

$$\begin{aligned}&E_0(\Gamma) = \emptyset,\\ &E_1(\Gamma) = \{a_0 \leftarrow \top\},\\ &E_2(\Gamma) = \{a_0 \leftarrow \top\} \cup \{b_{n,n} \leftarrow a_0 \ : \ n \geq 0\},\end{aligned}$$

and $E_3(\Gamma) = E_2(\Gamma)$. It follows that Γ has infinitely many essential rules: rule $a_0 \leftarrow \top$ of degree 1 and rules $b_{n,n} \leftarrow a_0$, for all n, of degree 2. The program consisting of these rules has the same stable model as Γ: $\{a_0, b_{0,0}, b_{1,1}, \dots\}$.

In Section 5 we will apply the concept of an essential rule to "propositional images" of programs with variables and arithmetic operations, and we will see that the behaviors observed in the examples above correspond to programs of three types: (1) programs for which the process of intelligent instantiation terminates; (2) programs for which this process can be launched but does not terminate; and (3) programs for which this process cannot be even launched. We will see also that if every rule of a program is supersafe then the program cannot belong to the third group.

2.3 Proof of Theorem 1

▶ **Lemma 5.** *Let Δ be a subset of a propositional program Γ, and let H be the set of all nonnegated atoms of the heads of the rules of Δ. If the body of every nontrivial rule of $\Gamma \setminus \Delta$ contains a nonnegated atom that does not belong to H then Γ and Δ have the same stable models.*

Proof. Consider first the case when the rules of Γ do not contain negation. We need to show that Γ and Δ have the same minimal models. Assume that M is a minimal model of Δ. Then $M \subseteq H$, so that the body of every nontrivial rule of $\Gamma \setminus \Delta$ contains a nonnegated atom that does not belong to M. It follows that M satisfies all rules of $\Gamma \setminus \Delta$, so that M is a model of Γ, and consequently a minimal model of Γ. In the other direction, assume than M is a minimal model of Γ. To show that M is minimal even among the models of Δ, consider a subset M' of M that satisfies all rules of Δ. Then $M' \cap H$ satisfies all rules of Δ as well, so that every nontrivial rule of $\Gamma \setminus \Delta$ contains a nonnegated atom that does not belong to $M' \cap H$. It follows that this set satisfies all rules of $\Gamma \setminus \Delta$, so that it is a model of Γ. Since it is a subset of a minimal model M of Γ, we can conclude that $M' \cap H = M$. Since M' is a subset of M, it follows that $M' = M$.

If some rules of Γ contain negation then consider the reducts Γ^M of Γ and Δ^M of Δ with respect to the same set M of atoms. It is clear that Δ^M is a subset of Γ^M, that H is the set of all nonnegated atoms of the heads of the rules of Δ^M, and that the body of every nontrivial rule of $\Gamma^M \setminus \Delta^M$ contains a nonnegated atom that does not belong to H. Furthermore, the rules of Γ^M do not contain negation. It follows, by the special case of the lemma proved earlier, that Γ^M and Δ^M have the same minimal models. In particular, M is a minimal model of Γ^M iff M is a minimal model of Δ^M. In other words, M is a stable model of Γ iff M is a stable model of Δ. ◀

To prove the theorem, consider the set H of all nonnegated atoms of the heads of the rules of $E(\Gamma)$. We will show that the body of every nontrivial rule of $\Gamma \setminus E(\Gamma)$ contains a nonnegated atom that does not belong to H; then the assertion of the theorem will follow from the lemma. Assume that R is a nontrivial rule of Γ such that all nonnegated atoms in the body of R belong to H. Then each of these atoms A is a nonnegated atom of the head of a rule that belongs to $E_k(\Gamma)$ for some k. This k can be chosen uniformly for all these atoms A: take the largest of the values of k corresponding to all nonnegated atoms in the head of R. Then R belongs to $E_{k+1}(\Gamma)$, and consequently to $E(\Gamma)$.

3 Programs with Variables and Arithmetic

The programming language defined in this section is a subset of the "Abstract Gringo" language AG [5]. The meaning of a program in this language is characterized by means of a transformation, denoted by τ, that turns rules and programs into their propositional images

– propositional programs in the sense of Section 2.1. The stable models of a program are defined as the stable models of its propositional image.[4]

3.1 Syntax

We assume that three disjoint sets of symbols are selected – *numerals*, *symbolic constants*, and *variables* – which do not contain the symbols

$$+ \quad - \quad \times \quad / \quad .. \tag{7}$$

$$= \quad \neq \quad < \quad > \quad \leq \quad \geq \tag{8}$$

$$\mathit{not} \quad \wedge \quad \vee \quad , \quad (\quad) \tag{9}$$

(The symbol .. is used to represent intervals.) We assume that a 1–1 correspondence between the set of numerals and the set $\mathbf{Z}$ of integers is chosen. The numeral corresponding to an integer n will be denoted by $\overline{n}$.

Terms are defined recursively, as follows:

- all numerals, symbolic constants, and variables are terms;
- if f is a symbolic constant and $\mathbf{t}$ is a tuple of terms separated by commas then $f(\mathbf{t})$ is a term;
- if t_1 and t_2 are terms and $\star$ is one of the symbols (7) then $(t_1 \star t_2)$ is a term.

An *atom* is an expression of the form $p(\mathbf{t})$, where p is a symbolic constant and $\mathbf{t}$ is a tuple of terms separated by commas.

A term or an atom is *precomputed* if it contains neither variables nor symbols (7). We assume a total order on precomputed terms such that for any integers m and n, $\overline{m} \leq \overline{n}$ iff $m \leq n$.

For any atom A, the expressions A, *not* A, *not not* A are *literals*. A *comparison* is an expression of the form $t_1 \prec t_2$ where t_1, t_2 are terms and $\prec$ is one of the symbols (8).

A *rule* is an expression of the form

$$H_1 \vee \cdots \vee H_k \leftarrow B_1 \wedge \cdots \wedge B_m \tag{10}$$

or a "choice rule" of the form

$$\{A\} \leftarrow B_1 \wedge \cdots \wedge B_m \tag{11}$$

$(k, m \geq 0)$, where each H_i and each B_j is a literal or a comparison, and A is an atom. A *program* is a set of rules.

A rule or another syntactic expression is *ground* if it does not contain variables. A rule (10) or (11) is *safe* if each variable occurring in it appears also in one of the expressions B_j which is an atom or a comparison.

3.2 Propositional Image of a Program

The signature of the propositional program $\tau\Pi$, defined below, is the set of precomputed atoms.

[4] Gebser et al. [5] write rules $H \leftarrow B$ of $\tau\Pi$ as implications $B \rightarrow H$. More importantly, H and B are allowed in that paper to contain implications and infinitely long conjunctions and disjunctions. This additional generality is not needed here because the programs that we study contain neither conditional literals nor aggregates.

3.2.1 Semantics of Ground Terms

Every ground term t represents a finite set $[t]$ of precomputed terms, which is defined recursively:

- if t is a numeral or a symbolic constant then $[t]$ is $\{t\}$;
- if t is $f(t_1,\dots,t_n)$ then $[t]$ is the set of terms $f(r_1,\dots,r_n)$ for all $r_1 \in [t_1],\dots,\ r_n \in [t_n]$;
- if t is $t_1 + t_2$ then $[t]$ is the set of numerals $\overline{n_1+n_2}$ for all integers n_1, n_2 such that $\overline{n_1} \in [t_1]$ and $\overline{n_2} \in [t_2]$; similarly when t is $t_1 - t_2$ or $t_1 \times t_2$;
- if t is t_1/t_2 then $[t]$ is the set of numerals $\overline{\lfloor n_1/n_2 \rfloor}$ for all integers n_1, n_2 such that $\overline{n_1} \in [t_1]$, $\overline{n_2} \in [t_2]$, and $n_2 \neq 0$;
- if t is $t_1 \mathinner{..} t_2$ then $[t]$ is the set of numerals $\overline{m}$ for all integers m such that, for some integers n_1, n_2,

$$\overline{n_1} \in [t_1], \qquad \overline{n_2} \in [t_2], \qquad n_1 \le m \le n_2.$$

For example, $[\overline{1} \mathinner{..} (\overline{7} - \overline{5})] = \{\overline{1}, \overline{2}\}$; if t contains a symbolic constant in the scope of an arithmetic operation then $[t] = \emptyset$.

3.2.2 Propositional Images of Ground Literals and Comparisons

If A is a ground atom $p(t_1,\dots,t_n)$ then

- $\tau_\wedge A$ stands for the conjunction of the atoms $p(r_1,\dots,r_n)$ for all $r_1 \in [t_1],\dots,r_n \in [t_n]$, and $\tau_\vee A$ is the disjunction of these atoms;
- $\tau_\wedge(\mathit{not}\ A)$ is $\neg\tau_\vee A$, and $\tau_\vee(\mathit{not}\ A)$ is $\neg\tau_\wedge A$;
- $\tau_\wedge(\mathit{not}\ \mathit{not}\ A)$ is $\neg\neg\tau_\wedge A$, and $\tau_\vee(\mathit{not}\ \mathit{not}\ A)$ is $\neg\neg\tau_\vee A$.

For any ground terms t_1, t_2,

- $\tau_\wedge(t_1 \prec t_2)$ is $\top$ if the relation $\prec$ holds between the terms r_1 and r_2 for all $r_1 \in [t_1]$ and $r_2 \in [t_2]$, and $\bot$ otherwise;
- $\tau_\vee(t_1 \prec t_2)$ is $\top$ if the relation $\prec$ holds between the terms r_1 and r_2 for some r_1, r_2 such that $r_1 \in [t_1]$ and $r_2 \in [t_2]$, and $\bot$ otherwise.

For example, $\tau_\vee(\overline{3} = \overline{1}..\overline{3})$ is $\top$.

3.2.3 Propositional Images of Rules and Programs

For any ground rule R of form (10), τR stands for the propositional rule

$$\tau_\wedge H_1 \vee \cdots \vee \tau_\wedge H_k \leftarrow \tau_\vee B_1 \wedge \cdots \wedge \tau_\vee B_m.$$

For any ground rule R of form (11), where A is $p(t_1,\dots,t_n)$, τR stands for the propositional rule

$$\bigwedge_{r_1 \in [t_1],\dots,r_n \in [t_n]} (p(r_1,\dots,r_n) \vee \neg p(r_1,\dots,r_n)) \leftarrow \tau_\vee B_1 \wedge \cdots \wedge \tau_\vee B_m.$$

A *ground instance* of a rule R is a ground rule obtained from R by substituting precomputed terms for variables. The propositional image τR of a rule R with variables is the set of the propositional images of the instances of R. For any program Π, $\tau\Pi$ is the union of the sets τR for all rules R of Π.

3.2.4 Examples

▶ **Example 6.** The propositional image of the ground rule

$$a(\overline{1}\,..\,\overline{3}) \leftarrow b(\overline{4}\,..\,\overline{6})$$

is the propositional rule

$$a(\overline{1}) \wedge a(\overline{2}) \wedge a(\overline{3}) \leftarrow b(\overline{4}) \vee b(\overline{5}) \vee b(\overline{6}).$$

▶ **Example 7.** The propositional image of the rule

$$\{a(X)\} \leftarrow X = \overline{1}\,..\,\overline{3}$$

consists of the propositional rules

$$\begin{array}{ll} a(\overline{n}) \vee \neg a(\overline{n}) \leftarrow \top & \text{for } n \in \{1,2,3\}, \\ a(r) \vee \neg a(r) \leftarrow \bot & \text{for all precomputed terms } r \text{ other than } \overline{1}, \overline{2}, \overline{3}. \end{array}$$

▶ **Example 8.** The propositional image of the program

$$\begin{array}{r} a(\overline{1}) \vee a(\overline{2}) \leftarrow \textit{not } a(\overline{0}), \\ b(X) \leftarrow a(X) \wedge a(X+\overline{1}) \end{array}$$

is

$$\begin{array}{rl} a(\overline{1}) \vee a(\overline{2}) \leftarrow \neg a(\overline{0}), & \\ b(\overline{n}) \leftarrow a(\overline{n}) \wedge a(\overline{n+1}) & \text{for all } n \in \mathbf{Z}, \\ b(r) \leftarrow a(r) \wedge \bot & \text{for all precomputed terms } r \text{ other than numerals.} \end{array}$$

The first two lines are similar to the propositional program from Example 2 (Section 2.2); the rules in the last line are trivial, as defined in Section 2.2.

▶ **Example 9.** The propositional image of the program

$$\begin{array}{r} a(\overline{1}) \leftarrow, \\ a(\overline{2} \times X) \leftarrow a(X) \end{array}$$

is

$$\begin{array}{rl} a(\overline{1}) \leftarrow \top, & \\ a(\overline{2n}) \leftarrow a(\overline{n}) & \text{for all } n \in \mathbf{Z}, \\ \top \leftarrow a(r) & \text{for all precomputed terms } r \text{ other than numerals.} \end{array}$$

The first two lines are similar to the propositional program from Example 3.

▶ **Example 10.** The propositional image of the program

$$\begin{array}{r} a(\overline{0}) \leftarrow, \\ b(X,Y) \leftarrow a(Y-X) \end{array}$$

is

$$\begin{array}{rl} a(\overline{0}) \leftarrow \top, & \\ b(\overline{m},\overline{n}) \leftarrow a(\overline{n-m}) & \text{for all } m,n \in \mathbf{Z}, \\ b(r,s) \leftarrow \bot & \text{for all precomputed terms } r,\ s \text{ such that at least one of them} \\ & \text{is not a numeral.} \end{array}$$

The first two lines are similar to the propositional program from Example 4.

4 Supersafe Rules

The idea of the definition below can be explained in terms of a "guessing game." Imagine that you and I are looking at a safe rule R, and I form a ground instance of R by substituting precomputed terms for its variables in such a way that all comparisons in the body of the rule become true. I do not show you that instance, but for every term that occurs as an argument of a nonnegated atom in its body I tell you what the value of that term is. If R is supersafe then on the basis of this information you will be able to find out which terms I substituted for the variables of R; or, at the very least, you will be able to restrict the possible choices to a finite set. If, on the other hand, R is not supersafe then the information that I give you will be compatible with infinitely many substitutions.

Consider, for example, the rule

$$p(X, Y, Z) \leftarrow X = \overline{5}\,..\,\overline{7} \wedge q(\overline{2} \times Y) \wedge \mathit{not}\ q(\overline{3} \times Y) \wedge Y = Z + \overline{1}. \tag{12}$$

Imagine that I chose an instance of this rule such that both comparisons in its body are true, and told you that the value of $\overline{2} \times Y$ in that instance is, for example, 10. You will be able to conclude that the value chosen for Y is 5, and that consequently the value of Z is 4. About the value that I chose for X you will be able to say that it is one of the numbers 5, 6, and 7. We see that rule (12) has only three ground instances compatible with the information about the value of $\overline{2} \times Y$ that I gave you; the rule is supersafe.

On the other hand, if we replace $\overline{2} \times Y$ in rule (12) by $\overline{0} \times Y$ then the situation will be different: I will tell you that the value of $\overline{0} \times Y$ is $\overline{0}$, and this information will not allow you to restrict the possible substitutions to a finite set. The modified rule is not supersafe.

4.1 Definition of Supersafety

A term or an atom is *interval-free* if it does not contain the interval symbol (..). We will define when a safe rule R is supersafe assuming that R satisfies the following additional condition:

All nonnegated atoms in the body of R are interval-free. (IF)

This simplifying assumption eliminates rules like the one in Example 6. It is useful because, for a term t containing intervals, the set $[t]$ may have more than one element; in the description of the guessing game we glossed over this complication when we talked above about *the* value of a term as if it were a uniquely defined object. On the other hand, if a rule R satisfies condition (IF) then for every term t occurring in a nonnegated atom in the body of a ground instance of R the set $[t]$ has at most one element. (An atom violating condition (IF) can be eliminated using a new variable; for instance, we can replace $b(\overline{4}..\overline{6})$ in Example 4 by $b(X) \wedge X = \overline{4}..\overline{6}$.)

The *positive body arguments* of R are the members of the tuples $\mathbf{t}$ for all nonnegated atoms $p(\mathbf{t})$ in the body of R. For example, the only positive body argument of (12) is $\overline{2} \times Y$. The values of positive body arguments constitute the information about an instance of the rule that is available to you in the guessing game.

The instances of a rule that are allowed in the guessing game can be characterized by "acceptable tuples of terms," defined as follows. Let $\mathbf{x}$ be the list of all variables occurring in R, and let $\mathbf{r}$ be a tuple of precomputed terms of the same length as $\mathbf{x}$. We say that $\mathbf{r}$ is *acceptable (for R)* if

$$\begin{array}{ccccc}
\cdots & \overline{5},\overline{-1},\overline{-2} & \overline{5},\overline{0},\overline{-1} & \overline{5},\overline{1},\overline{0} & \cdots \\
\cdots & \overline{6},\overline{-1},\overline{-2} & \overline{6},\overline{0},\overline{-1} & \overline{6},\overline{1},\overline{0} & \cdots \\
\cdots & \overline{7},\overline{-1},\overline{-2} & \overline{7},\overline{0},\overline{-1} & \overline{7},\overline{1},\overline{0} & \cdots
\end{array}$$

Figure 1 Acceptable tuples for rules (12) and (13).

(i) for each comparison C in the body of R, $\tau_\vee(C^{\mathbf{x}}_{\mathbf{r}}) = \top$;[5]

(ii) for each positive body argument t of R, the set $[t^{\mathbf{x}}_{\mathbf{r}}]$ is non-empty (and consequently is a singleton).

For instance, a tuple r_1, r_2, r_3 is acceptable for rule (12) if

- r_1 is one of the numerals $\overline{5}$, $\overline{6}$, $\overline{7}$, so that $\tau_\vee(r_1 = \overline{5}\,..\,\overline{7}) = \top$;
- r_2 is a numeral $\overline{l}$ (rather than symbolic constant), so that the set $[\overline{2} \times r_2]$ is non-empty;
- r_3 is the numeral $\overline{l-1}$, so that $\tau_\vee(r_2 = r_3 + \overline{1}) = \top$.

(See Figure 1.)

The information about the values of positive arguments that I give you in the guessing game can be described in terms of equivalence classes of acceptable tuples. About acceptable tuples $\mathbf{r}$, $\mathbf{s}$ we say that they are *equivalent* if for each positive body argument t of rule R, $[t^{\mathbf{x}}_{\mathbf{r}}] = [t^{\mathbf{x}}_{\mathbf{s}}]$. In the case of rule (12), for example, acceptable tuples r_1, r_2, r_3 and s_1, s_2, s_3 are equivalent iff r_2 equals s_2 (so that $[\overline{2} \times r_2] = [\overline{2} \times s_2]$). In Figure 1, each column is an equivalence class of this relation.

We say that R is *supersafe* if all equivalence classes of acceptable tuples for it are finite.

For example, rule (12) is supersafe because each equivalence class of acceptable tuples for it has 3 elements. Consider now the rule obtained from (12) by replacing $\overline{2} \times Y$ with $\overline{0} \times Y$:

$$p(X, Y, Z) \leftarrow X = \overline{5}\,..\,\overline{7} \,\wedge\, q(\overline{0} \times Y) \,\wedge\, \neg\, q(\overline{3} \times Y) \;\wedge\; Y = Z + \overline{1}. \tag{13}$$

The set of acceptable tuples does not change, but now all of them are equivalent: for any $\mathbf{r}$ and $\mathbf{s}$,

$$[\overline{0} \times r_2] = [\overline{0} \times s_2] = \{\overline{0}\}.$$

The only equivalence class is the set of all acceptable tuples, so that the rule is not supersafe.

It is easy to check that rules (1), (5), (6) and all rules in Examples 7–9 are supersafe,[6] and that rules (2)–(4) and the second rule in Example 10 are not.

The concept of supersafety can be applied also to individual variables occurring in a rule. As before, let R be a safe rule satisfying condition (IF). Let $\mathbf{x}$ be the list of all variables $X_1, \dots, X_n$ occurring in R, and let $\mathbf{r}$ be an acceptable tuple of precomputed terms $r_1, \dots, r_n$. We say that a variable X_i is *supersafe* in R if, for every equivalence class E of acceptable tuples, the i-th projection of E (that is, the set of the terms r_i over all tuples $r_1, \dots, r_n$ from E) is finite. It is easy to see that R is supersafe iff all variables occurring in R are supersafe. Indeed, a subset E of the Cartesian product of finitely many sets is finite iff all projections of E are finite.

As an example, consider the only equivalence class of acceptable tuples for rule (13), shown in Figure 1. The first projection of that set is $\{\overline{5}, \overline{6}, \overline{7}\}$; the second projection is the set of all numerals, and the third projection is the set of all numerals as well. Consequently, the variable X is supersafe, and the variables Y and Z are not.

[5] By $C^{\mathbf{x}}_{\mathbf{r}}$ we denote the result of substituting the terms $\mathbf{r}$ for the variables $\mathbf{x}$ in C.

[6] In the syntax of Section 3.1, rules (5) and (6) would be written as $p(X, Y) \leftarrow X = Y \wedge q(X)$ and $p(X) \leftarrow X + \overline{3} = \overline{4}$.

4.2 Supersafety in the Absence of Arithmetic Operations

As could be expected, the difference between safety and supersafety disappears in the absence of arithmetic operations. In the following theorem, R is a safe rule satisfying condition (IF).

▶ **Theorem 11.** *If a positive body argument t of R does not contain the arithmetic operations*

$$+ \qquad - \qquad \times \qquad /$$

then all variables occurring in t are supersafe.

Proof. We will show that if X_i occurs in a positive body argument t of R that does not contain arithmetic operations then the i-th projection of any equivalence class of acceptable tuples is a singleton. We will prove, in other words, that for any pair of equivalent acceptable tuples $\mathbf{r}$ and $\mathbf{s}$, $r_i = s_i$. Assume that $\mathbf{r}$ is equivalent to $\mathbf{s}$. Then $[t^{\mathbf{x}}_{\mathbf{r}}] = [t^{\mathbf{x}}_{\mathbf{s}}]$. Since t is interval-free and does not contain arithmetic operations, and $\mathbf{r}$, $\mathbf{s}$ are precomputed, both $t^{\mathbf{x}}_{\mathbf{r}}$ and $t^{\mathbf{x}}_{\mathbf{s}}$ are precomputed also, so that $[t^{\mathbf{x}}_{\mathbf{r}}]$ is the singleton $\{t^{\mathbf{x}}_{\mathbf{r}}\}$, and $[t^{\mathbf{x}}_{\mathbf{s}}]$ is the singleton $\{t^{\mathbf{x}}_{\mathbf{s}}\}$. It follows that the term $t^{\mathbf{x}}_{\mathbf{r}}$ is the same as $t^{\mathbf{x}}_{\mathbf{s}}$. Since X_i is a member of the tuple $\mathbf{x}$ and occurs in t, we can conclude that $r_i = s_i$. ◀

▶ **Corollary 12.** *If the body of R contains neither arithmetic operations nor comparisons then R is supersafe.*

Indeed, since R is safe and its body does not contain comparisons, every variable occurring in R occurs also in one of its positive body arguments.

4.3 Supersafety is Undecidable

The conditions imposed on variables by GRINGO (see Footnote 2) ensure their supersafety, but they can be relaxed without losing this property. There is no need, for example, to reject the rule

$$p(X) \leftarrow q(X/\overline{2})$$

– it is supersafe. The use of unnecessarily strong restrictions on variables in the design of GRINGO can be explained by the desire to make the grounding algorithm less complicated.

There is, however, a more fundamental reason why the class of rules accepted by GRINGO for grounding does not exactly match the class of supersafe rules:

▶ **Theorem 13.** *Membership in the class of supersafe rules is undecidable.*

Proof. The undecidable problem of determining whether a Diophantine equation has a solution [8] can be reduced to deciding whether a rule is supersafe as follows. The safe rule

$$p(Y) \leftarrow f(\mathbf{x}) = \overline{0} \times Y,$$

where $f(\mathbf{x})$ is a polynomial with integer coefficients and Y is a variable different from the members of $\mathbf{x}$, is supersafe iff the equation $f(\mathbf{x}) = 0$ has no solutions. Indeed, if the equation has no solutions then the set of acceptable tuples is empty and the rule is trivially supersafe. If it has a solution $\mathbf{r}$ then the set of acceptable tuples is infinite, because an acceptable tuple can be formed from $\mathbf{r}$ by appending any numeral $\overline{n}$. All acceptable tuples form one equivalence class, because the rule in question has no positive body arguments. ◀

5 Intelligent Instantiation

5.1 Intelligent Instantiation as Selecting Essential Instances

Consider a program Π such that its rules satisfy condition (IF). As observed in Section 4.1, for every term t occurring in a nonnegated atom in the body of a ground instance of a rule of Π, the set $[t]$ has at most one element. It follows that for every nonnegated atom A in the body of a ground instance of a rule of Π, the formula $\tau_\vee A$ is either an atom or the symbol $\bot$. Consequently, the body of every rule of the propositional image $\tau\Pi$ of Π is a conjunction of formulas of three types: symbols $\top$ and $\bot$, atoms, and formulas beginning with negation. In other words, the definition of an essential rule in Section 2.2 is applicable to the propositional program $\tau\Pi$, and we can talk about its essential rules.

For instance, if Π is the program from Example 8 then the propositional program $\tau\Pi$ has two essential rules:

$$a(\overline{1}) \vee a(\overline{2}) \leftarrow \neg a(\overline{0}) \tag{14}$$

of degree 1, and

$$b(\overline{1}) \leftarrow a(\overline{1}) \wedge a(\overline{2}) \tag{15}$$

of degree 2.

If, for every k, $\tau\Pi$ has only finitely many essential rules of degree k, as in this example, then the stepwise process of generating the essential rules of $\tau\Pi$ of higher and higher degrees can be thought of as a primitive, but useful, theoretical model of the process of intelligent instantiation. It is primitive in the sense that this process involves not only identifying essential instances but also simplifying them. In application to the program from Example 8, GRINGO will not only find the essential instances (14) and (15); it will also simplify rule (14) by dropping its body.

The supersafety of all rules of a program guarantees the possibility of launching the process of intelligent instantiation, although without guarantee of termination:

▶ **Theorem 14.** *If Π is a finite program, and all rules of Π are supersafe, then each of the sets $E_k(\tau\Pi)$ is finite.*

The program from Example 10 shows that the assertion of the theorem would be incorrect without the supersafety assumption. The propositional image of that program has infinitely many essential rules of degree 2 – rules $b(\overline{n}, \overline{n}) \leftarrow a(\overline{0})$ for all integers n.

5.2 Proof of Theorem 14

5.2.1 Plan of the Proof

The assertion of the theorem will be derived from the two lemmas stated below.

Consider a finite program Π such that all rules of Π are supersafe. For any rule R of Π and any set S of atoms from $\tau\Pi$, by $\rho(R, S)$ we denote the set of all tuples $\mathbf{r}$ of precomputed terms that are acceptable for R such that all nonnegated atoms of the body of $\tau(R^{\mathbf{x}}_{\mathbf{r}})$ (where $\mathbf{x}$ is the list of variables of R) belong to S.

▶ **Lemma 15.** *If R is safe and S is finite then $\rho(R, S)$ is finite.*

By S_k we denote the set of the nonnegated atoms of the heads of the rules of $E_k(\tau\Pi)$.

▶ **Lemma 16.** *Every rule of $E_{k+1}(\tau\Pi)$ has the form $\tau(R^{\mathbf{x}}_{\mathbf{r}})$, where R is a rule of Π, $\mathbf{x}$ is the list of its variables, and $\mathbf{r}$ belongs to $\rho(R, S_k)$.*

Given these lemmas, Theorem 14 can be proved by induction on k as follows. If $E_k(\tau\Pi)$ is finite then S_k is finite as well. By Lemma 15, we can further conclude that for every rule R of Π, $\rho(R, S_k)$ is finite. Hence, by Lemma 16, $E_{k+1}(\tau\Pi)$ is finite as well.

5.2.2 Proof of Lemma 15

Let B be the set of positive body arguments of R, and let T be the set of the members of the tuples $\mathbf{t}$ for all atoms $p(\mathbf{t})$ in S. For every function ϕ from B to T, by $\rho_\phi(R, S)$ we denote the subset of $\rho(R, S)$ consisting of the tuples $\mathbf{r}$ such that $\phi(t) \in [t^{\mathbf{x}}_{\mathbf{r}}]$. We will prove the following two assertions:

Claim 1: The subsets $\rho_\phi(R, S)$ cover the whole set $\rho(R, S)$.

Claim 2: Each subset $\rho_\phi(R, S)$ is finite.

It will follow then that $\rho(R, S)$ is finite, because there are only finitely many functions from B to T.

To prove Claim 1, consider an arbitrary tuple $\mathbf{r}$ from $\rho(R, S)$. We want to find a function ϕ from B to T such that $\mathbf{r}$ belongs to $\rho_\phi(R, S)$. For every term t from B, the set $[t^{\mathbf{x}}_{\mathbf{r}}]$, where $\mathbf{x}$ is the list of variables of R, is non-empty, in view of the fact that $\mathbf{r}$, like all tuples in $\rho(R, S)$, is acceptable for R. Since t is interval-free, we can further conclude that $[t^{\mathbf{x}}_{\mathbf{r}}]$ is a singleton. Choose the only element of this set as $\phi(t)$. Let us check that $\phi(t)$ belongs to T; it will be clear then that $\mathbf{r}$ belongs to $\rho_\phi(R, S)$. Since t is a positive body argument of R, it is a member of the tuple $\mathbf{u}$ for some atom $p(\mathbf{u})$ of the body of R. Then $\tau(p(\mathbf{u}^{\mathbf{x}}_{\mathbf{r}}))$ is a nonnegated atom in the body of $\tau(R^{\mathbf{x}}_{\mathbf{r}})$. It has the form $p(\mathbf{t})$, where $\mathbf{t}$ is a tuple of terms containing $\phi(t)$. Since $\mathbf{r}$ belongs to $\rho(R, S)$, the atom $p(\mathbf{t})$ belongs to S, so that $\phi(t)$ belongs to T.

To prove Claim 2, note that all tuples from $\rho_\phi(R, S)$ are equivalent to each other. Indeed, if $\mathbf{r_1}$ and $\mathbf{r_2}$ belong to $\rho_\phi(R, S)$ then, for every t from B, $\phi(t)$ belongs both to $[t^{\mathbf{x}}_{\mathbf{r_1}}]$ and to $[t^{\mathbf{x}}_{\mathbf{r_2}}]$; since both sets are singletons, it follows that they are equal to each other. We showed, in other words, that $\rho_\phi(R, S)$ is a subset of a class of equivalent tuples. Since R is supersafe, this equivalence class is finite.

5.2.3 Proof of Lemma 16

Every rule of $\tau\Pi$ is obtained by applying τ to an instance $R^{\mathbf{x}}_{\mathbf{r}}$ of some rule R of Π. Assuming that a rule $\tau(R^{\mathbf{x}}_{\mathbf{r}})$ belongs to $E_{k+1}(\tau\Pi)$, we need to show that $\mathbf{r}$ belongs to $\rho(R, S_k)$. In other words, we need to check, first, that r is acceptable for R, and second, that all nonnegated atoms of the body of $\tau(R^{\mathbf{x}}_{\mathbf{r}})$ belong to S_k. The first property follows from the fact that all rules of $E_{k+1}(\tau\Pi)$ are nontrivial, because if $\mathbf{r}$ is not acceptable for R then the body of $\tau(R^{\mathbf{x}}_{\mathbf{r}})$ includes the conjunctive term $\bot$. According to the definition of S_k, the second property can be expressed as follows: every nonnegated atom of the body of $\tau(R^{\mathbf{x}}_{\mathbf{r}})$ is a nonnegated atom of the head of some rule of $E_k(\tau\Pi)$. This is immediate from the assumption that rule $\tau(R^{\mathbf{x}}_{\mathbf{r}})$ belongs to $E_{k+1}(\tau\Pi)$.

6 Conclusion

Supersafety is a property of rules with variables and arithmetic operations. If all rules of a program are supersafe then the process of accumulating the ground instances of its rules that are essential for finding its stable models will produce only a finite set of rules at every step.

This paper extends earlier work on the mathematics of the input language of GRINGO [5]. Unlike other publications on the theory of safe rules and intelligent instantiation in answer set programming [2, 3, 4, 6, 9], it concentrates on the difficulties related to the use of arithmetic operations. It is limited, however, to programs without GRINGO constructs that involve local variables – conditional literals and aggregates. Extending the theory of supersafety to local variables is a topic for future work.

Acknowledgements. Thanks to Amelia Harrison, Roland Kaminski, Dhananjay Raju, and the anonymous referees for useful comments.

References

1 Evgenii Balai, Michael Gelfond, and Yuanlin Zhang. Towards answer set programming with sorts. In *Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, pages 135–147, 2013.

2 Annamaria Bria, Wolfgang Faber, and Nicola Leone. Normal form nested programs. In *Proceedings of European Conference on Logics in Artificial Intelligence (JELIA)*, 2008.

3 Pedro Cabalar, David Pearce, and Agustin Valverde. A revised concept of safety for general answer set programs. In *Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, 2009.

4 Francesco Calimeri, Susanna Cozza, Giovambattista Ianni, and Nicola Leone. Computable functions in ASP: theory and implementation. In *Proceedings of International Conference on Logic Programming (ICLP)*, pages 407–424, 2008.

5 Martin Gebser, Amelia Harrison, Roland Kaminski, Vladimir Lifschitz, and Torsten Schaub. Abstract Gringo. *Theory and Practice of Logic Programming*, 15:449–463, 2015.

6 Joohyung Lee, Vladimir Lifschitz, and Ravi Palla. Safe formulas in the general theory of stable models (preliminary report). In *Proceedings of International Conference on Logic Programming (ICLP)*, pages 672–676, 2008.

7 Vladimir Lifschitz, Lappoon R. Tang, and Hudson Turner. Nested expressions in logic programs. *Annals of Mathematics and Artificial Intelligence*, 25:369–389, 1999.

8 Yuri Matiyasevich. *Hilbert's Tenth Problem.* MIT Press, 1993.

9 Norman McCain and Hudson Turner. Language independence and language tolerance in logic programs. In Pascal Van Hentenryck, editor, *Proceedings Eleventh International Conference on Logic Programming*, pages 38–57, 1994.

An Answer Set Programming Framework for Reasoning About Truthfulness of Statements by Agents

Tran Cao Son[1], Enrico Pontelli[2], Michael Gelfond[3], and Marcello Balduccini[4]

1 Dept. Computer Science, New Mexico State University, Las Cruces, NM, USA
tson@cs.nmsu.edu
2 Dept. Computer Science, New Mexico State University, Las Cruces, NM, USA
epontell@cs.nmsu.edu
3 Dept. Computer Science, Texas Tech University, Lubbock, TX, USA
michael.gelfond@ttu.edu
4 Dept. Computer Science, Drexel University, Philadelphia, PA, USA
marcello.balduccini@drexel.edu

Abstract

We propose a framework for answering the question of whether statements made by an agent can be believed, in light of observations made over time. The basic components of the framework are a formalism for reasoning about actions, changes, and observations and a formalism for default reasoning. The framework is suitable for concrete implementation, e.g., using answer set programming for asserting the truthfulness of statements made by agents, starting from observations, knowledge about the actions of the agents, and a theory about the "normal" behavior of agents.

1998 ACM Subject Classification F.4.1 Mathematical Logic, I.2.3 Deduction and Theorem Proving, I.2.4 Knowledge Representation Formalisms and Methods, I.2.11 Distributed Artificial Intelligence

Keywords and phrases Agents, ASP, Reasoning, Knowledge

Digital Object Identifier 10.4230/OASIcs.ICLP.2016.8

1 Introduction

In this extended abstract, we are interested in reasoning about the truthfulness of statements made by agents. We assume that we can observe the world as well as agents' actions. The basis for our judgments will be composed of our observations, performed along a linear time line, along with our commonsense knowledge about agents' behavior and the world. We assume that observations are *true* at the time they are made, and will stay true until additional pieces of information prove otherwise. Our judgments reflect what we believe. They might not correspond to the ground truth and could change over time. This is because we often have to make our judgment in presence of incomplete information. This makes reasoning about the truthfulness of statements made by agents *non-monotonic*. Furthermore, our judgment against a statement is *independent* of whether or not we trust the agent from whom the statement originated. This is illustrated in the next example.

▶ **Example 1.**

- *Time* t_0: When we first meet, John said that his family is poor (property *poor* is true). It is likely that we would believe John—since we have no reasons to conclude otherwise.
- *Time* t_1: We observe the fact that John attends an expensive college (property *in_college* is true). Since students attending the college are *normally* from rich families (default d_1), this would lead us to conclude that John has lied to us. We indicate that the default d_1 is the reason to draw such conclusion, i.e., we changed our belief on the property *poor*.
- *Time* t_2: We observe the fact that John has a need-based scholarship (property *has_scholarship* is true). Since a student's hardship is *usually* derived from the family's financial situation (default d_2), this fact allows us to withdraw the conclusion that John is a liar, made at time instance t_1. It is still insufficient for us to conclude that John's family is poor.
  The situation might be different if, for example, we have a preference among defaults. In this example, if we are inclined to believe in the conclusion of d_2 more than that of d_1, then we would believe that John's family is poor and thus restore our trust in John's original statement (i.e., truth of *poor*).

In this extended abstract, we

1. present the formalization of an abstract model to represent and reason about truthfulness of agent's statements (briefly summarized in the next section); and
2. discuss the steps for a concrete realization of the model using Answer Set Programming.

2 A General Model for Reasoning about Truthfulness of Statements made by Agents

In this section, we propose a general framework for representing, and reasoning about, the truthfulness of (statements made by) agents[1]. The framework can be instantiated using specific paradigms for reasoning about actions and change and for non-monotonic reasoning. We assume that

- It is possible to observe the properties of the world and the occurrences of the agents' actions over time (e.g., we observe that John buys a car, John is a student, etc.). Let us denote with O_a and O_w the set of action occurrences and the set of observations about the world over time, respectively.
- We have adequate knowledge about the agents' actions and their effects (e.g., the action of buying a car requires that the agent has money and its execution will result in the agent owning a car). This knowledge is represented by an action theory *Act* in a suitable logic A, that allows reasoning about actions' effects and consequent changes to the world. Let $\models_A$ denote the entailment relation defined within the logical framework A used to describe *Act*.
- We have commonsense knowledge about "normal" behavior (e.g., a person attending an expensive school normally comes from a rich family, a person obtaining need-based scholarship usually comes from a poor family). This knowledge is represented by a default theory with preferences *Def*, that enables reasoning about the state of the world and deriving conclusions whenever necessary. Let $\models_D$ denote the entailment relation defined over the default theory framework defining *Def*.

[1] From now on, we will often use "the truthfulness of agents" interchangeably with "the truthfulness of statements made by agents."

The set of observations O_w in Example 1 includes the observations such as 'John comes from a poor family' at time point t_0, 'John attends an expensive college' at time point t_1, and 'John receives a need-based scholarship' at time point t_2. In this particular example we do not have any action occurrences, i.e., $O_a = \emptyset$. Our default theory D consists of d_1 and d_2, which allow us to make conclusions regarding whether John comes from a rich family or not.

Let us consider a theory $T = (O_a, O_w, \mathit{Act}, \mathit{Def})$ and the associated entailment relations $\models_A$ and $\models_D$. We are interested in answering the question of whether a statement asserting that a proposition p is true at a certain time step t, denoted by $p[t]$, is true or not. Specifically, we would like to define the entailment relation $\models$ between T and $p[t]$. Intuitively, this can be done in two steps:

- Compute possible models $W[t]$ of the world at the time point t from Act, O_a, and O_w ($\models_A$); and
- Determine whether p is true given Def and $W[t]$ (using $\models_D$).

Let us assume that the equation $W[t] = \{z \mid \mathit{Act} \cup O_a \cup O_w \models_A z[t]\}$ characterizes any of the states of the world at time step t given Act, O_a, and O_w (based on the semantics of $\models_A$). The entailment relation between T and $p[t]$ can be defined as follows.

$$T \models p[t] \quad \Leftrightarrow \quad \forall W[t].\left(\ W[t] = \{z \mid \mathit{Act} \cup O_a \cup O_w \models_A z[t]\} \Rightarrow \mathit{Def} \cup W[t] \models_D p\ \right) \qquad (1)$$

Note that this definition also allows one to identify elements of O_a and O_w which, when obtained, will result in the confirmation or denial of $T \models p[t]$. As such, a system that obeys (1) can also be used by users who are interested in what they need to do in order to believe in a statement about p at the time step t, given their beliefs about the behavior of the observed agents.

3 Reasoning about Truthfulness of Agents Using ASP

To develop a concrete system for reasoning about truthfulness of agents using (1), specific formalizations of Act and Def need to be developed. There is a large body of research related to these two areas, and deciding which one to use depends on the system developer. Well-known formalisms for reasoning about actions and change, such as action languages [4], situation calculus [8], etc., can be employed for Act (and $\models_A$). Approaches to default reasoning with preferences, such as those proposed in [1, 2, 3, 5]), can be used for Def (and $\models_D$). In addition, let us note that, in the literature, $\models_D$ can represent *skeptical* or *credulous* reasoning; and the model does not specify how observations are collected. Deciding which type of reasoning is suitable or how to collect observations is an important issue, but it is application-dependent and beyond the scope of this extended abstract. Using the formalisms in [5] and [4] for default reasoning and reasoning about actions and change, respectively, we can implement a system for reasoning about truthfulness of agents using answer set programming (ASP) [6, 7] with the following steps:

- Extending the framework in [5] to allow observations at different time points and developing ASP rules for reasoning with observations; for instance, the language needs to allow facts of the form $\mathit{obs}(p, t)$ —fluent literal p is true at time t—and ASP rules for reasoning about defaults and rules given observations at different time point need to be developed.
- Defining a query language for reasoning about statements of agents at different time points; more specifically, given an ASP program Π encoding the theory described in the previous item and a statement $\mathit{stm}(p, t)$—stating that literal p holds at time t—how does Π helps identify whether or not the statement is true or false; for instance, one can say that if Π entails $p[t]$ with respect to the answer set semantics then the statement is true.

- Allowing observations of the form $occ(a, t)$—action a occurs at time t—and developing ASP rules for reasoning about preconditions of actions as well as effects of actions need to be included. More specifically, we can add ASP rules stating that if an action a occurs at time point t then its preconditions must hold at time t, i.e., its preconditions must be observed at time t; furthermore, its effects must hold (or be observed) at time $t + 1$.

4 Conclusions

We proposed a general framework for reasoning about the truthfulness of statements made by an agent. We discussed how the framework can be implemented using ASP using well-known methodologies for reasoning about actions and change and for default reasoning with preferences. The framework does not assume complete knowledge about the agent being observed and the reasoning process builds on observations about the state of the world and occurrences of actions. We had developed an ASP implementation of the framework and explored the use of the framework in simple scenarios derived from man-in-the-middle attacks. The details can be found in the full version of this extended abstract.

References

1 G. Brewka and T. Eiter. Preferred answer sets for extended logic programs. *Artificial Intelligence*, 109:297–356, 1999.

2 G. Brewka and T. Eiter. Prioritizing default logic. In *Intellectics and Computational Logic*, volume 19 of *Applied Logic Series*, pages 27–45. Kluwer, 2000.

3 J. Delgrande, T. Schaub, and H. Tompits. A framework for compiling preferences in logic programs. *Theory and Practice of Logic Programming*, 3(2):129–187, March 2003.

4 M. Gelfond and V. Lifschitz. Action Languages. *Electronic Transactions on Artificial Intelligence*, 3(6), 1998.

5 M. Gelfond and T. C. Son. Reasoning about prioritized defaults. In *Selected Papers from the Workshop on Logic Programming and Knowledge Representation 1997*, pages 164–223. Springer Verlag, LNAI 1471, 1998.

6 V. Marek and M. Truszczyński. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: a 25-year Perspective*, pages 375–398, 1999.

7 I. Niemelä. Logic programming with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25(3,4):241–273, 1999.

8 R. Reiter. *KNOWLEDGE IN ACTION: Logical Foundations for Specifying and Implementing Dynamical Systems.* The MIT Press, 2001.

Answer Set Solving with Generalized Learned Constraints*

Martin Gebser[1], Roland Kaminski[2], Benjamin Kaufmann[3], Patrick Lühne[4], Javier Romero[5], and Torsten Schaub[6]

1 University of Potsdam, Potsdam, Germany
2 University of Potsdam, Potsdam, Germany
3 University of Potsdam, Potsdam, Germany
4 University of Potsdam, Potsdam, Germany
5 University of Potsdam, Potsdam, Germany
6 University of Potsdam, Potsdam, Germany; and
INRIA, Rennes, France

Abstract

Conflict learning plays a key role in modern Boolean constraint solving. Advanced in satisfiability testing, it has meanwhile become a base technology in many neighboring fields, among them *answer set programming* (ASP). However, learned constraints are only valid for a currently solved problem instance and do not carry over to similar instances. We address this issue in ASP and introduce a framework featuring an integrated feedback loop that allows for reusing conflict constraints. The idea is to extract (propositional) conflict constraints, generalize and validate them, and reuse them as integrity constraints. Although we explore our approach in the context of dynamic applications based on transition systems, it is driven by the ultimate objective of overcoming the issue that learned knowledge is bound to specific problem instances. We implemented this workflow in two systems, namely, a variant of the ASP solver *clasp* that extracts integrity constraints along with a downstream system for generalizing and validating them.

1998 ACM Subject Classification D.1.6 Logic Programming, I.2.3 Deduction and Theorem Proving

Keywords and phrases Answer Set Programming, Conflict Learning, Constraint Generalization, Generalized Constraint Feedback

Digital Object Identifier 10.4230/OASIcs.ICLP.2016.9

1 Introduction

Modern solvers for *answer set programming* (ASP) such as *cmodels* [13], *clasp* [11], and *wasp* [1] owe their high effectiveness to advanced Boolean constraint processing techniques centered on *conflict-driven constraint learning* (CDCL; [2]). Unlike pure backtracking, CDCL analyzes encountered conflicts and acquires new constraints while solving, which are added to the problem specification to prune the remaining search space. This strategy often leads to considerably reduced solving times compared to simple backtracking. However, constraints learned in this way are propositional and only valid for the currently solved logic program. Learned constraints can thus only be reused as is for solving the very same problem; they cannot be transferred to solving similar problems, even if they share many properties. For illustration, consider a maze problem, which consists of finding the shortest way out of a

* This work was partially supported by DFG-SCHA-550/9.

labyrinth. When solving an instance, the solver might learn that shortest solutions never contain a move west followed by a move east, that is, a simple loop. However, the solver only learns this for a specific step but cannot transfer the information to other steps, let alone for solving any other maze instance.

In what follows, we address this shortcoming and introduce a framework for *reusing* learned constraints, with the ultimate objective of overcoming the issue that learned knowledge is bound to specific instances. Reusing learned constraints consists of enriching a program with conflict constraints learned in a previous run. More precisely, our approach proceeds in four steps: (1) extracting constraints while solving, (2) generalizing them, which results in candidates, (3) validating the candidates, and (4) enriching the program with the valid ones. Since this mechanism involves a feedback step, we refer to it as *constraint feedback*. We implemented our framework as two systems, a variant of the ASP solver *clasp* 3 addressing step (1), referred to as *xclasp*, and a downstream system dealing with steps (2) and (3), called *ginkgo*. Notably, we use ASP for implementing different proof methods addressing step (3). The resulting integrity constraints can then be used to enrich the same or "similar" problem instances. To be more precise, we apply our approach in the context of automated planning, as an exemplar of a demanding and widespread application area representative of dynamic applications based on transition systems. As such, our approach readily applies to other related domains, such as action languages or model checking. Furthermore, automated planning is of particular interest because it involves invariants. Although such constraints are specific to a planning problem, they are often independent of the planning instance and thus transferable from one instance of a problem to another. Returning to the above maze example, this means that the constraint avoiding simple loops does not only generalize to all time steps, but is moreover independent of the particular start and exit position.

2 Background

A *logic program* is a set of rules of the form

$$a_0 \leftarrow a_1, \dots, a_m, {\sim}a_{m+1}, \dots, {\sim}a_n \tag{1}$$

where each a_i is a first-order atom for $0 \leq i \leq n$ and "$\sim$" stands for *default negation*. If $n = 0$, rule (1) is called a *fact*. If a_0 is omitted, rule (1) represents an integrity constraint. Further language constructs exist but are irrelevant to what follows (cf. [3]). Rules with variables are viewed as shorthands for the set of their ground instances. Whenever we deal with authentic source code, we switch to typewriter font and use "`:-`" and "`not`" instead of "$\leftarrow$" and "$\sim$"; otherwise, we adhere to the ASP language standard [4]. Semantically, a ground logic program induces a collection of *answer sets*, which are distinguished models of the program determined by answer set semantics; see [12] for details.

Accordingly, the computation of answer sets of logic programs is done in two steps. At first, an ASP grounder instantiates a given logic program. Then, an ASP solver computes the answer sets of the obtained ground logic program. In CDCL-based ASP solvers, the computation of answer sets relies on advanced Boolean constraint processing. To this end, a ground logic program P is transformed into a set Δ_P of *nogoods*, a common (negative) way to represent constraints [8]. A nogood can be understood as a set $\{\ell_1, \dots, \ell_n\}$ of literals representing an invalid partial truth assignment. Logically, this amounts to the formula $\neg(\ell_1 \wedge \dots \wedge \ell_n)$, which in turn can be interpreted as an integrity constraint of the form "$\leftarrow \ell_1, \dots, \ell_n$." By representing a total assignment as a set S of literals, one for each available atom, S is a *solution* for a set Δ of nogoods if $\delta \not\subseteq S$ for all $\delta \in \Delta$. Conversely, S is *conflicting* if $\delta \subseteq S$ for some $\delta \in \Delta$. Such a nogood is called a *conflict nogood* (and the starting point of conflict analysis in CDCL-based solvers). Finally, given a nogood δ and a set S representing a partial assignment, a literal $\ell \notin S$ is *unit-resulting* for δ with respect to S if $\delta \setminus S = \{\overline{\ell}\}$, where $\overline{\ell}$ is the complement of ℓ. Such a nogood δ is called a *reason* for ℓ. That is, if all but one literal of a nogood are contained in an assignment, the complement of the remaining literal must hold in any solution extending the current assignment. *Unit propagation* is the iterated process of extending assignments with unit-resulting literals until no further literal is unit-resulting for any nogood. For instance, consider the partial assignment $\{a \mapsto t, b \mapsto f\}$ represented by $\{a, {\sim}b\}$. Then, ${\sim}c$ is unit-resulting for $\{a, c\}$, leading to the extended assignment $\{a, {\sim}b, {\sim}c\}$. In other words, $\{a, c\}$ is a reason for ${\sim}c$ in $\{a, {\sim}b, {\sim}c\}$. In this way, nogoods provide reasons explaining why literals belong to a solution. Note that any individual assignment is obtained by either a choice operation or unit propagation. Accordingly, assignments are partitioned into *decision levels*. Level zero comprises all initially propagated literals; each higher decision level consists of one choice literal along with successively propagated literals. Further Boolean constraint processing techniques can be used to analyze and recombine inherent reasons for conflicts, as described in Section 3.1. We refer the reader to [11] for a detailed account of the aforementioned concepts.

3 Generalization of Learned Constraints

This section presents our approach by following its four salient steps. At first, we detail how conflict constraints are extracted while solving a logic program and turned into integrity constraints. Then, we describe how the obtained integrity constraints can be generalized by replacing specific terms by variables. Next, we present ASP-based proof methods for validating the generated candidate constraints. For clarity, these methods are developed in the light of our application area of automated planning. Finally, we close the loop and discuss the range of problem instances that can be enriched by the resulting integrity constraints.

While we implemented constraint extraction as an extension to *clasp*, referred to as *xclasp*, our actual constraint feedback framework involving constraint generalization and validation is comprised in the *ginkgo* system. The implementation of both systems is detailed in Section 4.

3.1 Extraction

Modern CDCL solvers gather knowledge in the form of conflict nogoods while solving. Accessing these learned nogoods is essential for our approach. To this end, we have to instrument a solver such as *clasp* to record conflict nogoods resulting from conflict analysis. This necessitates a modification of the solver's conflict resolution scheme, as the learned nogoods can otherwise contain auxiliary literals (standing for unnamed atoms, rule bodies, or aggregates) having no symbolic representation.

The needed modifications are twofold, since literals in conflict nogoods are either obtained by a choice operation or by unit propagation. On the one hand, enforcing named choice literals can be done by existing means, namely, the heuristic capacities of *clasp*. To this end, it is enough to instruct *clasp* to strictly prefer atoms in the symbol table (declared via `#show` statements) for nondeterministic choices.[1]

On the other hand, enforcing learned constraints with named literals only needs changes to *clasp*'s internal conflict resolution scheme. In fact, *clasp*, as many other ASP and SAT solvers,

[1] This is done by launching *clasp* with the options `--heuristic=domain --dom-mod=1,16`.

uses the *first unique implication point* (1UIP) scheme [20]. In this scheme, the original conflict nogood is transformed by successive resolution steps into another conflict nogood containing only a single literal from the decision level at which the conflict occurred. This is either the last choice literal or a literal obtained by subsequent propagation. Each resolution step takes a conflict nogood δ containing a literal ℓ and resolves it with a reason ϵ for ℓ, resulting in the conflict nogood $(\delta \setminus \{\ell\}) \cup (\epsilon \setminus \{\overline{\ell}\})$. We rely upon this mechanism for eliminating unnamed literals from conflict nogoods. To this end, we follow the 1UIP scheme but additionally resolve out all unnamed (propagated) literals. We first derive a conflict nogood with a single *named* literal from the conflicting decision level and then resolve out all unnamed literals from other levels. As with 1UIP, the strategy is to terminate resolution as early as possible. In the best case, all literals are named and we obtain the same conflict nogood as with 1UIP. In the worst case, all propagated literals are unnamed and thus resolved out. This yields a conflict nogood comprised of choice literals, whose naming is enforced as described above.[2] Hence, provided that the set of named atoms is sufficient to generate a complete assignment by propagation, our approach guarantees all conflict nogoods to be composed of named literals. Finally, each resulting conflict nogood $\{\ell_1, \dots, \ell_n\}$ is output as an integrity constraint "$\leftarrow \ell_1, \dots, \ell_n$."

Eliminating unnamed literals burdens conflict analysis with additional resolution steps that result in weaker conflict nogoods and heuristic scores. To quantify this, we conducted experiments contrasting solving times with *clasp*'s 1UIP scheme and our named variant, with and without the above heuristic modification (yet without logging conflict constraints). We ran the configurations up to 600 seconds on each of the 100 instances of track 1 of the 2015 ASP competition. Timeouts were accounted for as 600 seconds. *clasp*'s default configuration solved 70 instances in 28 014 seconds, while the two named variants solved 65 in 29 982 and 63 in 29 700 seconds, respectively. Additionally, we ran all configurations on the 42 instances of our experiments in Section 5. While *clasp* solved all instances in 5596 seconds, the two named variants solved 22 in 16 133 and 16 in 17 607 seconds, respectively. Given that these configurations are meant to be used offline, we consider this loss as tolerable.

3.2 Selection

In view of the vast amount of learnable constraints, it is indispensable to select a restricted subset for constraint feedback. To this end, we allow for selecting a given number of constraints satisfying certain properties. We consider the

1. length of constraints (longest vs. shortest),
2. number of decision levels associated with their literals[3] (highest vs. lowest), and
3. time of recording (first vs. last).

To facilitate the selection, *xclasp* initially records all learned conflict constraints (within a time limit), and the *ginkgo* system then picks the ones of interest downstream.

The simplest form of reusing learned constraints consists of enriching an instance with subsumption-free propositional integrity constraints extracted from a previous run on the same instance. We refer to this as *direct constraint feedback*. We empirically studied the impact of this feedback method along with the various selection options in [19] and for brevity only summarize our results here. Our experiments indicate that direct constraint feedback

[2] This worst-case scenario corresponds to the well-known *decision scheme*, using conflict clauses containing choice literals only (obtained by resolving out all propagated literals). Experiments with a broad benchmark set [19] showed that our named 1UIP-based scheme uses only 41 % of the time needed with the decision scheme.

[3] This is known as the *literal block distance* (LBD).

generally improves performance and leads to no substantial degradation. This applies to runtime but also to the number of conflicts and decisions. We observed that solving times decrease with the number of added constraints,[4] except for two benchmark classes[5] showing no pronounced effect. This provided us with the pragmatic insight that the addition of constraints up to a magnitude of 10 000 does not hamper solving. The analysis of the above criteria yielded that (1) preferring short constraints had no negative effect over long ones but sometimes led to significant improvements, (2) the number of decision levels had no significant impact, with a slight advantage for constraints with fewer ones, and (3) the moment of extraction ranks equally well, with a slight advantage for earlier extracted constraints. All in all, we observe that even this basic form of constraint feedback can have a significant impact on ASP solving, though its extent is hard to predict. This is not as obvious as it might seem, since the addition of constraints slows down propagation, and initially added constraints might not yet be of value at the beginning of solving.

3.3 Generalization

The last section indicated the prospect of improving solver performance through constraint feedback. Now, we take this idea one step further by generalizing the learned constraints before feeding them back. The goal of this is to extend the applicability of extracted information and make it more useful to the solver ultimately. To this end, we proceed in two steps. First, we produce candidates for generalized conflict constraints from learned constraints. But since the obtained candidates are not necessarily valid, they are subject to validation. Invalid candidates are rejected, valid ones are kept. We consider two ways of generalization, namely, minimization and abstraction. *Minimization* eliminates as many literals as possible from conflict constraints. The smaller a constraint, the more it prunes the search space. *Abstraction* consists of replacing designated constants in conflict constraints by variables. This allows for extending the validity of a conflict constraint from a specific object to all objects of the same domain. This section describes generalization by minimization and abstraction, while the validation of generalized constraints is detailed in Section 3.4.

3.3.1 Minimization

Minimization aims at finding a minimal subset of a conflict constraint that still constitutes a conflict. Given that we extract conflicts in the form of integrity constraints, this amounts to eliminating as many literals as possible. For example, when solving a Ricochet Robots puzzle encoded by a program P, our extended solver *xclasp* might extract the integrity constraint

$$\leftarrow {\sim}go(red, up, 3), go(red, up, 4), {\sim}go(red, left, 5) \tag{2}$$

This established conflict constraint tells us that $P \cup \{h \leftarrow C, \leftarrow {\sim}h\}$ is unsatisfiable for $C = \{{\sim}go(red, up, 3), go(red, up, 4), {\sim}go(red, left, 5)\}$. The minimization task then consists of determining some minimal subset C' of C such that $P \cup \{h \leftarrow C', \leftarrow {\sim}h\}$ remains unsatisfiable, which in turn means that no answer set of P entails all of the literals in C'.

To traverse (proper) subsets C' of C serving as candidates, our *ginkgo* system pursues a greedy approach that aims at eliminating literals one by one. For instance, given C as above, it may start with $C' = C \setminus \{{\sim}go(red, up, 3)\}$ and check whether $P \cup \{h \leftarrow C', \leftarrow {\sim}h\}$ is

[4] We varied the number of extracted constraints from 8 to 16 384 in steps of factor $\sqrt{2}$.
[5] These classes consist of Solitaire and Towers of Hanoi puzzles.

unsatisfiable. If so, "$\leftarrow C'$" is established as a valid integrity constraint; otherwise, the literal $\sim go(red, up, 3)$ cannot be eliminated. Hence, depending on the result, either $C' \setminus \{\ell\}$ or $C \setminus \{\ell\}$ is checked next, where ℓ is one of the remaining literals $go(red, up, 4)$ and $\sim go(red, left, 5)$. Then, (un)satisfiability is checked again for the selected literal ℓ, and ℓ is either eliminated or not before proceeding to the last remaining literal.

Clearly, the minimal subset C' determined by this greedy approach depends on the order in which literals are selected to check and possibly eliminate them. Moreover, checking whether $P \cup \{h \leftarrow C', \leftarrow \sim h\}$ is unsatisfiable can be hard, and in case P itself is unsatisfiable, eventually taking $C' = \emptyset$ amounts to solving the original problem. The proof methods of *ginkgo*, described in Section 3.4, refer to problem relaxations to deal with the latter issue.

3.3.2 Abstraction

Abstraction aims at deriving candidate conflict constraints by replacing constants in ground integrity constraints with variables covering their respective domains. For illustration, consider integrity constraint (2) again. While this constraint is specific to a particular robot (*red*), it might also be valid for all the other available robots:

$$\leftarrow robot(R), \sim go(R, up, 3), go(R, up, 4), \sim go(R, left, 5)$$

Here, the predicate *robot* delineates the domain of robot identifiers. Further candidates can be obtained by extending either direction *up* or *left* to any possible direction. In both cases, we extend the scope of constraints from objects to unstructured domains.

Unlike this, the third parameter of the *go* predicate determines the time step at which the robot moves and belongs to the ordered domain of nonnegative integers. Thus, the conflict constraint might be valid for any sequence of points in time, given by the predicate *time*:

$$\leftarrow time(T), time(T+1), time(T+2), \sim go(red, up, T), go(red, up, T+1), \sim go(red, left, T+2)$$

The time domain is of particular interest when it comes to checking candidates, since it allows for identifying invariants in transition systems (see Section 3.4). This is a reason why the current prototype of *ginkgo* focuses on abstracting temporal constants to variables. In fact, *ginkgo* extracts all time points $t_1, \ldots, t_n$ in a constraint in increasing order and replaces them by $T, T + (t_2 - t_1), \ldots, T + (t_n - t_1)$, where T is a variable and $t_i < t_{i+1}$ for $0 < i < n$. We refer to integrity constraints obtained by abstraction over a domain of time steps as *temporal constraints*, denote them by "$\leftarrow C[T]$," where T is the introduced temporal variable, and refer to the difference $t_n - t_1$ as the *degree.*

3.4 Validation

Validating an integrity constraint is about showing that it holds in all answer sets of a logic program. To this end, we use counterexample-oriented methods that can be realized in ASP. Although the respective approach at the beginning of Section 3.3.1 is universal, as it applies to any program, it has two drawbacks. First, it is instance-specific, and second, proof attempts face the hardness of the original problem. With hard instances, as encountered in planning, this is impracticable, especially when checking many candidates. Also, proofs neither apply to other instances of the same planning problem nor carry over to different horizons (plan lengths). To avoid these issues, we pursue a problem-specific approach by concentrating on invariants of transition systems (induced by planning problems). Accordingly, we restrict ourselves to temporal abstractions, as described in Section 3.3, and require problem-specific information, such as state and action variables.

In what follows, we develop two ASP-based proof methods for validating candidates in problems based on transition systems. We illustrate the proof methods below for sequential planning and detail their application in Section 4. We consider *planning problems* consisting of a set F of fluents and a set A of actions, along with *instances* containing an initial state I and a goal condition. Letting $A[t]$ and $F[t]$ stand for action and fluent variables at time step t, a set $I[0]$ of facts over $F[0]$ represents the initial state and a logic program $P[t]$ over $A[t]$ and $F[t-1] \cup F[t]$ describes the transitions induced by the actions of a planning problem (cf. [17]). That is, the two validation methods presented below and corresponding ASP encodings given in [9] do not rely on the goal.

3.4.1 Inductive Method

The idea of using ASP for conducting proofs by induction traces back to verifying properties in game descriptions [14]. To show that a temporal constraint "$\leftarrow C[T]$" of degree k is invariant to a planning problem represented by $I[0]$ and $P[t]$, two programs must be unsatisfiable:

$$I[0] \;\cup\; P[1] \cup \cdots \cup P[k] \;\cup\; \{h(0) \leftarrow C[0], \leftarrow {\sim} h(0)\} \quad (3)$$

$$S[0] \;\cup\; P[1] \cup \cdots \cup P[k+1] \;\cup\; \{h(0) \leftarrow C[0], \leftarrow h(0)\} \;\cup\; \{h(1) \leftarrow C[1], \leftarrow {\sim} h(1)\} \quad (4)$$

Program (3) captures the induction base and rejects a candidate if it is satisfied (starting) at time step 0. Note that when a constraint spans k different time points, all trajectories of length k starting from the initial state are examined.

The induction step is captured in program (4) by using a program $S[0]$ for producing all possible predecessor states (marked by "0"). To this end, $S[0]$ contains a choice rule "$\{f(0)\} \leftarrow$" for each fluent $f(0)$ in $F[0]$. Moreover, program (4) rejects a candidate if the predecessor state (starting at time step 0) violates the candidate or if the successor state (starting at 1) satisfies it. To apply the candidate to the successor step, it is shifted by 1 via $h(1)$. That is, the induction step requires one more time step than the base. If both programs (3) and (4) are unsatisfiable, the candidate is validated. Although the obtained integrity constraint depends on the initial state, it is independent of the goal and applies to varying horizons. Hence, the generalized constraint cannot only be used for enriching the planning instance at hand but also carries over to instances with different horizons and goals.

3.4.2 State-Wise Method

We also consider a simpler validation method that relies on exhaustive state generation. This approach replaces the two-fold induction method with a single search for counterexamples:

$$S[0] \;\cup\; P[1] \cup \cdots \cup P[k] \;\cup\; \{h(0) \leftarrow C[0], \leftarrow {\sim} h(0)\} \quad (5)$$

As in the induction step above, a state is nondeterministically generated via $S[0]$. But instead of performing the step, program (5) rejects a candidate if it is satisfied in the generated state. As before, the candidate is validated if program (5) is unsatisfiable. While this simple proof method is weaker than the inductive one, it is independent of the initial state, and validated generalized constraints thus carry over to all instances of a planning problem. We empirically contrast both approaches in Section 5.

3.5 Feedback

Combining all the previously described steps allows us to enrich logic programs with validated generalized integrity constraints. We call this process *generalized constraint feedback*.

The scope of our approach is delineated by the chosen proof methods. First, they deal with problems based on transition systems. Second, both methods are incomplete, since they might find infeasible counterexamples stemming from unreachable states. However, both methods rely on relatively inexpensive proofs, since candidates are bound by their degree rather than the full horizon. This also makes valid candidates independent of goal conditions and particular horizons; state-wise proven constraints are even independent of initial states.

4 Implementation

We implemented our knowledge generalization framework as two systems: *xclasp* is a variant of the ASP solver *clasp* 3 capable of extracting learned constraints while solving, and the extracted constraints are then automatically generalized and validated offline by *ginkgo*. In this way, *ginkgo* produces generalized constraints that can be reused through generalized constraint feedback. Both *xclasp* and *ginkgo* are available at the Potassco Labs website.[6]

4.1 *xclasp*

xclasp implements the instrumentation described in Section 3.1 as a standalone variant of *clasp* 3.1.4 extended by constraint extraction. The option `--log-learnts` outputs learned integrity constraints so that the output can be readily used by any downstream application. The option `--logged-learnt-limit=n` stops solving once `n` constraints were logged. Finally, the named-literals resolution scheme is invoked with `--resolution-scheme=named`.

4.2 *ginkgo*

ginkgo incorporates the different techniques developed in Section 3. After recording learned constraints, *ginkgo* offers postprocessing steps, one of which is *sorting* logged constraints by multiple criteria. This option is interesting for analyzing the effects of reusing different types of constraints. Another postprocessing step used throughout this paper is (propositional) *subsumption*, that is, removing subsumed constraints. In fact, *xclasp* often learns constraints that are subsets of previous ones (and thus more general and stronger). For example, when solving the 3-Queens puzzle, recorded integrity constraints might be subsumed by "$\leftarrow queen(2,2)$," as a single queen in the middle attacks entire columns and rows.

Figure 1 illustrates *ginkgo*'s generalization procedure. In our setting, the input to *ginkgo* consists of a planning problem, an instance, and a fixed horizon. First, *ginkgo* begins to extract a specified number of learned constraints by solving the instance with our modified solver *xclasp*. Then, *ginkgo* abstracts the learned constraints over the time domain, which results in a set of candidates (see Section 3.3.2). These candidates are validated and optionally minimized (see Section 3.3.1) one by one. For this purpose, *ginkgo* uses either of the two presented validation methods (see Section 3.4), where the candidates are validated in ascending order of degree. This is sensible because the higher the degree, the larger is the search space for counterexamples. Among candidates with the same degree, the ones with fewer literals are tested first, given that the optional minimization of constraints (using the same proof method as for validation) requires less steps for them. Moreover, proven candidates are immediately added to the input logic program in order to strengthen future proofs (while unproven ones are discarded). Finally, *ginkgo* terminates after successfully generalizing a user-specified

[6] `http://potassco.sourceforge.net/labs.html`

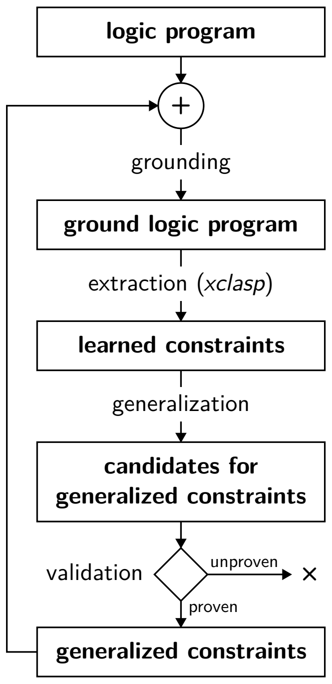


Figure 1 *ginkgo*'s procedure for automatically generalizing learned constraints.

number of constraints. The generalized constraints can then be used to enrich the same or a related logic program via generalized constraint feedback.

ginkgo offers multiple options to steer the constraint generalization procedure. `--horizon` specifies the planning horizon. The validation method is selected via `--proof-method`. `--minimization-strategy` defines whether constraint minimization is used. `--constraints-to-extract` decides how many constraints *ginkgo* extracts before starting to validate them, where the extraction step can also be limited with `--extraction-timeout`. By default, *ginkgo* tests all initially extracted constraints before extracting new ones.[7] Alternatively, new constraints may be extracted after each successful proof (controlled via `--testing-policy`). Candidates exceeding a specific degree (`--max-degree`) or number of literals (`--max-number-of-literals`) may be skipped. Additionally, candidates may be skipped if the proof takes too long (`--hypothesis-testing-timeout`). *ginkgo* terminates after generalizing a number of constraints specified by `--constraints-to-prove` (or if *xclasp*'s constraints are exhausted).

5 Evaluation

To evaluate our approach, we instruct *ginkgo* to learn and generalize constraints autonomously on a set of benchmark instances. These instances stem from the International Planning Competition (IPC) series and were translated to ASP with *plasp* [10].

First, we study how the solver's runtime is affected by generalized constraint feedback – that is, enriching instances with generalized constraints that were obtained beforehand with *ginkgo*. In a second experiment, generalized constraint feedback is performed after *varying*

[7] Note that extracting more constraints is only necessary if the initial chunk of learned constraints does not lead to the requested number of generalized constraints. In practice, this rarely happens when choosing a sufficient number of constraints to extract initially.

Table 1 Configurations of *ginkgo* for studying generalized constraint feedback.

	validation method	minimization	constraint feedback
(a)	state-wise	on	generalized
(b)	inductive	on	generalized
(c)	state-wise	off	generalized
(d)	state-wise	on	direct

the instances' horizons. Among other things, this allows us to study scenarios in which constraints are first generalized using *simplified* settings to speed up the solving process of the *actual* instances later on. The benchmark sets are available at *ginkgo*'s website.[6]

5.1 Generalized Constraint Feedback

In this experiment, we use *ginkgo* to generalize a specific number of learned constraints for each instance. Then, we enrich the instances via generalized constraint feedback and measure how the enriched instances relate to the original ones in terms of runtime. This setup allows us to assess whether reusing generalized constraints improves solving the individual instances.

The benchmark set consists of 42 instances from the 2000, 2002, and 2006 IPCs and covers nine planning domains: Blocks World (8), Driver Log (4), Elevator (11), FreeCell (4), Logistics (5), Rovers (1), Satellite (3), Storage (4), and Zeno Travel (2). We selected instances with solving times within 10 to 600 seconds on the benchmark system (using *clasp* with default settings). For 33 instances, we used minimal horizons. We chose higher horizons for the remaining nine instances because timeouts occurred with minimal horizons.

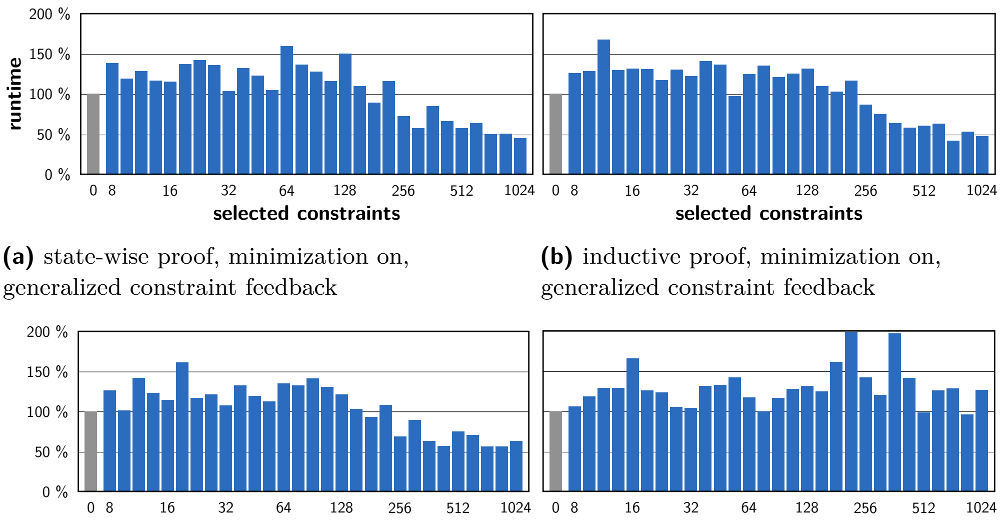
Given an instance and a fixed horizon, 1024 generalized constraints are first generated offline with *ginkgo*. Afterward, the solving time of the instance is measured multiple times. Each time, the instance is enriched with the first n generalized constraints, where n varies between 8 and 1024 in exponential steps. The original instance is solved once more without any feedback for reference. Afterward, the runtimes of the enriched instances are compared to the original ones. All runtimes are measured with *clasp*'s default configuration, not *xclasp*.

We perform this experiment with the four *ginkgo* configurations shown in Table 1.

First, we select the state-wise proof method and enable minimization (a). We chose this configuration as a reference because the state-wise proof method achieves instance independence (see Section 3.4.2) and because minimization showed to be useful in earlier experiments [19]. To compare the two validation methods presented in this paper, we repeat the experiment with the inductive proof method (b). In configuration (c), we disable constraint minimization to assess the benefit of this technique. Finally, configuration (d) replaces generalized with direct constraint feedback (that is, the instances are not enriched with the generalized constraints but the ground learned constraints they stem from). With configuration (d), we can evaluate whether generalization renders learned constraints more useful.

We fix *ginkgo*'s other options across all configurations. Generalization starts after *xclasp* extracted 16 384 constraints or after 600 seconds. Candidates with degrees greater than 10 or more than 50 literals are skipped, and proofs taking more than 10 seconds are aborted. After *ginkgo* terminates, the runtimes of the original and enriched instances are measured with a limit of 3600 seconds. Timeouts are penalized with PAR-10 (36 000 seconds). The benchmarks were run on a Linux machine with Intel Core i7-4790K at 4.4 GHz and 16 GB RAM.

As Figure 2a shows, generalized constraint feedback reduced the solver's runtime by up to 55 %. The runtime decreases the more generalized constraints are selected for feedback.



(a) state-wise proof, minimization on, generalized constraint feedback

(b) inductive proof, minimization on, generalized constraint feedback

(c) state-wise proof, minimization off, generalized constraint feedback

(d) state-wise proof, minimization on, direct constraint feedback

Figure 2 Runtimes after generalized constraint feedback with four different *ginkgo* configurations.

On average, validating a candidate constraint took 73 ms for grounding and 22 ms for solving in reference configuration (a). 38 % of all proofs were successful, and *ginkgo* terminated after 1169 seconds on average. The tested candidates had an average degree of 2.2 and contained 9.3 literals. Constraint minimization eliminated 63 % of all literals in generalized constraints.

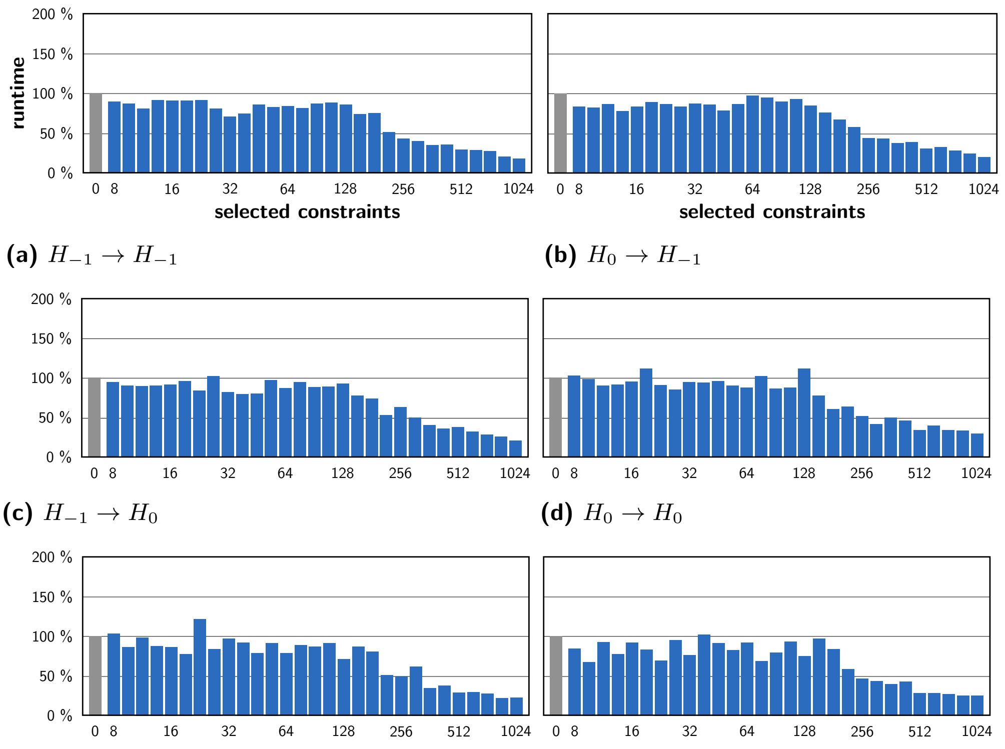
While lacking the instance independence of the state-wise proof, the supposedly stronger inductive proof did not lead to visibly different results (see Figure 2b). Additionally, validating candidate constraints took about 2.3 times longer. With 2627 seconds, the average total runtime of *ginkgo* was 2.2 times higher with the inductive proof method. Disabling constraint minimization had rather little effect on the generalized constraints' utility in terms of solver runtime, as seen in Figure 2c. However, without constraint minimization, *ginkgo*'s runtime was reduced to 332 seconds (a factor of 3.5 compared to the reference configuration). Interestingly, direct constraint feedback was never considerably useful for the solver (see Figure 2d). Hence, we conclude that learned constraints are indeed strengthened by generalization.

5.2 Generalized Constraint Feedback with Varying Horizons

This experiment evaluates the *generality* of the proven constraints – that is, whether priorly generalized constraints improve the solving performance on similar instances. For this purpose, we use *ginkgo* to extract and generalize constraints on the benchmark instances with fixed horizons. Then, we vary the horizons of the instances and solve them again, after enriching them with the previously generalized constraints.

We reuse the 33 instances with minimal (optimal) horizons from Section 5.1, referring to them as the H_0 set. In addition, we analyze two new benchmark sets. H_{-1} consists of the H_0 instances with horizons reduced by 1, which renders all instances in H_{-1} unsatisfiable. In another benchmark set, H_{+1}, we increase the fixed horizon of the H_0 instances by 1.[8]

[8] The alleged small change of the horizon by 1 is motivated by maintaining the hardness of the problem.



(a) $H_{-1} \rightarrow H_{-1}$ **(b)** $H_0 \rightarrow H_{-1}$

(c) $H_{-1} \rightarrow H_0$ **(d)** $H_0 \rightarrow H_0$

(e) $H_{-1} \rightarrow H_{+1}$ **(f)** $H_0 \rightarrow H_{+1}$

Figure 3 Runtimes after generalized constraint feedback with varied horizons. In setting $H_x \rightarrow H_y$, constraints were extracted and generalized with benchmark set H_x and reused for solving H_y.

The benchmark procedure is similar to Section 5.1. This time, constraints are extracted and generalized on a specific benchmark set but then applied to the corresponding instances of another set. For instance, $H_{-1} \rightarrow H_0$ refers to the setting where constraints are generalized with H_{-1} and then reused while solving the respective instances in H_0. In total, we study six settings: $\{H_{-1}, H_0\} \rightarrow \{H_{-1}, H_0, H_{+1}\}$. The choice of $\{H_{-1}, H_0\}$ as sources reflects the extraction from unsatisfiable and satisfiable instances, respectively. To keep the results comparable across all configurations, we removed five instances whose reference runtime (without feedback) exceeded the time limit of 3600 seconds in at least one of H_{-1}, H_0, and H_{+1}. For this reason, the results shown in Figure 3 refer to the remaining 28 instances. In this experiment, the state-wise validation method and minimization are applied. The benchmark environment is identical to Section 5.1.

As Figure 3 shows, generalized constraint feedback permits varying the horizon with no visible penalty.

Across all six settings, the runtime improvements are very similar (up to 70 or 82 %, respectively). Runtime improvements are somewhat more pronounced when constraints are generalized with H_{-1} rather than H_0. Furthermore, generalized constraint feedback on H_{-1} is slightly more useful than on H_0 and H_{+1}. Apart from this, generalized constraint feedback seems to work well no matter whether the programs at hand are satisfiable or not.

6 Discussion

We have presented the systems *xclasp* and *ginkgo*, jointly implementing a fully automated form of generalized constraint feedback for CDCL-based ASP solvers. This is accomplished in a four-phase process consisting of extraction (and selection), generalization (via abstraction and minimization), validation, and feedback. While *xclasp*'s extraction of integrity constraints is domain-independent, the scope of *ginkgo* is delineated by the chosen proof method. Our focus on inductive and state-wise methods allowed us to study the framework in the context of transition-based systems, including the chosen application area of automated planning. We have demonstrated that our approach allows for reducing the runtime of planning problems by up to 55 %. Moreover, the learned constraints cannot only be used to accelerate a program at hand, but they moreover transfer to other goal situations, altered horizons, and even other initial situations (with the state-wise technique). In the latter case, the learned constraints are general enough to apply to all instances of a fixed planning problem. Interestingly, while both proof methods often failed to prove valid, handcrafted properties, they succeeded on relatively many automatically extracted candidates (about 38 %). Generally speaking, it is worthwhile to note that our approach had been impossible without ASP's first-order modeling language along with its distinction of problem encoding and instance.

Although *xclasp* and *ginkgo* build upon many established techniques, we are unaware of any other approach combining the same spectrum of techniques similarly. In ASP, the most closely related work was done in [26] in the context of the first-order ASP solver *omiga* [6]. Rules are represented as Rete networks, propagation is done by firing rules, and unfolding is used to derive new reusable rules. ASP-based induction was first used for verifying predefined properties in game descriptions [14]. Inductive logic programming in ASP [22, 16] is related in spirit but works from different principles, such as deriving rules compatible with positive and negative examples. In SAT, k-induction [24, 25] is a wide-spread technique in applications to model checking. Our state-wise proof method is similar to 0-induction. In FO(ID), [7] deals with detecting functional dependencies for deriving new constraints, where a constraint's validity is determined by a first-order theorem prover. In CP, automated modeling constitutes an active research area (cf. [21]). For instance, [5] addresses constraint reformulation by resorting to machine learning and theorem proving for extraction and validation. Finally, invariants in transition systems have been explored in several fields, among them general game playing [14], planning [23, 15], model checking [24, 25], and reasoning about actions [18]. While inductive and first-order proof methods are predominant, invariants are either assumed to be given or determined by dedicated algorithms.

Our approach aims at overcoming the restriction of learned knowledge to specific problem instances. However, it may also help to close the gap between highly declarative and highly optimized encodings by enriching the former through generalized constraint feedback.

References

1 M. Alviano, C. Dodaro, N. Leone, and F. Ricca. Advances in WASP. In F. Calimeri, G. Ianni, and M. Truszczyński, editors, *Proceedings of the Thirteenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'15)*, pages 40–54. Springer, 2015.

2 A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

3 G. Brewka, T. Eiter, and M. Truszczyński. Answer set programming at a glance. *Communications of the ACM*, 54(12):92–103, 2011.

4 F. Calimeri, W. Faber, M. Gebser, G. Ianni, R. Kaminski, T. Krennwallner, N. Leone, F. Ricca, and T. Schaub. ASP-Core-2: Input language format. Available at `https://www.mat.unical.it/aspcomp2013/ASPStandardization/`, 2012.

5 J. Charnley, S. Colton, and I. Miguel. Automated generation of implied constraints. In G. Brewka, S. Coradeschi, A. Perini, and P. Traverso, editors, *Proceedings of the Seventeenth European Conference on Artificial Intelligence (ECAI'06)*, pages 73–77. IOS Press, 2006.

6 M. Dao-Tran, T. Eiter, M. Fink, G. Weidinger, and A. Weinzierl. OMiGA : An open minded grounding on-the-fly answer set solver. In L. Fariñas del Cerro, A. Herzig, and J. Mengin, editors, *Proceedings of the Thirteenth European Conference on Logics in Artificial Intelligence (JELIA'12)*, pages 480–483. Springer, 2012.

7 B. De Cat and M. Bruynooghe. Detection and exploitation of functional dependencies for model generation. *Theory and Practice of Logic Programming*, 13(4-5):471–485, 2013.

8 R. Dechter. *Constraint Processing*. Morgan Kaufmann Publishers, 2003.

9 M. Gebser, R. Kaminski, B. Kaufmann, P. Lühne, J. Romero, and T. Schaub. Answer set solving with generalized learned constraints (extended version). Available at `http://www.cs.uni-potsdam.de/wv/publications/`, 2016.

10 M. Gebser, R. Kaminski, M. Knecht, and T. Schaub. plasp: A prototype for PDDL-based planning in ASP. In J. Delgrande and W. Faber, editors, *Proceedings of the Eleventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11)*, pages 358–363. Springer, 2011.

11 M. Gebser, B. Kaufmann, and T. Schaub. Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence*, 187-188:52–89, 2012.

12 M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.

13 E. Giunchiglia, Y. Lierler, and M. Maratea. Answer set programming based on propositional satisfiability. *Journal of Automated Reasoning*, 36(4):345–377, 2006.

14 S. Haufe, S. Schiffel, and M. Thielscher. Automated verification of state sequence invariants in general game playing. *Artificial Intelligence*, 187-188:1–30, 2012.

15 M. Helmert. Concise finite-domain representations for PDDL planning tasks. *Artificial Intelligence*, 173(5-6):503–535, 2009.

16 M. Law, A. Russo, and K. Broda. Inductive learning of answer set programs. In E. Fermé and J. Leite, editors, *Proceedings of the Fourteenth European Conference on Logics in Artificial Intelligence (JELIA'14)*, pages 311–325. Springer, 2014.

17 V. Lifschitz. Answer set programming and plan generation. *Artificial Intelligence*, 138(1-2):39–54, 2002.

18 F. Lin. Discovering state invariants. In D. Dubois, C. Welty, and M. Williams, editors, *Proceedings of the Ninth International Conference on Principles of Knowledge Representation and Reasoning (KR'04)*, pages 536–544. AAAI Press, 2004.

19 P. Lühne. Generalizing learned knowledge in answer set solving. Master's thesis, Hasso Plattner Institute, Potsdam, 2015.

20 J. Marques-Silva and K. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

21 B. O'Sullivan. Automated modelling and solving in constraint programming. In M. Fox and D. Poole, editors, *Proceedings of the Twenty-fourth National Conference on Artificial Intelligence (AAAI'10)*, pages 1493–1497. AAAI Press, 2010.

22 R. Otero. Induction of stable models. In C. Rouveirol and M. Sebag, editors, *Proceedings of the Eleventh International Conference on Inductive Logic Programming (ILP'01)*, pages 193–205. Springer, 2001.

23 J. Rintanen. An iterative algorithm for synthesizing invariants. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI'00)*, pages 806–811. AAAI/MIT Press, 2000.

24 M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In W. Hunt and S. Johnson, editors, *Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design (FMCAD'00)*, pages 108–125. Springer, 2000.

25 Y. Vizel, G. Weissenbacher, and S. Malik. Boolean satisfiability solvers and their applications in model checking. *Proceedings of the IEEE*, 103(11):2021–2035, 2015.

26 A. Weinzierl. Learning non-ground rules for answer-set solving. In D. Pearce, S. Tasharrofi, E. Ternovska, and C. Vidal, editors, *Proceedings of the Second Workshop on Grounding and Transformation for Theories with Variables (GTTV'13)*, pages 25–37, 2013.

PρLog: Combining Logic Programming with Conditional Transformation Systems (Tool Description)*

Besik Dundua[1], Temur Kutsia[2], and Klaus Reisenberger-Hagmayer[3]

1 **Vekua Institute of Applied Mathematics, Tbilisi State University, Tbilisi, Georgia**
2 **RISC, Johannes Kepler University Linz, Linz, Austria**
3 **Johannes Kepler University Linz, Linz, Austria**

Abstract

PρLog extends Prolog by conditional transformations that are controlled by strategies. We give a brief overview of the tool and illustrate its capabilities.

1998 ACM Subject Classification D.1.6 Logic Programming, F.4.2 Grammars and Other Rewriting Systems, D.3.2 Language Classifications

Keywords and phrases Conditional transformation rules, strategies, Prolog

Digital Object Identifier 10.4230/OASIcs.ICLP.2016.10

1 Brief overview

PρLog is a tool that combines, on the one hand, the power of logic programming and, on the other hand, flexibility of strategy-based conditional transformation systems. Its terms are built over function symbols without fixed arity, using four different kinds of variables: for individual terms, for sequences of terms, for function symbols, and for contexts. These variables help to traverse tree forms of expressions both in horizontal and vertical directions, in one or more steps. A powerful matching algorithm helps to replace several steps of recursive computations by pattern matching, which facilitates writing short and intuitively quite clear code. By the backtracking engine, nondeterministic computations are modeled naturally. Prolog's meta-programming capabilities allowed to easily write a compiler from PρLog programs (that consist of a specific Prolog code, actually) into pure Prolog programs.

PρLog program clauses either define user-constructed strategies by transformation rules or are ordinary Prolog clauses. Prolog code can be used freely in PρLog programs, which is especially convenient when built-ins, arithmetics, or input-output features are needed.

PρLog is based on the ρLog calculus [15], whose inference system is basically the SLDNF-resolution, with normal logic program semantics [14]. Therefore, Prolog was a natural choice to implement it. The ρLog calculus has been influenced by the ρ-calculus [5], which, in itself, is a foundation for the rule-based programming system ELAN [2]. There are some other languages for programming by rules, such as, e.g., ASF-SDF [16], CHR [11], Claire [4], Maude [6], Stratego [17], Tom [1]. The ρLog calculus and, consequently, PρLog differs from

* This research is partially supported by the Austrian Science Fund (FWF) under the projects P 24087-N18 and P 28789-N32, and by the Rustaveli National Science Foundation under the grants FR/508/4-120/14, FR/325/4-120/14, and YS15 2.1.2 70.

them, first of all, by its pattern matching capabilities. Besides, it adopts logic programming semantics (clauses are first class concepts, rules/strategies are expressed as clauses) and makes a heavy use of strategies to control transformations. We showed its applicability for XML transformation and Web reasoning [7], and in modeling rewriting strategies [9].

Here we briefly describe the current status of PρLog. A more detailed overview can be found in [10]. The system can be downloaded from its Web page `http://www.risc.jku.at/people/tkutsia/software/prholog/`. The current version has been tested for SWI-Prolog [18] version 7.2.3 or later.

2 How PρLog works

PρLog atoms are supposed to transform term sequences. Transformations are labeled by what we call *strategies*. Such labels (which themselves can be complex terms, not necessarily constant symbols) help to construct more complex transformations from simpler ones.

An instance of a transformation is finding duplicated elements in a sequence and removing one of them. We call this process double merging. The following strategy implements it:

$$\text{merge\_doubles} :: (s\_X,\ i\_x,\ s\_Y,\ i\_x,\ s\_Z) \Longrightarrow (s\_X,\ i\_x,\ s\_Y,\ s\_Z).$$

Here merge_doubles is the strategy name. It is followed by the separator :: which separates the strategy name from the transformation. Then comes the transformation itself in the form *lhs* $\Longrightarrow$ *rhs*. It says that if the sequence in *lhs* contains duplicates (expressed by two copies of the variable i_x, which can match individual terms and therefore, is called an *individual variable*) somewhere, then from these two copies only the first one should be kept in *rhs*. That "somewhere" is expressed by three sequence variables, where s_X stands for the subsequence of the sequence before the first occurrence of i_x, s_Y takes the subsequence between two occurrences of i_x, and s_Z matches the remaining part. These subsequences remain unchanged in the *rhs*. Note that one does not need to code the actual search process of doubles explicitly. The matching algorithm does the job instead, looking for an appropriate instantiation of the variables. There can be several such instantiations.

Now one can ask a question, e.g., to merge doubles in a sequence (1, 2, 3, 2, 1):

$$\text{?- merge\_doubles} :: (1,\ 2,\ 3,\ 2,\ 1) \Longrightarrow s\_Result.$$

PρLog returns two different substitutions: $\{s\_Result \mapsto (1,\ 2,\ 3,\ 2)\}$ and $\{s\_Result \mapsto (1,\ 2,\ 3,\ 1)\}$. They are computed via backtracking. Each of them is obtained from (1, 2, 3, 2, 1) by merging one pair of duplicates. A completely double-free sequence is just a normal form of this single-step transformation. PρLog has a built-in strategy for computing normal forms, denoted by **nf**, and we can use it to define a new strategy merge_all_doubles in the following clause (where :-, as in Prolog, stands for the inverse implication):

$$\text{merge\_all\_doubles} :: s\_X \Longrightarrow s\_Y \ \text{:-}\ \mathbf{nf}(\text{merge\_doubles}) :: s\_X \Longrightarrow s\_Y,\ !.$$

The effect of **nf** here is that it starts applying merge_doubles to s_X, and repeats this process iteratively as long as it is possible, i.e., as long as doubles can be merged in the obtained sequences. When merge_doubles is no more applicable, it means that the normal form of the transformation is reached and it is returned in s_Y. The Prolog cut at the end cuts the alternative ways of computing the same normal form. In general, Prolog primitives and clauses can be used freely in PρLog. Now, for the query

$$\text{?- merge\_all\_doubles} :: (1,\ 2,\ 3,\ 2,\ 1) \Longrightarrow s\_Result.$$

we get a single answer $s\_Result \mapsto (1, 2, 3)$. Instead of using the cut, we could have defined merge_all_doubles purely in PρLog terms, with the help of a built-in strategy **first_one**. It applies to a sequence of strategies (in the clause below there is only one such strategy, **nf**(merge_doubles)), finds the first one among them which successfully transforms the input sequence (s_X below), and gives back just *one result* of the transformation (in s_Y):

$$\text{merge\_all\_doubles} :: s\_X \Longrightarrow s\_Y \text{ :- } \mathbf{first\_one}(\mathbf{nf}(\text{merge\_doubles})) :: s\_X \Longrightarrow s\_Y.$$

PρLog is good not only in selecting arbitrarily many subexpressions in "horizontal direction" (by sequence variables), but also in working in "vertical direction", selecting subterms at arbitrary depth. *Context variables* provide this flexibility, by matching the context above the subterm to be selected. A context is a term with a single "hole" in it. When it applies to a term, the latter is "plugged in" the hole, replacing it. There is yet another kind of variable, called *function variable*, which stands for a function symbol. With the help of these constructs and the merge_doubles strategy, it is pretty easy to define a transformation that merges two identical branches in a tree, represented as a term:

$$\begin{aligned}&\text{merge\_double\_branches} :: c\_Con(f\_Fun(s\_X)) \Longrightarrow c\_Con(f\_Fun(s\_Y)) \text{ :-}\\ &\qquad \text{merge\_doubles} :: s\_X \Longrightarrow s\_Y.\end{aligned}$$

Here c_Con is a context variable and f_Fun is a function variable. This is a naming notation in PρLog, to start a variable name with the first letter of the kind of variable (*i*ndividual, *s*equence, *f*unction, *c*ontext), followed by the underscore. After the underscore, there comes the actual name. For anonymous variables, we write just $i\_$, $s\_$, $f\_$, $c\_$.

Now, we can ask to merge double branches in a given tree:

$$\text{?- merge\_double\_branches} :: f(g(a, b, a, h(c, c)), g(a, b, h(c))) \Longrightarrow i\_Result.$$

PρLog returns two different substitutions via backtracking:

$$\{i\_Result \mapsto f(g(a, b, h(c, c)), g(a, b, h(c)))\},$$
$$\{i\_Result \mapsto f(g(a, b, a, h(c)), g(a, b, h(c)))\}.$$

To obtain the first one, c_Con matched to the context $f(\circ, g(a, b, h(c)))$ (where $\circ$ is the hole), f_Fun to the symbol g, and s_X to the sequence $(a, b, a, h(c, c))$. merge_doubles transformed $(a, b, a, h(c, c))$ to $(a, b, h(c, c))$. The other result is obtained by matching c_Con to $f(g(a, b, a, \circ), g(a, b, h(c)))$, f_Fun to h, s_X to (c, c), and merging the c's in the latter.

One can have an arbitrary sequence (not necessarily a variable) in the right hand side of transformations in the queries, e.g., instead of i_Result above we could have had $c\_C(h(c, c))$, asking for the context of the result that contains $h(c, c)$. Then the output would be $\{c\_C \mapsto f(g(a, b, \circ), g(a, b, h(c)))\}$.

Similar to merging all doubles in a sequence above, we can also define a strategy that merges all identical branches in a tree repeatedly, as **first_one**(**nf**(merge_double_branches)). It would give $f(g(a, b, h(c)))$ for the input term $f(g(a, b, a, h(c, c)), g(a, b, h(c)))$.

PρLog execution principle is based on depth-first inference with leftmost literal selection in the goal. If the selected literal is a Prolog literal, then it is evaluated in the standard way. If it is a PρLog atom of the form $st :: \tilde{s}_1 \Longrightarrow \tilde{s}_2$, due to the syntactic restriction called well-modedness (formally defined in [9]), st and $\tilde{s}_1$ do not contain variables. Then a (renamed copy of a) program clause $st' :: \tilde{s}'_1 \Longrightarrow \tilde{s}'_2$:- *body* is selected, such that st' matches st and $\tilde{s}'_1$ matches $\tilde{s}_1$ with a substitution σ. Next, the selected literal in the query is replaced with the conjunction $(body)\sigma$, **id** :: $\tilde{s}_2'\sigma \Longrightarrow \tilde{s}_2$, where **id** is the built-in strategy for identity: it succeeds iff its rhs matches the lhs. Evaluation continues further with this new query. Success and failure are defined in the standard way. Backtracking explores other alternatives that may come from matching the selected query literal to the head of the same program clause in a different way (since context/sequence matching is finitary, see, e.g., [8, 12, 13]), or to the head of another program clause. Negative literals are processed by negation-as-failure.

The PρLog distribution consists of the main file, parser, compiler, the library of built-in strategies, and a part responsible for matching. PρLog programs are written in files with the extension `.rho`. A PρLog session is initiated withing Prolog by consulting the main file. After that, the user can load a `.rho` file, which is parsed and compiled into a Prolog code. PρLog queries are also transformed into Prolog queries, which are then executed.

PρLog can be used in any development environment that is suitable for SWI-Prolog. We provide a special Emacs mode for PρLog, which extends the Prolog mode for Emacs [3]. It supports syntax highlighting, makes it easy to load PρLog programs and anonymize variables via the menu, etc. A tracing tool for PρLog is under development.

One can summarize the main advantages of PρLog as follows: compact and declarative code; capabilities of expression traversal without explicitly programming it; the ability to use clauses in a flexible order with the help of strategies. Besides, PρLog has access to the whole infrastructure of its underline Prolog system. These features make PρLog suitable for nondeterministic computations, manipulating XML documents, implementing rule-based algorithms and their control, etc.

References

1 Emilie Balland, Paul Brauner, Radu Kopetz, Pierre-Etienne Moreau, and Antoine Reilles. Tom: Piggybacking rewriting on Java. In Franz Baader, editor, *Term Rewriting and Applications, 18th International Conference, RTA 2007*, volume 4533 of *Lecture Notes in Computer Science*, pages 36–47. Springer, 2007.

2 Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Marian Vittek. Elan: A logical framework based on computational systems. *ENTCS*, 4, 1996.

3 Stefan D. Bruda. Prolog mode for Emacs (version 1.25), 2003. Available from `https://bruda.ca/emacs/prolog_mode_for_emacs`.

4 Yves Caseau, François-Xavier Josset, and François Laburthe. CLAIRE: combining sets, search and rules to better express algorithms. *TPLP*, 2(6):769–805, 2002.

5 Horatiu Cirstea and Claude Kirchner. The rewriting calculus - Parts I and II. *Logic Journal of the IGPL*, 9(3):339–410, 2001.

6 Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Jose F. Quesada. Maude: specification and programming in rewriting logic. *Theor. Comput. Sci.*, 285(2):187–243, 2002.

7 Jorge Coelho, Besik Dundua, Mário Florido, and Temur Kutsia. A rule-based approach to XML processing and web reasoning. In Pascal Hitzler and Thomas Lukasiewicz, editors, *RR 2010*, volume 6333 of *LNCS*, pages 164–172. Springer, 2010.

8 Hubert Comon. Completion of rewrite systems with membership constraints. Part II: Constraint solving. *J. Symb. Comput.*, 25(4):421–453, 1998.

9 Besik Dundua, Temur Kutsia, and Mircea Marin. Strategies in PρLog. In Maribel Fernández, editor, *9th Int. Workshop on Reduction Strategies in Rewriting and Programming, WRS 2009*, volume 15 of *EPTCS*, pages 32–43, 2009.

10 Besik Dundua, Temur Kutsia, and Klaus Reisenberger-Hagmayer. An overview of PρLog. RISC Report Series 16-05, Research Institute for Symbolic Computation, Johannes Kepler University Linz, Austria, 2016.

11 Thom W. Frühwirth. Theory and practice of Constraint Handling Rules. *J. Log. Program.*, 37(1-3):95–138, 1998.

12 Temur Kutsia. Solving and Proving in Equational Theories with Sequence Variables and Flexible Arity Symbols. RISC Report Series 02-09, RISC, University of Linz, 2002. PhD Thesis.

13 Temur Kutsia and Mircea Marin. Matching with regular constraints. In Geoff Sutcliffe and Andrei Voronkov, editors, *LPAR*, volume 3835 of *Lecture Notes in Computer Science*, pages 215–229. Springer, 2005.

14 John Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 2nd edition, 1987.

15 Mircea Marin and Temur Kutsia. Foundations of the rule-based system ρLog. *Journal of Applied Non-Classical Logics*, 16(1-2):151–168, 2006.

16 Mark van den Brand, Arie van Deursen, Jan Heering, Hayco de Jong, Merijn de Jonge, Tobias Kuipers, Paul Klint, Leon Moonen, Pieter A. Olivier, Jeroen Scheerder, Jurgen J. Vinju, Eelco Visser, and Joost Visser. The Asf+Sdf meta-environment: a component-based language development environment. *Electr. Notes Theor. Comput. Sci.*, 44(2):3–8, 2001.

17 Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. In Aart Middeldorp, editor, *Rewriting Techniques and Applications, 12th International Conference, RTA 2001*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–362. Springer, 2001.

18 Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012.

Grounded Fixpoints and Active Integrity Constraints[*]

Luís Cruz-Filipe

Dept. Mathematics and Computer Science, University of Southern Denmark, Odense, Denmark
lcfilipe@gmail.com

Abstract

The formalism of active integrity constraints was introduced as a way to specify particular classes of integrity constraints over relational databases together with preferences on how to repair existing inconsistencies. The rule-based syntax of such integrity constraints also provides algorithms for finding such repairs that achieve the best asymptotic complexity.

However, the different semantics that have been proposed for these integrity constraints all exhibit some counter-intuitive examples. In this work, we look at active integrity constraints using ideas from algebraic fixpoint theory. We show how database repairs can be modeled as fixpoints of particular operators on databases, and study how the notion of grounded fixpoint induces a corresponding notion of grounded database repair that captures several natural intuitions, and in particular avoids the problems of previous alternative semantics.

In order to study grounded repairs in their full generality, we need to generalize the notion of grounded fixpoint to non-deterministic operators. We propose such a definition and illustrate its plausibility in the database context.

1998 ACM Subject Classification H.2.7 Database Administration, D.1.6 Logic Programming

Keywords and phrases grounded fixpoints, active integrity constraints

Digital Object Identifier 10.4230/OASIcs.ICLP.2016.11

1 Introduction

The classical definition of model of a logic theory requires models to be deductively closed. An alternative phrasing of this fact is saying that models are fixpoints of some entailment operator, and indeed the semantics of many modern logic frameworks can be described as (minimal) fixpoints of particular operators – in particular, those of logic programs, default logics, or knowledge representation formalisms based on argumentation.

Several of these formalisms focus on models that can be constructed "from the ground up" (such as the minimal model of a positive logic program). Grounded fixpoints of lattice operators, studied in [5], were proposed with the intent of capturing this notion in the formal setting of algebraic fixpoint theory, and were shown to abstract from many useful types of fixpoints in logic programming and knowledge representation.

In this work, we are interested in applying this intuition within the context of databases with integrity constraints – formulas that describe logical relations between data in a database, which should hold at all times. We focus on the particular formalism of active integrity constraints (AICs), which not only specify an integrity constraint, but also give indications on how inconsistent databases can be repaired. Although not all integrity constraints can be

[*] Supported by the Danish Council for Independent Research, Natural Sciences, grant DFF-1323-00247.

expressed in this formalism, AICs capture the class of integrity constraints that can be written in denial clausal form, which includes many examples that are important in practice [14]. Using AICs, one can distinguish between different types of repairs that embody typical desirable properties – minimality of change [12, 24], the common sense law of inertia [20], or non-circular justification for repair actions [7]. These intuitions capture many aspects of the idea of "building a model from the ground up", present in grounded fixpoints. However, the semantics of both founded [6] and justified repairs [7] exhibit counter-intuitive behaviors, which led to the proposal of well-founded repairs [9]. These in turn are not modular with respect to stratification of repairs [8], rendering their computation problematic.

In this paper we show that repairs of inconsistent databases can be characterized as fixpoints of a particular operator, with minimality of change corresponding to being a minimal fixpoint, and that both founded and well-founded repairs can be described as fixpoints with additional properties. We then study grounded fixpoints of this operator, and show that they include all founded and well-founded repair, but not all justified repairs. In particular, grounded fixpoints avoid the circularity issues found in founded repairs, while including some intuitive non-justified repairs.

To study at AICs in their full generality, we need to consider non-deterministic operators. While there is currently no notion of grounded fixpoint of a non-deterministic operator, we show that we can define this concept in the context of AICs in a manner that naturally generalizes the deterministic definition. We then show how this in turn yields a plausible definition of grounded fixpoints of non-deterministic operators within the general framework of algebraic fixpoint theory.

Related work. Database consistency has long since been recognized as an important problem in knowledge management. Especially in relational databases, integrity constraints have been used for decades to formalize relationships between data in the database that are dictated by its semantics [2, 4].

Whenever an integrity constraint is violated, it is necessary to change the database in order to regain consistency. This process of bringing the database back to consistency is known as *database repair*, and the problem of database repair is to determine whether such a transformation is possible. Typically, there are several possible ways of repairing an inconsistent database, and several criteria have been proposed to evaluate them. *Minimality of change* [12, 24] demands that the database be changed as little as possible, while the *common-sense law of inertia* [20] states that every change should have an underlying reason. While these criteria narrow down the possible database repairs, human interaction is ultimately required to choose the "best" possible repair [22].

Database management systems typically implement integrity constraints as a variant of event-condition-action rules (ECAs, [22, 23]), for which rule processing algorithms have been proposed and a procedural semantics has been defined. However, their lack of declarative semantics makes it difficult to understand the behavior of multiple ECAs acting together and to evaluate rule-processing algorithms in a principled way. Active integrity constraints (AICs) [14] are inspired by the same principle, encoding an integrity constraint together with preferred update actions to repair it. The update actions are limited to addition and removal of tuples from the database, as this suffices to implement the three main operations identified in the seminal work of Abiteboul [1]. AICs follow the tradition of expressing database dependencies through logic programming, which is common namely in the setting of deductive databases [17, 19, 20].

The declarative semantics for AICs [6, 7] is based on the concept of founded and justified repairs, motivated by different interpretations of the common-sense law of inertia, and the operational semantics for AICs [9] allows their direct computation by means of intuitive tree algorithms, which have been implemented over SQL databases [10]. However, neither founded nor justified repairs are completely satisfactory, as counter-intuitive examples have been produced exhibiting limitations of both types of repairs. Similar flaws have been exposed for the alternative notion of well-founded repairs [9].

Deciding whether a database can be repaired is typically a computationally hard problem. In the framework of AICs, the complexity of this problem depends on the type of repairs allowed, varying between NP-complete and Σ_p^2. Because of this intrinsic complexity, techniques to split a problem in several smaller ones are important in practice. A first step in this direction was taken in [18], but that work explicitly forbids cyclic dependencies. A more general study, in the context of AICs, was undertaken in [8], which studies conditions under which a set of constraints can be split into smaller sets, whose repairs may then be computed separately.

In the more general setting of knowledge bases with more powerful reasoning abilities, the problem of computing repairs is much more involved than in databases, as it amounts to solving an abduction problem [15]. In those frameworks, AICs can help greatly with finding repairs, and we are currently investigating how this formalism can be applied outside the database world [11].

The operational semantics for AICs proposed in [9] was inspired by Antoniou's survey on semantics of default logic [3]. The realization that Reiter's original semantics for default logic [21] defines extensions by means of what is essentially a fixpoint definition naturally leads to the question of whether we can characterize repairs of inconsistent databases in a similar way. Indeed, some connections between the semantics for AICs and logic programming have been discussed in [7], and fixpoints play a crucial role in defining several semantics for logic programs [13]. These include the standard construction of minimal models of positive logic programs and the notion of answer sets (via the Gelfond–Lifschitz transform). Fixpoints also abound in other domains of logic; many of these occurrences of fixpoints are summarized in [5], and showing that several of them can be seen as instances of the same abstract notion constitutes one of those authors' motivation for studying grounded fixpoints.

2 Preliminaries

In this section we review the concepts and results that are directly relevant for the remainder of the presentation: grounded fixpoints of lattice operators [5], the formalism of active integrity constraints [14], founded [6], justified [7] and well-founded [9] (weak) repairs, and parallelization results for these.

Grounded fixpoints. A partial order is a binary relation that is reflexive, antisymmetric and transitive. A set L equipped with a partial order $\leq$ is called a *poset* (for partially ordered set), and it is customary to write $x < y$ if $x, y \in L$ are such that $x \leq y$ and $x \neq y$. Given $S \subseteq L$, an *upper bound* of S is an element x such that $s \leq x$ for all $s \in S$, and x is a *least upper bound* (lub) or *join* of S if $x \leq y$ for all upper bounds y of S, and we write $x = \bigvee S$. The notion of *(greatest) lower bound*, or *meet*, is dually defined, and written $\bigwedge S$. Meets and joins, if they exist, are necessarily unique. For binary sets, it is standard practice to write $x \wedge y$ and $x \vee y$ instead of $\bigwedge\{x, y\}$ and $\bigvee\{x, y\}$.

A *complete lattice* is a poset in which every set has a join and a meet. In particular, complete lattices have a greatest element $\top$ and a smallest element $\bot$. The *powerset lattice* of a set S is $\langle \wp(S), \subseteq \rangle$, whose elements are the subsets of S ordered by inclusion. The powerset lattice is a complete lattice with joins given by union and meets given by intersection. Its greatest element is S, and its smallest element is $\emptyset$.

A lattice operator is a function $\mathcal{O} : L \to L$. A *fixpoint* of $\mathcal{O}$ is an element $x \in L$ for which $\mathcal{O}(x) = x$. If $x \leq y$ for all fixpoints y of $\mathcal{O}$, then x is said to be the *least* (or *minimal fixpoint* of $\mathcal{O}$. Lattice operators do not need to have fixpoints, but *monotone* operators (i.e. those for which $x \leq y$ implies $\mathcal{O}(x) \leq \mathcal{O}(y)$) always have a minimal fixpoint.

We will be interested in two particular kinds of fixpoints, introduced in [5]. We summarize the definitions and Propositions 3.3, 3.5 and 3.8 from that work.

▶ **Definition 1.** Let $\mathcal{O}$ be an operator over a lattice $\langle L, \leq \rangle$. An element $x \in L$ is:

- *grounded* for $\mathcal{O}$ if $\mathcal{O}(x \wedge v) \leq v$ implies $x \leq v$, for all $v \in L$;
- *strictly grounded* for $\mathcal{O}$ if there is no $y \in L$ such that $y < x$ and $(\mathcal{O}(y) \wedge x) \leq y$.

▶ **Lemma 2.** *Let $\mathcal{O}$ be an operator over a lattice $\langle L, \leq \rangle$.*

1. *All strictly grounded fixpoints of $\mathcal{O}$ are grounded.*
2. *If $\langle L, \leq \rangle$ is a powerset lattice, then all grounded fixpoints of $\mathcal{O}$ are strictly grounded.*
3. *All grounded fixpoints of $\mathcal{O}$ are minimal.*

We will be working mostly in a powerset lattice, so throughout this paper we will treat the notions of strictly grounded and grounded as equivalent.

Active integrity constraints (AICs). The formalism of AICs was originally introduced in [14], but later simplified in view of the results in [6]. We follow the latter's definition, with a more friendly and simplified notation.

We assume a fixed set At of *atoms* (typically, closed atomic formulas of a first-order theory); subsets of At are *databases*. A *literal* is either an atom (a) or its negation ($\neg a$), and a database DB satisfies a literal ℓ, denoted $DB \models \ell$, if: ℓ is an atom $a \in DB$, or ℓ is $\neg a$ and $a \notin DB$. An *update action* α has the form $+a$ or $-a$, where $a \in At$; $+a$ and $-a$ are *dual* actions, and we represent the dual of α by α^D. Update actions are intended to change the database: $+a$ adds a to the database (formally: it transforms DB into $DB \cup \{a\}$), while $-a$ removes it (formally: it transforms DB into $DB \setminus \{a\}$). A set of update actions $\mathcal{U}$ is *consistent* if it does not contain an action and its dual. A consistent set of update actions $\mathcal{U}$ acts on a database DB by updating DB by means of all its actions simultaneously; we denote the result of this operation by $\mathcal{U}(DB)$.

Literals and update actions are related by natural mappings lit and ua, where $\mathsf{lit}(+a) = a$, $\mathsf{lit}(-a) = \neg a$, $\mathsf{ua}(a) = +a$ and $\mathsf{ua}(\neg a) = -a$. An AIC is a rule r of the form

$$\ell_1, \ldots, \ell_n \supset \alpha_1 \mid \ldots \mid \alpha_k \tag{1}$$

where $n, k \geq 1$ and $\{\mathsf{lit}(\alpha_1^D), \ldots, \mathsf{lit}(\alpha_k^D)\} \subseteq \{\ell_1, \ldots, \ell_n\}$. The intuition behind this notation is as follows: the *body* of the rule, $\mathsf{body}(r) = \ell_1, \ldots, \ell_n$ describes an inconsistent state of the database. If $DB \models \ell_1 \wedge \ldots \wedge \ell_n$, which we write as $DB \models \mathsf{body}(r)$, then r is *applicable*, and we should fix this inconsistency by applying one of the actions in the *head* of r, $\mathsf{head}(r) = \alpha_1 \mid \ldots \mid \alpha_k$. The syntactic restriction was motivated by the observation [6] that actions that do not satisfy this condition may be removed from $\mathsf{head}(r)$ without changing the semantics of AICs, which we now describe.

Generic integrity constraints were previously written as first-order clauses with empty head (see [14]), and we can see AICs as a generalization of this concept: an integrity constraint $\ell_1 \wedge \ldots \wedge \ell_n \to \bot$ expresses no preferences regarding repairs, and thus corresponds to the (closed instances of the) AIC $\ell_1, \ldots, \ell_n \supset \mathsf{ua}(\ell_1)^D \mid \ldots \mid \mathsf{ua}(\ell_n)^D$. Our presentation essentially treats At as a set of propositional symbols, following [7]; for the purposes of this paper, the distinction is immaterial (we can identify an AIC including variables with the set of its closed instances), but our choice makes the presentation much simpler.

A set of update actions $\mathcal{U}$ is a *weak repair* for DB and a set η of AICs (shortly, for $\langle DB, \eta\rangle$) if: (i) every action in $\mathcal{U}$ changes DB and (ii) $\mathcal{U}(DB) \not\models \mathsf{body}(r)$ for all $r \in \eta$. Furthermore, if $\mathcal{U}$ is minimal wrt set inclusion, then $\mathcal{U}$ is said to be a *repair*; repairs are also minimal among all sets satisfying only condition (ii), embodying the principle of *minimality of change* [24] explained earlier.

▶ **Definition 3.** A set of update actions $\mathcal{U}$ is *founded* wrt $\langle DB, \eta\rangle$ if, for every $\alpha \in \mathcal{U}$, there exists $r \in \mathcal{U}$ such that $\alpha \in \mathsf{head}(r)$ and $\mathcal{U}(DB) \models \mathsf{body}(r) \setminus \{\mathsf{lit}(\alpha^D)\}$. A *founded (weak) repair* is a (weak) repair that is founded.

The intuition is as follows: in a founded weak repair, every action has *support* in the form of a rule that "requires" its inclusion in $\mathcal{U}$. We will use the (equivalent) characterization of founded sets: $\mathcal{U}$ is founded iff, for every $\alpha \in \mathcal{U}$, there is a rule r such that $\alpha \in \mathsf{head}(r)$ and $(\mathcal{U} \setminus \{\alpha\})(DB) \models \mathsf{body}(r)$.

However, Caroprese *et al.* [7] discovered that there can be founded repairs exhibiting *circularity of support* (see Example 17 below), and they proposed the stricter notion of justified repair.

▶ **Definition 4.** Let $\mathcal{U}$ be a set of update actions and DB be a database.

- The *no-effect actions* wrt DB and $\mathcal{U}$ are the actions that do not affect either DB or $\mathcal{U}(DB)$: $\mathsf{neff}_{DB}(\mathcal{U}) = \{+a \mid a \in DB \cap \mathcal{U}(DB)\} \cup \{-a \mid a \notin DB \cup \mathcal{U}(DB)\}$.
- The set of *non-updateable literals* of an AIC r is $\mathsf{body}(r) \setminus \mathsf{lit}\left(\mathsf{head}(r)^D\right)$, where the functions lit and $\cdot^D$ are extended to sets in the natural way.
- $\mathcal{U}$ is *closed under* η if, for each $r \in \eta$, $\mathsf{ua}(\mathsf{nup}(r)) \subseteq \mathcal{U}$ implies $\mathsf{head}(r) \cap \mathcal{U} \neq \emptyset$.
- $\mathcal{U}$ is a *justified action set* if it is the least superset of $\mathcal{U} \cup \mathsf{neff}_{DB}(\mathcal{U})$ closed under η.
- $\mathcal{U}$ is a justified (weak) repair if $\mathcal{U}$ is a (weak) repair and $\mathcal{U} \cup \mathsf{neff}_{DB}(\mathcal{U})$ is a justified action set.

The notion of justified weak repair, however, is extremely complicated and unwieldy in practice, due to its quantification over sets of size comparable to that of DB. Furthermore, it excludes some repairs that seem quite reasonable and for which it can be argued that the circularity of support they exhibit is much weaker (see Example 20). This motivated proposing yet a third kind of weak repair: well-founded repairs, that are defined by means of an operational semantics inspired by the syntax of AICs [9].

▶ **Definition 5.** Let DB be a database and η be a set of AICs. The *well-founded repair tree* for $\langle DB, \eta\rangle$ is built as follows: its nodes are labeled by sets of update actions, with root $\emptyset$; the descendants of a node with consistent label $\mathcal{U}$ are all sets of the form $\mathcal{U} \cup \{\alpha\}$ such that there exists a rule $r \in \eta$ with $\alpha \in \mathsf{head}(r)$ and $\mathcal{U}(DB) \models \mathsf{body}(r)$. The consistent leaves of this tree are *well-founded weak repairs* for $\langle DB, \eta\rangle$.

Equivalently, a weak repair $\mathcal{U}$ for $\langle DB, \eta\rangle$ is well-founded iff there exists a sequence of actions $\alpha_1, \dots, \alpha_n$ such that $\mathcal{U} = \{\alpha_1, \dots, \alpha_n\}$ and, for each $1 \leq i \leq n$, there exists a rule r_i such that $\{\alpha_1, \dots, \alpha_{i-1}\}(DB) \models \mathsf{body}(r_i)$ and $\alpha_i \in \mathsf{head}(r_i)$.

The availability of multiple actions in the heads of AICs makes the construction of repairs non-deterministic, and a normalization procedure was therefore proposed in [7]. An AIC r is *normal* if $|\mathsf{head}(r)| = 1$. If r is an AIC of the form in (1), then $\mathcal{N}(r) = \{\ell_1, \dots, \ell_n \supset \alpha_i \mid 1 \leq i \leq k\}$, and $\mathcal{N}(\eta) = \bigcup\{\mathcal{N}(r) \mid r \in \eta\}$. It is straightforward to check that $\mathcal{U}$ is a weak repair (respectively, repair, founded (weak) repair or well-founded (weak) repair) for $\langle DB, \eta \rangle$ iff $\mathcal{U}$ is a weak repair (resp. repair, founded (weak) repair or well-founded (weak) repair) for $\langle DB, \mathcal{N}(\eta) \rangle$; however, this equivalence does not hold for justified (weak) repairs, as shown in [7].

Parallelization. Determining whether a database satisfies a set of AICs is linear on both the size of the database and the number of constraints. However, determining whether an inconsistent database can be repaired is a much harder problem – NP-complete, if any repair is allowed, but Σ_2^P-complete, when repairs have to be founded or justified. (Here, Σ_2^P is the class of problems that can be solved in non-deterministic polynomial time, given an oracle that can solve any NP-complete problem.) This complexity only depends on the size of the set of AICs [7]. In the normalized case, several of these problems become NP-complete; even so, separating a set of AICs into smaller sets that can be processed independently has a significant practical impact [8].

There are two important splitting techniques: *parallelization*, which splits a set of AICs into smaller sets for which the database can be repaired independently (in principle, in parallel); and *stratification*, which splits a set of AICs into smaller sets, partially ordered, such that repairs can be computed incrementally using a topological sort of the order. We shortly summarize the definitions and results from [8].

▶ **Definition 6.** Let η_1 and η_2 be two sets of AICs over a common set of atoms At.

- η_1 and η_2 are *strongly independent*, $\eta_1 \perp\!\!\!\perp \eta_2$, if, for each pair of rules $r_1 \in \eta_1$ and $r_2 \in \eta_2$, $\mathsf{body}(r_1)$ and $\mathsf{body}(r_2)$ contain no common or dual literals.
- η_1 and η_2 are *independent*, $\eta_1 \perp \eta_2$, if, for each pair of rules $r_1 \in \eta_1$ and $r_2 \in \eta_2$, $\mathsf{lit}(\mathsf{head}(r_i))$ and $\mathsf{body}(r_{3-i})$ contain no common or dual literals, for $i = 1, 2$.
- η_1 *precedes* η_2, $\eta_1 \prec \eta_2$, if, for each pair of rules $r_1 \in \eta_1$ and $r_2 \in \eta_2$, $\mathsf{lit}(\mathsf{head}(r_2))$ and $\mathsf{body}(r_1)$ contain no common or dual literals, but not conversely.

From the syntactic restrictions on AICs, it follows that $\eta_1 \perp\!\!\!\perp \eta_2$ implies $\eta_1 \perp \eta_2$. Given two sets of AICs η_1 and η_2 a set of update actions $\mathcal{U}$, let $\mathcal{U}_i = \mathcal{U} \cap \{\alpha \mid \alpha \in \mathsf{head}(r), r \in \eta_i\}$.

▶ **Lemma 7.** *Let η_1 and η_2 be sets of AICs, $\eta = \eta_1 \cup \eta_2$, and $\mathcal{U}$ be a set of update actions.*

1. *If $\eta_1 \perp\!\!\!\perp \eta_2$, then $\mathcal{U}$ is a repair for $\langle DB, \eta \rangle$ iff $\mathcal{U} = \mathcal{U}_1 \cup \mathcal{U}_2$ and $\mathcal{U}_i$ is a repair for $\langle DB, \eta_i \rangle$, for $i = 1, 2$.*
2. *If $\eta_1 \perp \eta_2$, then $\mathcal{U}$ is a founded/well-founded/justified repair for $\langle DB, \eta \rangle$ iff $\mathcal{U} = \mathcal{U}_1 \cup \mathcal{U}_2$ and $\mathcal{U}_i$ is a founded/well-founded/justified repair for $\langle DB, \eta_i \rangle$, for $i = 1, 2$.*
3. *If $\eta_1 \prec \eta_2$, then $\mathcal{U}$ is a founded/justified repair for $\langle DB, \eta \rangle$ iff $\mathcal{U} = \mathcal{U}_1 \cup \mathcal{U}_2$, $\mathcal{U}_1$ is a founded/justified repair for $\langle DB, \eta_1 \rangle$ and $\mathcal{U}_2$ is a founded/justified repair for $\langle \mathcal{U}_1(DB), \eta_2 \rangle$.*

3 Repairs as Fixpoints

In this section we show how a set of AICs induces an operator on a suitably defined lattice. This operator is in general non-deterministic; in order to reuse the results from algebraic fixpoint theory, we restrict our attention to the case of normalized AICs, and delay the discussion of the general case to a later section.

The operator $\mathcal{T}$. Throughout this paragraph, we assume DB to be a fixed database over a set of atoms At and η to be a set of AICs over At.

The intuitive reading of an AIC r naturally suggests an operation on sets of update actions $\mathcal{U}$, defined as "if $\mathcal{U}(DB) \models \mathsf{body}(r)$ holds, then add $\mathsf{head}(r)$ to $\mathcal{U}$". However, this definition quickly leads to inconsistent sets of update actions, which we want to avoid. We therefore propose a slight variant of this intuition.

▶ **Definition 8.** Let $\mathcal{U}$ and $\mathcal{V}$ be consistent sets of update actions over At. The set $\mathcal{U} \uplus \mathcal{V}$ is defined as $(\mathcal{U} \cup \{\alpha \in \mathcal{V} \mid \alpha^D \notin \mathcal{U}\}) \setminus \{\alpha \in \mathcal{U} \mid \alpha^D \in \mathcal{V}\}$.

This operation models sequential composition of repairs in the following sense: if every action in $\mathcal{U}$ changes DB and every action in $\mathcal{V}$ changes $\mathcal{U}(DB)$, then $(\mathcal{U} \uplus \mathcal{V})(DB) = \mathcal{V}(\mathcal{U}(DB))$. Furthermore, if $\mathcal{U}$ and $\mathcal{V}$ are both consistent, then so is $\mathcal{U} \uplus \mathcal{V}$.

We can identify subsets of At with sets of update actions by matching each atom a with the corresponding action that changes the database (i.e. $-a$ if $a \in DB$ and $+a$ otherwize). We will abuse notation and use this bijection implicitly, so that we can reason over the powerset lattice $\langle \wp(At), \subseteq \rangle$ as having sets of update actions as elements.

▶ **Definition 9.** The operator $\mathcal{T}_\eta^{DB} : \wp(At) \to \wp(\wp(At))$ is defined as follows: $\mathcal{U} \uplus \mathcal{V} \in \mathcal{T}_\eta^{DB}(\mathcal{U})$ iff $\mathcal{V}$ can be constructed by picking exactly one action from the head of each rule r such that $\mathcal{U}(DB) \models \mathsf{body}(r)$.

Each set $\mathcal{V}$ may contain less update actions than there are rules r for which $\mathcal{U}(DB) \models \mathsf{body}(r)$, as the same action may be chosen from the heads of different rules; and there may be rules r for which $|\mathsf{head}(r) \cap \mathcal{V}| > 1$. This is illustrated in the following simple example.

▶ **Example 10.** Let $DB = \{a, b\}$ and $\eta = \{a, b, \neg c \supset -a \mid -b; \quad a, b, \neg d \supset -a \mid -b\}$. Then $\mathcal{T}_\eta^{DB}(\emptyset) = \{\{-a\}, \{-b\}, \{-a, -b\}\}$: the bodies of both rules are satisfied in DB, and we can choose $-a$ from the heads of both, $-b$ from the heads of both, or $-a$ from one and $-b$ from the other.

The syntactic restrictions on AICs guarantee that all sets $\mathcal{V}$ in the above definition are consistent: if $+a, -a^D \in \mathcal{V}$, then there are rules r_1 and r_2 such that $\neg a \in \mathsf{body}(r_1)$ and $a \in \mathsf{body}(r_2)$ with $\mathcal{U}(DB) \models \mathsf{body}(r_i)$ for $i = 1, 2$, which is impossible. In the interest of legibility, we will write $\mathcal{T}$ instead of $\mathcal{T}_\eta^{DB}$ whenever DB and η are clear from the context.

The normalized case. In the case that η contains only normalized AICs, the set $\mathcal{T}(\mathcal{U})$ is a singleton, and we can see $\mathcal{T}$ as a lattice operator over $\langle \wp(At), \subseteq \rangle$. We will assume this to be the case throughout the remainder of this section, and by abuse of notation use $\mathcal{T}$ also in this situation. In the normalized case, we thus have

$$\mathcal{T}(\mathcal{U}) = \mathcal{U} \uplus \{\mathsf{head}(r) \mid \mathcal{U}(DB) \models \mathsf{body}(r)\}.$$

Since we can always transform η in a set of normalized AICs by the transformation $\mathcal{N}$ defined above, in most cases it actually suffices to consider this simpler scenario, which warrants its study. The exception is the case of justified repairs for non-normalized AICs, which we defer to a later section. All our results also apply to general integrity constraints by seeing them as AICs with maximal heads and applying $\mathcal{N}$ to the result.

The operator $\mathcal{T}$ characterizes the notions of weak repair, repair, founded and well-founded sets of update actions.

▶ **Lemma 11.** *U is a weak repair for $\langle DB, \eta \rangle$ iff $\mathcal{U}$ is a fixpoint of $\mathcal{T}$.*

▶ **Lemma 12.** *U is a repair for $\langle DB, \eta \rangle$ iff $\mathcal{U}$ is a minimal fixpoint of $\mathcal{T}$.*

▶ **Lemma 13.** *A consistent set of update actions $\mathcal{U}$ is founded wrt $\langle DB, \eta \rangle$ iff, for all $\alpha \in \mathcal{U}$, it is the case that $\alpha \in \mathcal{T}(\mathcal{U} \setminus \{\alpha\})$.*

▶ **Lemma 14.** *A weak repair $\mathcal{U}$ for $\langle DB, \eta\rangle$ is well-founded iff there is an ordering $\alpha_1, \ldots, \alpha_n$ of the elements of $\mathcal{U}$ such that $\alpha_i \in \mathcal{T}(\{\alpha_1, \ldots, \alpha_{i-1}\})$ for each $i = 1, \ldots, n$.*

The correspondence between justified repairs and answer sets for particular logic programs [7] shows that justified repairs can also be characterized in a related manner. However, since answer sets of a logic program are models of its Gelfond–Lifschitz transform, the corresponding characterization in terms would be as fixpoints of the corresponding operator for a similarly derived set of AICs, rather than of $\mathcal{T}$. This characteristic of justified repairs also explains the rather unexpected behavior we will see later, in § 5.

Grounded fixpoints of $\mathcal{T}$. Founded, well-founded and justified repairs were all introduced with the purpose of characterizing a class of repairs whose actions are supported (there is a reason for having them in the set), and that support is not circular; in particular, these repairs should be constructible "from the ground up", which was the motivation for defining well-founded repairs. However, all notions exhibit unsatisfactory examples: there exist founded repairs with circular support [7] and repairs with no circular support that are not justified [9]; well-founded repairs, on the other hand, are not stratifiable [8], which impacts their computation in practice.

Following the intuition in [5] that grounded fixpoints capture the idea of building fixpoints "from the ground up", we propose the following notion of $\mathcal{T}$.

▶ **Definition 15.** A repair $\mathcal{U}$ for $\langle DB, \eta\rangle$ is *grounded* if $\mathcal{U}$ is a grounded fixpoint of $\mathcal{T}$.

Since we are working within a powerset lattice, the notions of grounded and strictly grounded fixpoints coincide. As it turns out, the latter notion is most convenient for the proofs of our results. We thus characterize grounded repairs as repairs $\mathcal{U}$ such that: if $\mathcal{V} \subsetneq \mathcal{U}$, then $\mathcal{T}(\mathcal{V}) \cap \mathcal{U} \not\subseteq \mathcal{V}$. Equivalently: if $\mathcal{V} \subsetneq \mathcal{U}$, then $\mathcal{T}(\mathcal{V}) \cap (\mathcal{U} \setminus \mathcal{V}) \neq \emptyset$.

Since all grounded fixpoints are minimal, it makes no sense to define grounded weak repairs. The notion of grounded fixpoint therefore intrinsically embodies the principle of minimality of change, unlike other kinds of weak repairs previously defined. Furthermore, grounded repairs also embody the notion of "support" previously defined.

▶ **Lemma 16.** *Every grounded repair for $\langle DB, \eta\rangle$ is both founded and well-founded.*

However, the notion of grounded repair is strictly stronger than both of these: the first example, from [9], also shows that some forms of circular justifications are avoided by grounded repairs.

▶ **Example 17.** Let $DB = \{a, b\}$ and $\eta = \{a, \neg b \supset -a;\quad a, \neg c \supset +c;\quad \neg a, b \supset -b;\quad b, \neg c \supset +c\}$. Then $\mathcal{U} = \{-a, -b\}$ is a founded repair that is not grounded: $\mathcal{V} = \emptyset$ satisfies $\mathcal{T}(\mathcal{V}) \cap \mathcal{U} = \{+c\} \cap \mathcal{U} = \emptyset \subseteq \mathcal{V}$. The more natural repair $\mathcal{U}' = \{+c\}$ is both founded and grounded.

▶ **Example 18.** Let $DB = \emptyset$ and $\eta = \{a, \neg b, \neg c \supset +c;\quad \neg a, \neg b \supset +b;\quad \neg a \supset +a\}$. There are two well-founded repairs for $\langle DB, \eta\rangle$: $\mathcal{U}_1 = \{+a, +c\}$ (obtained by applying the last rule and then the first) and $\mathcal{U}_2 = \{+b, +a\}$ (obtained by applying the second rule and then the last). However, $\mathcal{U}_2$ is not founded ($+b$ is not founded), so it cannot be grounded: indeed, $\mathcal{V} = \{+a\}$ is a strict subset of $\mathcal{U}_2$, and $\mathcal{T}(\mathcal{V}) \cap \mathcal{U} = \{+a, +b\} \cap \mathcal{U} = \emptyset \subseteq \mathcal{V}$.

Also in this last example the grounded repair ($\mathcal{U}_1$) is somewhat more natural.

We now investigate the relation to justified repairs, and find that all justified repairs are grounded, but not conversely – confirming our earlier claim that the notion of justified repair is too strong.

▶ **Lemma 19.** *Every justified repair for $\langle DB, \eta \rangle$ is grounded.*

This result is not very surprising: justified weak repairs are answer sets of a particular logic program (Theorem 6 in [7]), and in turn answer sets of logic programs are grounded fixpoints of the consequence operator (see remark at the top of § 5 in [5]). However, the translation defined in [7] is from logic programs to databases with AICs (rather than the other way around), so Lemma 19 is *not* a direct consequence of those results.

The notion of justified repair is also stricter than that of grounded repair, as the following example from [7] shows.

▶ **Example 20.** Let $DB = \{a, b\}$ and $\eta = \{a, b \supset -a; \quad a, \neg b \supset -a; \quad \neg a, b \supset -b\}$. Then $\mathcal{U} = \{-a, -b\}$ is not justified (see [7]), but it is grounded: if $-a \in \mathcal{V} \subsetneq \mathcal{U}$, then $\mathcal{T}(\mathcal{V}) \cap \mathcal{U}$ contains $-b \in \mathcal{U} \setminus \mathcal{V}$, else $\mathcal{T}(\mathcal{V}) \cap \mathcal{U}$ contains $-a \in \mathcal{U} \setminus \mathcal{V}$.

This example was used in [9] to point out that justified repairs sometimes eliminate "natural" repairs; in this case, the first rule clearly motivates the action $-a$, and the last rule then requires $-b$. This is in contrast to Example 17, where there was no clear reason to include either $-a$ or $-b$ in a repair. So grounded repairs avoid this type of unreasonable circularities, without being as restrictive as justified repairs.

We thus have that grounded repairs are always founded and well-founded; the next example shows that they do not correspond to the intersection of those classes.

▶ **Example 21.** Assume that $DB = \emptyset$ and η contains the following integrity constraints.

$$\neg a, \neg b \supset +a \qquad a, \neg b \supset +b \qquad \neg a, b \supset -b \qquad a, b, \neg c \supset +c \qquad a, \neg b, c \supset +b \qquad \neg a, b, c \supset +a$$

Then $\mathcal{U} = \{+a, +b, +c\}$ is a repair for $\langle DB, \eta \rangle$: the first three constraints require $+a$ and $+b$ to be included in any repair for $\langle DB, \eta \rangle$, and the last three state that no 2-element subset of $\mathcal{U}$ is a repair. Furthermore, $\mathcal{U}$ is founded (the three last rules ensure that) and well-founded (starting with $\mathcal{U}$, the rules force us to add $+a$, $+b$ and $+c$, in that order).

However, $\mathcal{U}$ is not strictly grounded for $\mathcal{T}$: if $\mathcal{V} = \{+b\}$, then $\mathcal{V} \subsetneq \mathcal{U}$, but $\mathcal{T}(\mathcal{V}) \cap \mathcal{U} = \emptyset \cap \mathcal{U} = \emptyset \subseteq \mathcal{V}$.

In this situation, $\mathcal{U}$ actually seems reasonable; however, observe that the support for its actions *is* circular: it is the three rules in the second row that make $\mathcal{U}$ founded, and none of them is applicable to DB. Also, note that $\mathcal{V}(DB)$ is a database for which the given set η behaves very awkwardly: the only applicable AIC tells us to remove b, but the only possible repair is actually $\{+a, +c\}$.

We do not feel that this example weakens the case for studying ground repairs, though: the consensual approach to different notions of repair is that they express *preferences*. In this case, where $\langle DB, \eta \rangle$ admits no grounded repair, it is sensible to allow a repair in a larger class – and a repair that is both founded and well-founded is a good candidate. The discussion in § 8 of [7] already proposes such a "methodology": choose a repair from the most restrictive category (justified, founded, or any). We advocate a similar approach, but dropping justified repairs in favor of grounded repairs, and preferring well-founded to founded repairs.

The relations between the different classes of repairs are summarized in the picture below.

$$\begin{tikzcd}
\mathcal{F} \arrow[dr, dash, "\subsetneq"] \arrow[dd, dash, "\neq"'] & & \\
& \mathcal{G} \arrow[r, dash, "\subsetneq"] & \mathcal{J} \\
\mathcal{WF} \arrow[ur, dash, "\subsetneq"'] & &
\end{tikzcd}$$

We conclude this section with a note on complexity.

▶ **Theorem 22.** *The problem of deciding whether there exist grounded repairs for* $\langle DB, \eta\rangle$ *is* Σ_2^P*-complete.*

This result still holds if we allow a truly first-order syntax for AICs, where the atoms can include variables that are implictly universally quantified.

4 Parallelism

Lemma 7 shows that splitting a set of AICs into smaller ones transforms the problem of deciding whether an inconsistent database can be repaired (and computing founded or justified repairs) into smaller ones, with important practical consequences. The goal of this section is to show that grounded repairs enjoy similar properties. This is even more relevant, as deciding whether grounded repairs exist is presumably[1] more complex than for the other cases, in view of Theorem 22. For parallelization, we will go one step further, and propose a lattice-theoretical concept of splitting an operator into "independent" operators in such a way that strictly grounded fixpoints can be computed in parallel.

We make some notational conventions for the remainder of this section. We will assume as before a fixed database DB and set of AICs η over the same set of atoms At. Furthermore, we will take η_1 and η_2 to be disjoint sets with $\eta = \eta_1 \cup \eta_2$, and write $\mathcal{T}_i$ for $\mathcal{T}_{\eta_i}^{DB}$. Also, we write $\hat{\imath}$ for $3 - i$ (so $\hat{\imath} = 1$ if $i = 2$ and vice-versa).

Independence. We begin with a simple consequence of independence.

▶ **Lemma 23.** *If* $\eta_1 \perp \eta_2$*, then* $\mathcal{T}_1$ *and* $\mathcal{T}_2$ *commute and* $\mathcal{T} = \mathcal{T}_1 \circ \mathcal{T}_2 = \mathcal{T}_2 \circ \mathcal{T}_1$*.*

The converse is not true.

▶ **Example 24.** Let $\eta_1 = \{a, b \supset -b\}$ and $\eta_2 = \{\neg a, \neg b \supset +b\}$. Then $\eta_1 \not\perp \eta_2$, but $\mathcal{T}_1$ and $\mathcal{T}_2$ commute: if $a \in \mathcal{U}(DB)$, then $\mathcal{T}_1(\mathcal{T}_2(\mathcal{U})) = \mathcal{T}_1(\mathcal{U}) = \mathcal{T}_2(\mathcal{T}_1(\mathcal{U}))$; otherwise, $\mathcal{T}_1(\mathcal{T}_2(\mathcal{U})) = \mathcal{T}_2(\mathcal{U}) = \mathcal{T}_2(\mathcal{T}_1(\mathcal{U}))$.

▶ **Lemma 25.** *A set of update actions* $\mathcal{U}$ *is a grounded repair for* $\langle DB, \eta\rangle$ *iff* $\mathcal{U} = \mathcal{U}_1 \cup \mathcal{U}_2$ *and* $\mathcal{U}_1$ *is a grounded repair for* $\langle DB, \eta_1\rangle$ *and* $\mathcal{U}_2$ *is a grounded repair for* $\langle DB, \eta_2\rangle$*.*

These properties are actually not specific to operators induced by AICs, but can be formulated in a more general lattice-theoretic setting.

▶ **Definition 26.** Let $\langle L, \leq\rangle$ be a complete distributive lattice with complements. An operator $\mathcal{O} : L \to L$ is an (u, v)-operator, with $u \leq v \in L$, if, for every $x \in L$,

$$\mathcal{O}(x) = (\mathcal{O}(x \wedge v) \wedge u) \vee (x \wedge \bar{u})\,.$$

Intuitively, an (u, v)-operator only depends on the "v-part" of its argument, and the result only differs from the input in its "u-part". In this context, Proposition 3.5 of [5] applies, so grounded and strictly grounded fixpoints coincide; furthermore, we can extend the definition of independence to this setting and generalize Lemmas 23 and 25.

Observe that, by construction, $\mathcal{T}_\eta$ is a $(\mathcal{U}, \mathcal{V})$-operator with $\mathcal{U} = \{\mathsf{head}(r) \mid r \in \eta\}$ and $\mathcal{V} = \{\mathsf{ua}(l) \mid l \in \mathsf{body}(r), r \in \eta\}$.

[1] I.e., assuming that P $\neq$ NP.

▶ **Definition 27.** Two operators $\mathcal{O}_1, \mathcal{O}_2 : L \to L$ are *independent* if each $\mathcal{O}_i$ is an (u_i, v_i)-operator with $u_i \wedge v_i = \bot$.

▶ **Lemma 28.** *If $\mathcal{O}_1$ and $\mathcal{O}_2$ are independent, then $\mathcal{O}_1$ and $\mathcal{O}_2$ commute. In this case, if $\mathcal{O}$ is their composition, then $x \in L$ is (strictly) grounded for $\mathcal{O}$ iff $x = (x \wedge v_1) \vee (x \wedge v_2)$ and $x \wedge v_i$ is (strictly) grounded for $\mathcal{O}_i$.*

This provides an algebraic counterpart to the parallelization of AICs, albeit requiring that the underlying lattice be distributive and complemented: we say that $\mathcal{O}$ is parallelizable if there exist $\mathcal{O}_1$ and $\mathcal{O}_2$ in the conditions of Lemma 28, with $\mathcal{O} = \mathcal{O}_1 \circ \mathcal{O}_2$. As in the original work [8], it is straightforward to generalize these results to finite sets of independent operators.

Stratification. We now consider the case where η_1 and η_2 are not independent, but rather stratified, and show that part 3 of Lemma 7 also applies to grounded repairs.

▶ **Lemma 29.** *Suppose that $\eta_1 \prec \eta_2$. Then $\mathcal{U}$ is a grounded repair for $\langle DB, \eta\rangle$ iff $\mathcal{U} = \mathcal{U}_1 \cup \mathcal{U}_2$, $\mathcal{U}_1$ is a grounded repair for $\langle DB, \eta_1\rangle$, and $\mathcal{U}_2$ is a grounded repair for $\langle \mathcal{U}_1(DB), \mathcal{U}_2\rangle$.*

Unlike parallelization, there is no clear generalization of these results to a more general setting: the definition of $\mathcal{T}_2$ is dependent of the particular fixpoint for $\mathcal{T}_1$, and to express this dependency we are using the sets η_1 and η_2 in an essential way.

5 General AICs and Non-deterministic Operators

We now return to the original question of defining grounded repairs for databases with arbitrary (not necessarily normal) sets of active integrity constraints. This requires generalizing the definition of (strictly) grounded element to non-deterministic lattice operators, a question that was left open in [5]. We propose possible definitions for these concepts, and show that they exhibit desirable properties in our topic of interest.

Let $\mathcal{O} : L \to L$ be a lattice operator, and define its non-deterministic counterpart $\mathcal{O}^\uparrow : L \to \wp(L)$ by $\mathcal{O}^\uparrow(x) = \{\mathcal{O}(x)\}$. A reasonable requirement is that x should be (strictly) grounded for $\mathcal{O}^\uparrow$ iff x is (strictly) grounded for $\mathcal{O}$. Furthermore, in the case of AICs we can also define a converse transformation: since every set of AICs η can be transformed into a normalized set $\mathcal{N}(\eta)$, we will also require that $\mathcal{U}$ be a grounded repair for $\mathcal{T}_\eta$ iff $\mathcal{U}$ is a grounded repair for $\mathcal{T}_{\mathcal{N}(\eta)}$.

▶ **Definition 30.** Let $\mathcal{O} : L \to \wp(L)$ be a non-deterministic operator over a complete lattice L. An element $x \in L$ is:

- *grounded* for $\mathcal{O}$ if $(\bigvee \mathcal{O}(x \wedge v)) \leq v$ implies $x \leq v$;
- *strictly grounded* for $\mathcal{O}$ if there is no $v < x$ such that $(\bigvee \mathcal{O}(v)) \wedge x \leq v$.

Clearly these definitions satisfy the first criterion stated above: given $\mathcal{O} : L \to L$, $\bigvee(\mathcal{O}^\uparrow(x)) = \mathcal{O}(x)$ for every $x \in L$. The choice of a join instead of a meet is motivated by the second criterion, which we will show is satisfied by this definition. Furthermore, all grounded elements are again strictly grounded, and the two notions coincide over powerset lattices – the proofs in [5] are trivial to adapt to this case.

As before, we assume that the database DB is fixed, and omit it from the superscript in the operators below.

▶ **Lemma 31.** *For every $\mathcal{U}$, $\mathcal{T}_{\mathcal{N}(\eta)}(\mathcal{U}) \subseteq \bigcup \mathcal{T}_\eta(\mathcal{U})$.*

Note that the set $\{\bigcup \mathsf{head}(r) \mid \mathcal{U}(DB) \models \mathsf{body}(r), r \in \eta\}$ is consistent, due to the syntactic restrictions on AICs and the fact that all rules are evaluated in the same context.

▶ **Example 32.** The inclusion in Lemma 31 is, in general, strict: consider $DB = \emptyset, \mathcal{U} = \{+a\}$, and let $\eta = \{a, \neg b \supset -a \mid +b\}$. Then $\mathcal{N}(\eta)$ contains the two AICs $a, \neg b \supset -a$ and $a, \neg b \supset +b$. In this case, $\mathcal{T}_\eta(\mathcal{U}) = \{\emptyset, \{+a, +b\}\}$ and $\mathcal{T}_{\mathcal{N}(\eta)}(\mathcal{U}) = \{+b\}$.

▶ **Lemma 33.** *$\mathcal{U}$ is strictly grounded for $\mathcal{T}_\eta$ iff $\mathcal{U}$ is strictly grounded for $\mathcal{T}_{\mathcal{N}(\eta)}$.*

A fixpoint of a non-deterministic operator $\mathcal{O} : L \to \wp(L)$ is a value $x \in L$ such that $x \in \mathcal{O}(L)$ (see e.g. [16]). From the definition of $\mathcal{T}_\eta$, it is immediate that $\mathcal{U} \in \mathcal{T}_\eta(\mathcal{U})$ iff $\mathcal{T}(\mathcal{U}) = \{\mathcal{U}\}$. Furthermore, Lemmas 11 and 12 still hold in this non-deterministic case, allowing us to derive the following consequence of the previous lemma.

▶ **Corollary 34.** *$\mathcal{U}$ is a grounded repair for $\langle DB, \eta\rangle$ iff $\mathcal{U}$ is a grounded repair for $\langle DB, \mathcal{N}(\eta)\rangle$.*

Since repairs, founded repairs and well-founded repairs for η and for $\mathcal{N}(\eta)$ also coincide, we immediately obtain generalizations of Lemma 16 for the general setting, and the parallelization and independence results from § 4 also apply.

As observed in [7], normalization does not preserve justified repairs. Therefore, Lemma 19 does not guarantee that justified repairs are always grounded in the general case. Indeed, the next example shows that this is *not* true.

▶ **Example 35.** Let $DB = \emptyset$ and take η to be the following set of AICs.

$$\begin{array}{lllllllll}
a, b, \neg c \supset -a \mid -b \mid +c & (1) & a, \neg b \supset -a & (3) & \neg a, b, c \supset +a \mid -b \mid -c & (5) \\
\neg a, b, \neg c \supset +a \mid -b \mid +c & (2) & a, \neg b, c \supset +b \mid -c & (4) & \neg a, \neg b, c \supset +a \mid +b \mid -c & (6) \\
 & & & & \neg a, \neg b, \neg c \supset +a \mid +b \mid +c & (7)
\end{array}$$

Then $\mathcal{U} = \{+a, +b, +c\}$ is the only repair for $\langle DB, \eta\rangle$, and it is justified. Indeed, if $\mathcal{V} \subseteq \mathcal{U}$ is such that $\mathcal{V} \cup \mathsf{neff}_{DB}(\mathcal{U})$ is closed under η, then $\mathcal{V}$ must contain an action in the head of each of rules (1), (2), (5), (6) and (7). Since $\mathcal{V} \subseteq \mathcal{U}$, it follows that $+c \in \mathcal{V}$ (by (1)) and that $+a \in \mathcal{V}$ (by (5)). But then $\mathcal{V}$ contains the actions corresponding to the non-updatable literals in rule (4) (namely, $+a$), and hence also $+b \in \mathcal{V}$, so $\mathcal{V} = \mathcal{U}$.

However, $\mathcal{U}$ is not a strictly grounded fixpoint of $\mathcal{T}$: taking $\mathcal{V} = \{+a\}$, we see that the only rule applicable in $\mathcal{V}(DB)$ is rule (3), and thus $\mathcal{T}(\mathcal{V}) = \{\emptyset\}$, from which trivially $(\bigcup \mathcal{T}(\mathcal{V})) \cap \mathcal{U} \subseteq \mathcal{V}$.

An examination of the conditions under which $\langle DB, \eta\rangle$ may admit a justified repair that is not strictly grounded shows that this example is among the simplest possible. It is important to point out that $\mathcal{U}$ is also not a justified repair for $\langle DB, \mathcal{N}(\eta)\rangle$, either, which seems to suggest that origin of the problem lies in the standard interpretation of AICs with non-singleton heads. We plan to look further into the semantics of repairs for non-normal AICs in future work.

6 Conclusions and Future Work

We have presented a formalization of the theory of active integrity constraints in lattice theory, by showing how a set of AICs η over a database DB induces an operator $\mathcal{T}_\eta^{DB}$ over a suitably defined lattice of database repairs. We characterized the standard notions of (weak) repairs, founded and well-founded repairs in terms of this operator. By studying the grounded fixpoints of $\mathcal{T}_\eta^{DB}$ in the normalized case, we showed that we obtain a notion of repair that is stricter than founded or well-founded repairs, but more general than the problematic notion of justified repairs. Furthermore, by suitably extending the notions of grounded and strictly grounded fixpoint of a lattice operator to the non-deterministic case, we gained a general notion of grounded repair also in the non-normalized case. We also showed that grounded repairs are preserved under normalization, and that they share the parallelization and stratification properties of founded and justified repairs that are important for their practical applications.

Conversely, we were able to state some of the results in the database setting more generally. Thus, not only did we propose an extension of the notion of (strictly) grounded fixpoint to the case of non-deterministic lattice operators, but we also defined what it means for an operator to be parallelizable, and showed that several properties of parallelizable operators are not specific to the database case.

We believe the concept of grounded repair to be the one that better captures our intuitions on what a "good" repair is, in the framework of AICs. We plan to use this notion as the basis for future work on this topic, namely concerning the extension of AICs to more general knowledge representation formalisms, following the proposals in [11].

References

1 Serge Abiteboul. Updates, a new frontier. In Marc Gyssens, Jan Paredaens, and Dirk van Gucht, editors, *ICDT*, volume 326 of *LNCS*, pages 1–18. Springer, 1988.

2 Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases.* Addison Wesley, 1995.

3 Grigoris Antoniou. A tutorial on default logics. *ACM Computing Surveys*, 31(3):337–359, 1999.

4 Catriel Beeri and Moshe Y. Vardi. The implication problem for data dependencies. In *Colloquium on Automata, Languages and Programming*, pages 73–85, London, UK, 1981. Springer.

5 Bart Bogaerts, Joost Vennekens, and Marc Denecker. Grounded fixpoints and their applications in knowledge representation. *Artif. Intell.*, 224:51–71, 2015.

6 Luciano Caroprese, Sergio Greco, Cristina Sirangelo, and Ester Zumpano. Declarative semantics of production rules for integrity maintenance. In Sandro Etalle and Miroslaw Truszczynski, editors, *ICLP*, volume 4079 of *LNCS*, pages 26–40. Springer, 2006.

7 Luciano Caroprese and Miroslaw Truszczynski. Active integrity constraints and revision programming. *Theory Pract. Log. Program.*, 11(6):905–952, November 2011.

8 Luís Cruz-Filipe. Optimizing computation of repairs from active integrity constraints. In Christoph Beierle and Carlo Meghini, editors, *FoIKS*, volume 8367 of *LNCS*, pages 361–380. Springer, 2014.

9 Luís Cruz-Filipe, Patrícia Engrácia, Graça Gaspar, and Isabel Nunes. Computing repairs from active integrity constraints. In Hai Wang and Richard Banach, editors, *TASE*, pages 183–190. IEEE, July 2013.

10 Luís Cruz-Filipe, Michael Franz, Artavazd Hakhverdyan, Marta Ludovico, Isabel Nunes, and Peter Schneider-Kamp. repAIrC: A tool for ensuring data consistency by means of active integrity constraints. In Ana L.N. Fred, Jan L.G. Dietz, David Aveiro, Kecheng Liu, and Joaquim Filipe, editors, *KMIS*, pages 17–26. SciTePress, 2015.

11 Luís Cruz-Filipe, Isabel Nunes, and Peter Schneider-Kamp. Integrity constraints for general-purpose knowledge bases. In Marc Gyssens and Guillermo Ricardo Simari, editors, *FoIKS*, volume 9616 of *LNCS*, pages 235–254. Springer, 2016.

12 Thomas Eiter and Georg Gottlob. On the complexity of propositional knowledge base revision, updates, and counterfactuals. *Artif. Intell.*, 57(2–3):227–270, 1992.

13 Melvin Fitting. Fixpoint semantics for logic programming: a survey. *Theor. Comput. Sci.*, 278(1–2):25–51, 2002.

14 Sergio Flesca, Sergio Greco, and Ester Zumpano. Active integrity constraints. In Eugenio Moggi and David Scott Warren, editors, *PPDP*, pages 98–107. ACM, 2004.

15 Ahmed Guessoum. Abductive knowledge base updates for contextual reasoning. *J. Intell. Inf. Syst.*, 11(1):41–67, 1998.

16 Mohammed A. Khamsi, Vladik Kreinovich, and Driss Misane. A new method of proving the existence of answer sets for disjunctive logic programs. In *Proceedings of the Workshop on Logic Programming with Incomplete Information*, 1993.

17 V. Wiktor Marek and Miroslav Truszczynski. Revision programming, database updates and integrity constraints. In Georg Gottlob and Moshe Y. Vardi, editors, *ICDT*, volume 893 of *LNCS*, pages 368–382. Springer, 1995.

18 Enric Mayol and Ernest Teniente. Addressing efficiency issues during the process of integrity maintenance. In Trevor J.M. Bench-Capon, Giovanni Soda, and A Min Tjoa, editors, *DEXA*, volume 1677 of *LNCS*, pages 270–281. Springer, 1999.

19 Shamim A. Naqvi and Ravi Krishnamurthy. Database updates in logic programming. In Chris Edmondson-Yurkanan and Mihalis Yannakakis, editors, *PODS*, pages 251–262. ACM, 1988.

20 Teodor C. Przymusinski and Hudson Turner. Update by means of inference rules. *J. Log. Program.*, 30(2):125–143, 1997.

21 Raymond Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81–132, 1980.

22 Ernest Teniente and Antoni Olivé. Updating knowledge bases while maintaining their consistency. *VLDB J.*, 4(2):193–241, 1995.

23 Jennifer Widom and Stefano Ceri, editors. *Active Database Systems: Triggers and Rules For Advanced Database Processing.* Morgan Kaufmann, 1996.

24 Marianne Winslett. *Updating Logical Databases.* Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1990.

Constraint CNF: SAT and CSP Language Under One Roof

Broes De Cat[1] **and Yuliya Lierler**[2]

1 **Independent Researcher, Londerzeel, Belgium**
`broes.decat@gmail.com`

2 **University of Nebraska at Omaha, Omaha, USA**
`ylierler@unomaha.edu`

Abstract

A new language, called constraint CNF, is proposed. It integrates propositional logic with constraints stemming from constraint programming. A family of algorithms is designed to solve problems expressed in constraint CNF. These algorithms build on techniques from both propositional satisfiability and constraint programming. The result is a uniform language and an algorithmic framework, which allow us to gain a deeper understanding of the relation between the solving techniques used in propositional satisfiability and in constraint programming and apply them together.

1998 ACM Subject Classification I.2.4 Knowledge Representation Formalisms and Methods

Keywords and phrases Propositional Satisfiability, Constraint Programming

Digital Object Identifier 10.4230/OASIcs.ICLP.2016.12

1 Introduction

Propositional satisfiability (SAT) and constraint programming (CP) are two areas of automated reasoning that focus on finding assignments that satisfy declarative specifications. However, the typical declarative languages, solving techniques and terminology in both areas are quite different. As a consequence, it is not straightforward to see their relation and how they could benefit from eachother. In this work, we introduce a language called constraint CNF, which will allow a formal study of this relation. We propose a graph-based algorithmic framework suitable to describe a family of algorithms designed to solve problems expressed in constraint CNF or, in other words, to find models of constraint CNF formulas. The described algorithms build on ideas coming from both SAT and CP. We view constraint CNF as a uniform, simple language that allows us to conglomerate solving techniques of SAT and CP.

The idea of connecting CP with SAT is not novel. Many solving methods investigated in CP fall back on realizing the connection between the two fields and, in particular, on devising translations from constraint satisfaction problem (CSP) specifications to SAT problem specifications, e.g. [12]. Also, methods that combine CP and SAT in a more sophisticated manner exist [10, 2]. Somewhat orthogonal to these efforts is constraint answer set programming (CASP) [5], which attempts to enhance the SAT-like solving methods that are available for processing logic programs under stable model semantics with CP algorithms. It is reasonable to believe that the two distinct research areas CASP and CP, coming from different directions, move towards a common ground. Yet, capturing the common ground is difficult. Research on SAT, CP, CASP each rely on their established terminology and classical results in earlier literature. This makes it difficult to borrow on the knowledge discovered in one of the communities and yet not available in another. Here we undertake the effort of introducing a language that can serve as a unifying ground for the investigation in different automated reasoning communities. We believe that this language will foster and promote new insights and breakthroughs in research communities that consider the computational task of model building.

The paper starts by introducing syntax and semantics of constraint CNF and relating the language to propositional logic and constraint satisfaction problems. We then adapt a graph-based framework, pioneered by Nieuwenhuis et al. [9] for describing backtrack-search algorithms, and design a family of algorithms suitable to solve problems expressed in constraint CNF. We conclude the paper by discussing specific instantiations of such algorithms.

2 Constraint CNF

A *domain* is a set of values. For example, the *Boolean* domain consists of truth values $\mathbb{B} = \{\mathbf{t}, \mathbf{f}\}$, whereas some possible *integer* domains include $\mathbb{Z}$, $\mathbb{N}$, and $\{1, 2, 4\}$. A *signature* is a set of *function symbols.* Each function symbol f is assigned

- a nonnegative integer called the *arity* and denoted ar^f, and
- a non-empty domain for every integer i such that $0 \leq i \leq ar^f$, denoted $\hat{d}^f$ when $i = 0$ and $\hat{d}^f_i$ when $1 \leq i \leq ar^f$.

We call $\hat{d}^f$ the *range* of f. In addition, a function symbol f of nonzero arity can be assigned a specific function from $\hat{d}^f_1 \times \cdots \times \hat{d}^f_n$ to $\hat{d}^f$, in which case we say that f is *interpreted* and denote its interpretation by ι^f. A function symbol is *Boolean* if its range is Boolean. A *propositional symbol* is a 0-ary Boolean function symbol. For a signature Σ, its *domain* is defined as the union of all involved domains. A signature is *finite-domain* when its domain is finite.

For signature Σ, *terms* over Σ are defined recursively:

- a 0-ary function symbol is a term, and
- for an n-ary ($n > 0$) function symbol f, if $t_1, \ldots, t_n$ are terms then $f(t_1, \ldots, t_n)$ is a term.

For a term τ of the form $f(t_1, \ldots, t_n)$, by τ^0 we denote its function symbol f. We define the *range* of a term τ, denoted as $\hat{d}^\tau$, as the domain $\hat{d}^{\tau^0}$. For a term $f(t_1, \ldots, t_n)$, we associate each argument position $1 \leq i \leq n$ with the domain $\hat{d}^f_i$ and assume that $\hat{d}^{t_i} \subseteq \hat{d}^f_i$. A term is *Boolean* if its range is (a subset of) the Boolean domain.

Let Σ be a signature. An *atom* over Σ is a Boolean term over Σ, a *literal* over Σ is an atom a over Σ or its negation $\neg a$. A *clause* over Σ is a finite nonempty set of literals over Σ. A *formula* over Σ is a finite set of clauses over Σ. We sometime drop the explicit reference to the signature. For an atom a we call its negation $\neg a$ a *complement.* For a literal $\neg a$ we call an atom a a *complement.* By $\bar{l}$, we denote the complement of a literal l.

▶ **Example 1.** Consider the following sample problem specification. *Two items, named* 1 *and* 2*, are available for sale, with associated prices of* 100 *and* 500*. We want to buy at least one item, but we should not exceed our budget of* 200. A possible signature to express these statements consists of 0-ary function symbols i_1, i_2 and bgt (abbreviating budget), unary function symbols pr (abbreviating price), buy, and binary interpreted function symbols $+$ and $\leq$, so that

$$\begin{aligned}
&\hat{d}^{i_1} = \{1\}, \hat{d}^{i_2} = \{2\}, \hat{d}^{pr} = \{100, 500\},\\
&\hat{d}^{bgt} = \{200\}, \hat{d}^{buy} = \hat{d}^{\leq} = \mathbb{B},\\
&\hat{d}^{+} = \{200, 600, 1000\},\\
&\hat{d}^{pr}_1 = \hat{d}^{buy}_1 = \{1, 2\}, \hat{d}^{\leq}_1 = \{200, 600, 1000\},\\
&\hat{d}^{+}_1 = \hat{d}^{+}_2 = \{100, 500\}, \hat{d}^{\leq}_2 = \{200\}.
\end{aligned}$$

Function symbols $+$ and $\leq$ are assigned respective arithmetic functions. We name the described signature Σ_1. In the following, terms that are composed using these functions are often written in a common infix-style. We also drop parenthesis following the common conventions. For instance, expression

$$pr(i_1) + pr(i_2) \leq bgt$$

stands for the term

$$\leq (+(pr(i_1), pr(i_2)), bgt).$$

The requirements of the problem are expressed by formula φ_1, consisting of four clauses:[1]

$$\{buy(i_1), buy(i_2)\} \tag{1}$$
$$\{\neg buy(i_1), \neg buy(i_2), pr(i_1) + pr(i_2) \leq bgt\} \tag{2}$$
$$\{\neg buy(i_1), pr(i_1) \leq bgt\} \tag{3}$$
$$\{\neg buy(i_2), pr(i_2) \leq bgt\}. \tag{4}$$

Intuitively, clause (1) expresses that at least one item is bought. Clause (2) states that if two items are bought then the sum of their prices should not exceed the budget. Clauses (3) (respectively, clause (4)) state that if an item i_1 (respectively, item i_2) is bought then its price should not exceed the budget.

For an n-ary function symbol f, we call any function $\hat{d}_1^f \times \ldots \times \hat{d}_n^f \mapsto 2^{\hat{d}^f}$ *approximating.* For instance, for the function symbol pr of Σ_1 defined in Example 1, functions α_1, α_2, α_3, presented below

$$\begin{array}{llll} \alpha_1: & (1) \mapsto \{100\} & \alpha_2: & (1) \mapsto \{100\} \\ & (2) \mapsto \{100, 500\} & & (2) \mapsto \{500\} \\ \alpha_3: & (1) \mapsto \{100\} & & \\ & (2) \mapsto \emptyset & & \end{array}$$

are approximating. An approximating function is *defining* if, for all possible arguments, it returns a singleton set. For example, function α_2 is defining. We identify defining functions with functions that, instead of a singleton set containing a domain value, return the domain value itself. Thus, we can represent α_2 as a function that maps 1 to 100 and 2 to 500. We call a function *inconsistent* if for some arguments it returns the empty set. Function α_3 is inconsistent. We say that a function $\alpha : D_1 \times \cdots \times D_n \mapsto 2^D$ *reifies* a function $\alpha' : D_1 \times \cdots \times D_n \mapsto 2^D$ if for any n-tuple $\vec{x} \in D_1 \times \cdots \times D_n$ it holds that $\alpha(\vec{x}) \subseteq \alpha'(\vec{x})$. For example, functions α_2, α_3 reify α_1, and α_3 reifies α_2. The reification-relation is transitive and reflexive.

We are now ready to define a semantics of the constraint CNF formulas. An *interpretation* I over a signature Σ consists of an approximating function for every function symbol in Σ; for a function symbol $f \in \Sigma$, by f^I we denote the approximating function of f in I.

We call f^I an *interpretation of a function symbol* f in I. An interpretation that contains only defining functions is *total.* An interpretation that contains an inconsistent function is

[1] By introducing additional function symbols, the representation can be made more elaboration tolerant with regards to increasing the number of items. For simplicity of the example, a different representation was chosen here.

inconsistent. For interpretations I and I' over Σ we say that I *reifies* I' (or, I is a *reification of* I') if for every function symbol $f \in \Sigma$, f^I reifies $f^{I'}$.

Let Σ be a signature, τ be a term over Σ, and I be a total interpretation over Σ. By τ^I we denote the value *assigned* to a term τ by I, defined recursively as

- f^I if τ has the form f, and
- $f^I(t_1^I, \ldots, t_n^I)$ if τ has the form $f(t_1, \ldots, t_n)$.

We say that I *satisfies*

- an atom a over Σ, denoted $I \models a$, if $a^I = \mathbf{t}$,
- a literal $\neg a$ over Σ, denoted $I \models \neg a$, if $a^I = \mathbf{f}$,
- a clause $C = \{l_1, \ldots, l_n\}$ over Σ, denoted $I \models C$, if I satisfies any literal l_i in C,
- a formula $\varphi = \{C_1, \ldots, C_n\}$ over Σ, denoted $I \models \varphi$, if I satisfies every clause C_i in φ.

We say that I is a *model* of a formula φ if

- I satisfies φ and
- for any interpreted function symbol f in Σ, f^I coincides with ι^f.

We say that formula φ is satisfiable when φ has a model and *unsatisfiable*, otherwise.

▶ **Example 2** (Continued from Example 1). Consider signature Σ_1 and formula φ_1. All models of φ_1 interpret function symbols i_1, i_2, bgt as follows:

$$i_1 \quad [\mapsto 1] \quad i_2 \quad [\mapsto\ 2] \quad bgt \quad [\mapsto\ 200].$$

They differ in their interpretation of *pr* and *buy*. Indeed, there are five models, one of which is the following:

$$\begin{array}{lll} model_1 & pr & [(1) \mapsto 100, (2) \mapsto 500] \\ & buy & [(1) \mapsto \mathbf{t}, (2) \mapsto \mathbf{f}]. \end{array}$$

2.1 Relation to propositional logic and constraint programming

It is easy to see that in case when the signature is composed only of propositional symbols, constraint CNF formulas coincide with classic propositional logic formulas in conjunctive normal form (classic CNF formulas). Indeed, we can identify a clause $\{l_1, \ldots, l_n\}$ in constraint CNF with a clause $l_1, \cdots, l_n$ in propositional logic, whereas a constraint CNF formula corresponds to conjunction of its elements in propositional logic.

A *constraint satisfaction problem* (CSP) is a triple $\langle V, D, C\rangle$, where V is a set of variables, D is a finite set of values, and C is a set of constraints. Every constraint is a pair $\langle (v_1, \ldots, v_n), R\rangle$, where $v_i \in V$ $(1 \leq i \leq n)$ and R is an n-ary relation on D. An *assignment* is a function from V to D. An assignment ν *satisfies* a constraint $\langle (v_1, \ldots, v_n), R\rangle$ if $(\nu(v_1), \ldots, \nu(v_n)) \in R$. A *solution* to $\langle V, D, C\rangle$ is an assignment that satisfies all constraints in C. We map a CSP $\mathcal{C} = \langle V, D, C\rangle$ to an "equivalent" constraint CNF theory $F_{\mathcal{C}}$ as follows. We define a signature $\Sigma_{\mathcal{C}}$ to be composed of

- 0-ary function symbols f_v so that $\hat{d}^{f_v} = D$ for each variable $v \in V$, and
- interpreted n-ary Boolean function symbols f_c, one for each constraint $c = \langle (x_1, \ldots, x_n), R\rangle \in C$.

Function ι^{f_c} maps n-tuple d^n in Cartesian product D^n to $\mathbf{t}$ if $d^n \in R$, otherwise ι^{f_c} maps $d^n \in D^n$ to $\mathbf{f}$. For each constraint $c = \langle (v_1, \ldots, v_n), R\rangle \in C$, the constraint CNF $F_{\mathcal{C}}$ includes a clause $\{f_c(f_{v_1}, \ldots, f_{v_n})\}$. Models of $F_{\mathcal{C}}$ are in one-to-one correspondence with solutions of $\mathcal{C}$: indeed, an interpretation I is a model of $F_{\mathcal{C}}$ if and only if an assignment ν defined as follows $\nu(v) = f_v^I$ for each variable $v \in V$ is a solution to $\mathcal{C}$.

3 DPLL Approach for Constraint CNF

The DPLL decision algorithm [1] and its enhancement CDCL [8] are at the heart of most modern SAT solvers. These algorithms also became a basis for some of the search procedures in related areas such as satisfiability modulo theories [9], answer set programming [4], constraint answer set programming [5] and constraint programming [12, 10]. Here, we present an extension of DPLL that is applicable in the context of the constraint CNF language.

The DPLL algorithm is applied to a classic CNF formula. Let F be such a formula. Informally, the search space of DPLL on F consists of all assignments of the symbols in F. During its application, DPLL maintains a record of its computation state that corresponds to a currently considered family of assignments. When DPLL terminates, it either indicates that given formula F is unsatisfiable or the current state of computation corresponds to a model of F. Nieuwenhuis et al. [9] pioneered a graph-based (or transition system based) approach for representing the DPLL procedure (and its enhancements). They introduced the "Basic DPLL system", which is a graph so that its nodes represent possible states of computation of DPLL, while the edges represent possible transitions between the states. As a result, any execution of the DPLL algorithm can be mapped onto a path of the Basic DPLL system. Here we introduce a graph that captures a backtrack-search procedure for establishing whether a constraint CNF formula is satisfiable, an "entailment graph". We refer to a procedure captured by this graph as "an ENTAIL procedure". The relation between the entailment graph and an ENTAIL algorithm is similar to that between the Basic DPLL system and the DPLL algorithm. Before presenting the entailment graph we introduce two key concepts used in its definition: coherent encoding and entailment.

3.1 Coherent encodings and entailment

We begin by presenting some required terminology. An atom is *propositional* if it is a propositional symbol, a literal is *propositional* if it is a propositional atom or a negation of a propositional atom. We say that a signature is *propositional* if it is composed of propositional symbols only. For a propositional signature Σ, we define $\hat{\Sigma}$ as

$$\Sigma \cup \{\neg a \mid a \in \Sigma\}$$

It is easy to identify interpretations over a propositional signature Σ with sets of propositional literals over $\hat{\Sigma}$. Indeed, consider a mapping $\mathcal{L}$ from interpretations over Σ to $2^{\hat{\Sigma}}$ so that for an interpretation I over Σ, $\mathcal{L}(I)$ results in a set

$$\{f,\ \neg f \mid f^I = \emptyset\} \cup \{f \mid f^I = \mathbf{t}\} \cup \{\neg f \mid f^I = \mathbf{f}\}.$$

$\mathcal{L}^{-1}$ is a mapping from $2^{\hat{\Sigma}}$ to interpretations over Σ so that for a set M of literals over Σ, $\mathcal{L}^{-1}(M)$ is an interpretation where for every symbol $f \in \Sigma$

$$f^I = \begin{cases} \mathbf{t}, & \text{if } f \in M, \neg f \notin M \\ \mathbf{f}, & \text{if } f \notin M, \neg f \in M \\ \emptyset, & \text{if } f, \neg f \in M \\ \mathbb{B}, & \text{otherwise.} \end{cases}$$

A state of the DPLL procedure is meant to capture the family of assignments currently being explored. These families of assignments of classic CNF formulas can be referred to by means of sets of propositional literals. For instance, a state $\{a\ \neg b\}$ over a propositional signature $\{a\ b\ c\}$ intuitively suggests that assignments captured by the sets $\{a\ \neg b\ c\}$ and $\{a\ \neg b\ \neg c\}$ of literals are of immediate interest. The signature of a general constraint CNF formula goes beyond propositional symbols. To adapt the "propositional states" of DPLL to the constraint CNF formalism one has to ensure that the maintained state of computation can be mapped into an interpretation for a signature that includes non-propositional symbols. One approach to achieve this goal is to use auxiliary propositional symbols to encode the state of the approximating functions of non-propositional symbols in the formula's signature. This method is sometimes used by constraint programming solvers, for example, see [11]. We follow this approach in developing ENTAIL procedures.

From now on we assume only finite-domain signatures. We start by presenting a generalized concept of an "encoding" and summarize the important properties it should exhibit to be applicable in the scope of ENTAIL procedures that we present next. In the following section we illustrate that the equality or direct encoding studied in CP satisfies such properties.

An *encoding* is a 4-tuple (Σ, Σ', m, m'), where Σ is a signature, Σ' is a propositional signature, m is a function that maps interpretations in Σ into interpretations in Σ', and m' is a function that maps interpretations in Σ' into interpretations in Σ. We say that an encoding (Σ, Σ', m, m') is *coherent* when the following conditions (properties) hold

1. For a total interpretation I over Σ, $m(I)$ results in a total interpretation over Σ' and $I = m'(m(I))$.
2. For a total interpretation I' over Σ', $m'(I')$ results in either a total interpretation over Σ so that $I' = m(m'(I'))$ or an inconsistent interpretation over Σ.
3. For any interpretations I'_1, I'_2, and a literal l over Σ' such that (i) l is in I'_1 and its complement is in I'_2, and (ii) interpretations $m'(I'_1)$ and $m'(I'_2)$ are consistent, it holds that $m'(I'_1)$ does not reify $m'(I'_2)$.
4. For any consistent interpretations I'_1, I'_2 over Σ' such that I'_2 reifies I'_1, $m'(I'_2)$ reifies $m'(I'_1)$.
5. For any consistent interpretations I_1, I_2 over Σ such that I_2 reifies I_1, $m(I_2)$ reifies $m(I_1)$.
6. For any total interpretation I over Σ, a non-total interpretation I' over Σ' such that I reifies $m'(I')$, and any atom a in Σ' such that neither a nor $\neg a$ in I', it holds that I reifies $m'(I' \cup \{a\})$ or I reifies $m'(I' \cup \{\neg a\})$.
7. For any literal l over Σ', $\{l\} \subseteq m(m'(\{l\}))$.

The properties of coherent encodings allow us to shift between the interpretations in two signatures Σ and Σ' in a manner that proves to be essential to design of ENTAIL procedures for constraint CNF formulas.

Let Σ be a signature, φ be a formula over Σ, I be an interpretation over Σ, and f be a function symbol in Σ. Formula φ *entails* an approximating function f^α, denoted as $\varphi \models f^\alpha$, if for every model J of φ, f^J reifies f^α. Formula φ *entails* an approximating function f^α *with respect to interpretation* I, denoted as $\varphi \models_I f^\alpha$, when for every model J of φ that is a reification of I, f^J reifies f^α and f^α reifies f^I. Formula φ *entails* interpretation I, denoted as $\varphi \models I$ if $\varphi \models g^I$ for every function symbol g in Σ. Formula φ *entails* an interpretation I' over Σ *with respect to interpretation* I, denoted as $\varphi \models_I I'$ if $\varphi \models_I g^{I'}$ for every function symbol g in Σ. We now remark on some properties about entailment: (i) when there is no model of φ that reifies I then any approximating function is entailed w.r.t. I, (ii) when there is at least one model of φ that is a reification of I then no inconsistent approximating function is entailed, (iii) φ entails any interpretation including inconsistent ones when φ has no models, (iv) φ entails any interpretation (including inconsistent) I' w.r.t. I when φ has no models that reify I, and (v) φ entails any interpretation with respect to inconsistent interpretation I.

3.2 Abstract Constraint CNF Solver.

We are now ready to define nodes of the entailment graph and its transitions. For a set $\mathcal{B}$ of propositional atoms (which is also a propositional signature), a *state* relative to $\mathcal{B}$ is either the distinguished state *Failstate* or a (possibly empty) list M of literals over $\mathcal{B}$, where (i) no literal is repeated twice and (ii) some literals are possibly annotated by Δ. For instance, list $a\ \neg a^\Delta$ is a state relative to $\{a, b\}$, while $a\ a^\Delta$ is not. The tag Δ marks literals as *decision* literals. Frequently, we consider M as a set of literals and hence as an interpretation over a propositional signature, ignoring the annotations and the order among its elements. We say that M is *inconsistent* if some atom a and its negation $\neg a$ occur in it. E.g., states $b^\Delta\ \neg b$ and $b\ a\ \neg b$ are inconsistent.

Given an encoding $E = (\Sigma, \Sigma', m, m')$, we define the *entailment graph* $\text{ENT}_{\varphi,E}$ for a formula φ over Σ as follows. The set of nodes of $\text{ENT}_{\varphi,E}$ consists of the states relative to Σ'. The edges of the graph $\text{ENT}_{\varphi,E}$ are specified by four transition rules:

Entailment Propagate:	$M \Rightarrow M\ l$	if $\varphi \models_{m'(M)} I$ and $l \in m(I)$
Decide:	$M \Rightarrow M\ l^\Delta$	if $l \notin M$ and $\bar{l} \notin M$
Fail:	$M \Rightarrow \mathit{Failstate}$	if $\begin{cases} m'(M) \text{ is inconsistent, and} \\ \text{no decision literal is in } M \end{cases}$
Backtrack:	$P\ l^\Delta\ Q \Rightarrow P\ \bar{l}$	if $\begin{cases} m'(P\ l^\Delta\ Q) \text{ is inconsistent,} \\ \text{and no decision literal is in } Q. \end{cases}$

A node (state) in the graph is *terminal* if no edge originates in it. The following proposition gathers key properties of the graph $\text{ENT}_{\varphi,E}$ under assumption that E is a coherent encoding.

▶ **Proposition 3.** *For a signature Σ, a formula φ over Σ, and a coherent encoding $E = (\Sigma, \Sigma', m, m')$,*

(a) *graph $\text{ENT}_{\varphi,E}$ is finite and acyclic,*

(b) *any terminal state M of $\text{ENT}_{\varphi,E}$ other than Failstate is such that $m'(M)$ is a model of φ,*

(c) *state Failstate is reachable from $\emptyset$ in $\text{ENT}_{\varphi,E}$ if and only if φ has no models.*

Thus, to decide whether a CNF formula φ over Σ has a model, it is sufficient to (i) find any coherent encoding $E = (\Sigma, \Sigma', m, m')$ and (ii) find a path leading from node $\emptyset$ to a terminal node M in $\text{ENT}_{\varphi,E}$. If $M = \mathit{Failstate}$, φ has no models. Otherwise, M is a model of φ. Conditions (b) and (c) of Proposition 3 ensure the correctness of this procedure, while condition (a) ensures that this procedure terminates. We refer to any algorithm of this kind as an ENTAIL procedure.

An implementation of an ENTAIL algorithm in its full generality is infeasible due to the complexity of the condition of the *Entailment Propagate* transition rule. Yet, for various special cases, efficient methods exist. The DPLL algorithm for classic CNF formulas relies on this observation. Recall that classic CNF formulas can be viewed as constraint CNF formulas over a propositional signature. We now define the graph DP_F that coincides with aforementioned Basic DPLL system. Let E_p denote the encoding (Σ, Σ, id, id), where Σ is a propositional signature and id is an identity function from Σ to Σ. The set of nodes of DP_F coincide with the nodes of ENT_{F,E_p}. The edges of the graph DP_F are specified by the three transition rules of ENT_{F,E_p}, namely, *Decide*, *Fail*, *Backtrack*, and the clause-specific propagate rule

$$\textit{Unit Propagate}: \quad M \Rightarrow M\ l \quad \text{if} \left\{ \begin{array}{l} \{l_1, \ldots, l_n, l\} \in F \text{ and} \\ \{\overline{l_1}, \ldots, \overline{l_n}\} \subseteq M \end{array} \right.$$

It turns out that if the condition of the transition rule *Unit Propagate* holds then the condition of *Entailment Propagate* in the graph ENT_{F,E_p} also holds. The converse is not true. The DP_F graph is a subgraph of ENT_{F,E_p}. Yet, Proposition 3 holds for the graph DP_F. Proof of this claim was presented in [9, 7].

We now present several incarnations of the $\text{ENT}_{\varphi,E}$ framework that encapsulate the *Unit Propagate* rule of DPLL in a meaningful way.

Let Σ be a signature, $E = (\Sigma, \Sigma', m, m')$ a coherent encoding, φ a formula over Σ, and F a classic CNF formula over Σ'. We say that F *respects* φ when every model I of φ is such that $m(I)$ is also a model of F; we also say that F *captures* φ when F respects φ and every model M of F is such that $m'(M)$ is a model of φ. It is obvious that the graph DP_F can be used to decide whether formula φ has a model when F captures φ. We define a graph $\text{ENT-UP}_{\varphi,E,F}$ as follows: (i) its nodes are the nodes of $\text{ENT}_{\varphi,E}$, and (ii) its edges are defined by the transition rules of $\text{ENT}_{\varphi,E}$ and the transition rule *Unit Propagate*. It turns out that when F respects φ, the graphs $\text{ENT-UP}_{\varphi,E,F}$ and $\text{ENT}_{\varphi,E}$ coincide:

▶ **Proposition 4.** *For a coherent encoding $E = (\Sigma, \Sigma', m, m')$, a formula φ over Σ, a classic CNF formula F over Σ' that respects φ, and some nodes M and $M\ l$ in the graph $\text{ENT}_{\varphi,E}$, if the transition rule Unit Propagate suggests the edge between M and $M\ l$ then this edge is present in $\text{ENT}_{\varphi,E}$ (due to the transition rule Entailment Propagate).*

Consider a new graph $\text{ENT}'_{\varphi,E,F}$ constructed from $\text{ENT}_{\varphi,E}$ by dropping some of its edges. In particular, given a node M in $\text{ENT}_{\varphi,E}$ to which *Unit Propagate* is applicable, we drop all of the edges from M that cannot be characterized by the application of *Unit Propagate*. It turns out that Proposition 3 holds for the graph $\text{ENT}'_{\varphi,E,F}$, when F respects φ. This suggests that the "more respecting" the propositional formula is to a given constraint CNF formula, the more we can rely on the *Unit Propagate* rule of DPLL and the less we have to rely on propagations that go beyond Boolean reasoning. Another interesting propagator based on *Entailment Propagate* is due to the transition rule

$$\textit{Model Check}: \quad M \Rightarrow M\ l \quad \text{if} \left\{ \begin{array}{l} M \text{ is a model of } F, \\ \varphi \models_{m'(M)} I, \text{ and } l \in m(I) \end{array} \right.$$

This propagator is such that it is only applicable to the states that represent total interpretations over Σ'. It is easy to see that any edge due to *Model Check* is also an edge due to *Entailment Propagate*. It turns out that given classic CNF formula F that respects φ, Proposition 3 also holds for the graph $\text{ENT}''_{\varphi,E,F}$ constructed from $\text{ENT}'_{\varphi,E,F}$ by dropping all of the edges not due to *Unit Propagate* or *Model Check*. The essence of this graph is in the following: to adapt the DPLL algorithm for solving a constraint CNF formula φ over signature Σ, it is sufficient to (i) find some coherent encoding of the form $E = (\Sigma, \Sigma', m, m')$, (ii) find some classic CNF formula F over Σ' that respects φ, (iii) apply DPLL to F, and (iv) implement a check that given any model of F can verify whether that model is also a model of φ. Next section presents one specific coherent encoding and a family of mappings that given a constraint CNF formula produces a classic CNF formula respecting it.

4 Equality Encoding

Walsh [12] describes a mapping from CSP to SAT that he calls "direct encoding". Similar ideas are applicable in the realm of constraint CNF for producing a coherent encoding and classic CNF formulas respecting and capturing given constraint CNF formulas. We make this statement precise by (a) defining a coherent "equality" encoding $\mathcal{E}$ and, (b) introducing mappings from a constraint CNF formula φ to classic CNF formulas that respect φ.

For a function symbol f, we denote the Cartesian product $\hat{d}^f_1 \times \cdots \times \hat{d}^f_{ar^f}$ by $\hat{D}_f$. For a non-propositional function symbol $f \in \Sigma$, by $f^{\equiv}$ we denote the set of propositional symbols constructed as follows:

$$\{[f^{\vec{x}} \doteq v] \mid \vec{x} \in \hat{D}_f \text{ and } v \in \hat{d}^f\}.$$

For a signature Σ, by $\Sigma^{\equiv}$ we denote the signature that consists of all propositional symbols in Σ and the propositional symbols in $f^{\equiv}$ for every non-propositional function symbol f in Σ. For example, for Σ_1 defined in Example 1 signature $\Sigma_1^{\equiv}$ includes, among others, following elements

$$\begin{aligned}
&[i_1 \doteq 1];\ [i_2 \doteq 2];\ [bgt \doteq 200];\\
&[pr^1 \doteq 100];\ [pr^1 \doteq 500];\ [pr^2 \doteq 100];\ [pr^2 \doteq 500];\\
&[buy^1 \doteq \mathbf{t}];\ [buy^1 \doteq \mathbf{f}];\ [buy^2 \doteq \mathbf{t}];\ [buy^2 \doteq \mathbf{f}];\\
&[+^{100,100} \doteq 200];\ [+^{100,500} \doteq 200];\ [+^{200,100} \doteq 500];\\
&[\leq^{100,200} \doteq \mathbf{t}];\ [\leq^{600,200} \doteq \mathbf{t}];\ [\leq^{100,200} \doteq \mathbf{f}].
\end{aligned}$$

Intuitively, the collection of the symbols of the form $[f^{\vec{x}} \doteq v]$ in $f^{\equiv}$ is meant to "capture" the approximating function of non-propositional function symbol f in Σ by means of approximating functions for the elements of $f^{\equiv}$ in $\Sigma^{\equiv}$.

We now present a mapping ϵ_* from an approximating function α for a non-propositional function symbol f into an interpretation M over signature $f^{\equiv}$: every symbol $[f^{\vec{x}} \doteq v]$ in $f^{\equiv}$ is interpreted as

$$[f^{\vec{x}} \doteq v]^M = \begin{cases} \mathbf{t}, & \text{if } \alpha(\vec{x}) = \{v\} \\ \mathbf{f}, & \text{if } v \notin \alpha(\vec{x}) \\ \mathbb{B}, & \text{otherwise.} \end{cases}$$

For an interpretation I over Σ, by $\epsilon(I)$ we denote the interpretation over $\Sigma^{\equiv}$ constructed as follows (i) for every propositional symbol f in Σ, $f^{\epsilon(I)} = f^I$, and (ii) for every non-propositional function symbol f in Σ, $\epsilon(I)$ includes the elements of $\epsilon_*(f^I)$.

Similarly, for a non-propositional symbol f, we define a mapping $\epsilon^{\equiv}_*$ which, given an interpretation M over $f^{\equiv}$, maps M into an approximating function α for f: for every $\vec{x} \in \hat{D}_f$,

$$\alpha(\vec{x}) = \begin{cases} \emptyset, \text{if } [f^{\vec{x}} \doteq v]^M = [f^{\vec{x}} \doteq v']^M = \mathbf{t} \text{ and } v \neq v' \\ \{v\} \text{ otherwise, if } [f^{\vec{x}} \doteq v]^M = \mathbf{t} \\ \hat{d}^f \setminus \{v \mid [f^{\vec{x}} \doteq v]^M = \mathbf{f}\} \text{ otherwise.} \end{cases}$$

For an interpretation I and signature Σ, by $I[\Sigma]$ we denote the set of all approximating functions of Σ-elements in I: $\{f^I | f \in \Sigma\}$. For a signature Σ and an interpretation M over $\Sigma^{\equiv}$, by $\epsilon^{\equiv}(M)$ we denote the interpretation over Σ constructed as follows (i) for every propositional symbol f in Σ, $f^{\epsilon^{\equiv}(M)} = f^M$, and (ii) for every non-propositional function symbol f in Σ, $f^{\epsilon^{\equiv}(M)} = \epsilon^{\equiv}_*(M[f^{\equiv}])$.

For a signature Σ, we call an encoding $(\Sigma, \Sigma^{\equiv}, \epsilon, \epsilon^{\equiv})$ an *equality* encoding.

▶ **Proposition 5.** *For a signature Σ, its equality encoding is coherent.*

We now present several mappings from constraint CNF formulas to classic CNF formulas based on equality encoding. Consider a formula φ over signature Σ and the equality encoding $\mathcal{E} = (\Sigma, \Sigma^{\equiv}, \epsilon, \epsilon^{\equiv})$. By $F_{\varphi,\Sigma^{\equiv}}$ we denote a propositional formula constructed as the union of the following sets of clauses:

- for every interpreted function symbol f in Σ and every tuple $\vec{x}$ in $\hat{D}_f$, a set consisting of unit clauses over $\Sigma^{\equiv}$ that ensures that f is interpreted according to ι^f:

$$\bigcup_{v \in \hat{d}^f} \{\neg[f^{\vec{x}} \doteq v] \mid \iota^f(\vec{x}) \neq v\} \cup \\ \{[f^{\vec{x}} \doteq v] \mid \iota^f(\vec{x}) = v\}.$$

- for every other function symbol f in Σ and every tuple $\vec{x}$ in $\hat{D}_f$ (i) a clause over $\Sigma^{\equiv}$ that ensures that f is associated with an approximating function:

  $$\{[f^{\vec{x}} \doteq v] \ \mid \ v \in \hat{d}^f\}.$$

  and (ii) a set of clauses over $\Sigma^{\equiv}$ that ensures that each functional symbol is associated with a defining approximating function:

  $$\bigcup_{v,v' \in \hat{d}^f, v \neq v'} \{\neg[f^{\vec{x}} \doteq v], \neg[f^{\vec{x}} \doteq v']\}.$$

▶ **Proposition 6.** *For a signature Σ, a constraint CNF formula φ over Σ, and respective equality encoding $\mathcal{E} = (\Sigma, \Sigma^{\equiv}, \epsilon, \epsilon^{\equiv})$, propositional formula $F_{\varphi,\Sigma^{\equiv}}$ (as well as any formula over $\Sigma^{\equiv}$ constructed from $F_{\varphi,\Sigma^{\equiv}}$ by dropping some of its clauses) respects φ.*

This proposition tells us that we can use the graph $\text{ENT}'_{\varphi,E,F}$ and $\text{ENT}''_{\varphi,E,F}$ for verifying whether formula φ is satisfiable.

5 Proofs for "Constraint CNF"

Proof of Proposition 5.

Proof of Property 1. This property is apparent from the constructions of the mappings.

Proof of Property 2. By contradiction. Assume that for a total interpretation I' over Σ', $m'(I')$ resulted in a consistent non-total interpretation I over Σ. Thus, there is a function symbol $f \in \Sigma$ so that for some $\vec{x} \in \hat{D}_f$, $f^I(\vec{x})$ results in a set whose cardinality if greater than 1. From f^I construction it follows that (i) there is no $v \in \hat{d}^f$ such that $[f^{\vec{x}} \doteq v] = \mathbf{t}$ and (ii) there are at least two values $v, v' \in \hat{d}^f$ such that $v \neq v'$, $[f^{\vec{x}} \doteq v] \neq \mathbf{f}$ and $[f^{\vec{x}} \doteq v] \neq \mathbf{f}$. Consequently, atoms $[f^{\vec{x}} \doteq v]^I = [f^{\vec{x}} \doteq v']^{I'} = \mathbb{B}$. This contradicts the fact that I' is total.

Proof of Property 3. Consider interpretations I'_1, I'_2, and a literal l over Σ' such that (i) l is in I'_1 and its complement is in I'_2, and (ii) interpretations $\epsilon^{\equiv}(I'_1)$ and $\epsilon^{\equiv}(I'_2)$ are consistent. We show that it holds that $\epsilon^{\equiv}(I'_1)$ does not reify $\epsilon^{\equiv}(I'_2)$. Recall that an interpretation $\epsilon^{\equiv}(I'_1)$ reifies $\epsilon^{\equiv}(I'_2)$ if any function in $\epsilon^{\equiv}(I'_1)$ reifies a corresponding function in $\epsilon^{\equiv}(I'_2)$.

Case 1. Literal l is of the form $[f^{\vec{x}} \doteq v]$. Since $\epsilon^{\equiv}(I'_1)$ is consistent we derive that $[f^{\vec{x}} \doteq v]^{\epsilon^{\equiv}(I'_1)} = \{v\}$. By the definition of $\epsilon^{\equiv}$, $\epsilon^{\equiv}(I'_2)$ satisfies the following requirement $[f^{\vec{x}} \doteq v]^{\epsilon^{\equiv}(I'_2)} \subseteq \hat{d}^f \setminus \{v\}$. We derive that $\epsilon^{\equiv}(I'_1)$ does not reify $\epsilon^{\equiv}(I'_2)$.

Case 2. Literal l is of the form $\neg[f^{\vec{x}} \doteq v]$. By the definition of $\epsilon^{\equiv}$, we derive that $[f^{\vec{x}} \doteq v]^{\epsilon^{\equiv}(I'_1)} \subseteq \hat{d}^f \setminus \{v\}$. Since $\epsilon^{\equiv}(I'_1)$ is consistent, we also derive that $[f^{\vec{x}} \doteq v]^{\epsilon^{\equiv}(I'_1)} \neq \emptyset$. Since $\epsilon^{\equiv}(I'_2)$ is consistent we derive that

$$[f^{\vec{x}} \doteq v]^{\epsilon^{\equiv}(I'_2)} = \{v\} \tag{5}$$

Proof of Property 4. Take any two consistent interpretations I'_1 and I'_2 over $\Sigma^{\equiv}$ such that I'_2 reifies I'_1. We illustrate that $\epsilon^{\equiv}(I'_2)$ reifies $\epsilon^{\equiv}(I'_1)$. This is the case if for every nonpropositional function symbol $f \in \Sigma$, $f^{\epsilon^{\equiv}(I'_2)} \subseteq f^{\epsilon^{\equiv}(I'_1)}$.

Take any nonpropositional function symbol $f \in \Sigma$. Recall that atoms $f^{\equiv}$ are the ones that are used to define approximating function of f via mapping $\epsilon^{\equiv}$. Take any argument list $\vec{x} \in \hat{D}_f$. We illustrate that $f^{\epsilon^{\equiv}(I'_2)}(\vec{x}) \subseteq f^{\epsilon^{\equiv}(I'_1)}(\vec{x})$.

Case 1. I'_2 is such that $[f^{\vec{x}} \doteq v]^{I'_2} = [f^{\vec{x}} \doteq v']^{I'_2} = \mathbf{t}$ for some values $v \neq v'$. Then, $f^{\epsilon^{\equiv}(I'_2)}(\vec{x}) = \emptyset$. Condition $f^{\epsilon^{\equiv}(I'_2)} \subseteq f^{\epsilon^{\equiv}(I'_1)}$ trivially holds.

Case 2. I'_2 is such that $[f^{\vec{x}} \doteq v]^{I'_2} = \mathbf{t}$ for some value v, and there is no other value $v' \neq v$ such that $[f^{\vec{x}} \doteq v']^{I'_2} = \mathbf{t}$. By the $\epsilon^{\equiv}$ construction, $f^{\epsilon^{\equiv}(I'_2)}(\vec{x}) = \{v\}$. Since I'_2 is consistent and I'_2 reifies I'_1 it follows that (i) for v either $[f^{\vec{x}} \doteq v]^{I'_1} = \mathbf{t}$ or $[f^{\vec{x}} \doteq v]^{I'_1} = \mathbb{B}$ and (ii) there is no other value $v' \neq v$ such that $[f^{\vec{x}} \doteq v']^{I'_2} = \mathbf{t}$. Consequently, by $\epsilon^{\equiv}_*$ mapping definition, it is either $f^{\epsilon^{\equiv}(I'_2)}(\vec{x}) = \{v\}$ or $f^{\epsilon^{\equiv}(I'_2)}(\vec{x}) = \{v\} \cup S$ where S is some subset of $\hat{d}^f$. Consequently, condition $f^{\epsilon^{\equiv}(I'_2)} \subseteq f^{\epsilon^{\equiv}(I'_1)}$ holds.

Case 3. I'_2 is such that $[f^{\vec{x}} \doteq v]^{I'_2} = \mathbf{f}$ or $[f^{\vec{x}} \doteq v]^{I'_2} = \mathbb{B}$ for any $v \in \hat{d}^f$. Since I'_2 reifies I'_1, it also holds that $[f^{\vec{x}} \doteq v]^{I'_1} = \mathbf{f}$ or $[f^{\vec{x}} \doteq v]^{I'_1} = \mathbb{B}$ for any $v \in \hat{d}^f$ so that when $[f^{\vec{x}} \doteq v]^{I'_1} = \mathbf{f}$ it follows that $[f^{\vec{x}} \doteq v]^{I'_2} = \mathbf{f}$. By $\epsilon^{\equiv}_*$ construction (forth case apply), and it is apparent that $f^{\epsilon^{\equiv}(I'_2)}(\vec{x}) \subseteq f^{\epsilon^{\equiv}(I'_1)}(\vec{x})$.

Proof of Property 5. Consider consistent interpretations I_1, I_2 over Σ such that I_2 reifies I_1. We illustrate that $\epsilon(I_2)$ reifies $\epsilon(I_1)$. This is the case when for every (propositional) function symbol $f \in \Sigma^{\equiv}$, $f^{\epsilon(I_2)} = f^{\epsilon(I_1)}$ or $f^{\epsilon(I_1)} = \mathbb{B}$. This trivially holds for all propositional symbols f that are in $\Sigma \cap \Sigma^{\equiv}$. We consider here propositional symbols in $\Sigma^{\equiv} \setminus \Sigma$. Consider any symbol a of this kind. Symbol a is of the form $[f^{\vec{x}} \doteq v]$. From the fact that I_2 reifies I_1 following cases are possible:

Case 1. $f^{I_2}(\vec{x}) = f^{I_1}(\vec{x})$. From ϵ-mapping definition it follows that $a^{\epsilon(I_2)} = a^{\epsilon(I_1)}$.

Case 2. $f^{I_2}(\vec{x}) \subset f^{I_1}(\vec{x})$. From ϵ-mapping definition following cases are possible:

Case 2.1. $v \in f^{I_2}(\vec{x})$. It follows that there is also $v' \neq v$ so that $v, v' \in f^{I_1}(\vec{x})$. Consequently, $a^{\epsilon(I_1)} = \mathbb{B}$.

Case 2.2. $v \notin f^{I_2}(\vec{x})$ and $v \notin f^{I_2}(\vec{x})$. From ϵ-mapping definition it follows that $a^{\epsilon(I_2)} = a^{\epsilon(I_1)} = \mathbf{f}$.

Case 2.3. $v \notin f^{I_2}(\vec{x})$ and $v \in f^{I_1}(\vec{x})$.

Case 2.3.1. $f^{I_1}(\vec{x}) = \{v\}$. Then $f^{I_2}(\vec{x}) = \emptyset$. Impossible as I_2 is a consistent interpretation.

Case 2.3.2. Since, cardinality of $f^{I_1}(\vec{x})$ is greater than one and v in $f^{I_1}(\vec{x})$ it follows that $a^{\epsilon(I_1)} = \mathbb{B}$.

Proof of Property 6. Atom a is of the form $[f^{\vec{x}} \doteq v]$. It is easy to see that $\epsilon^{\equiv}(I' \cup \{a\})$ and $\epsilon^{\equiv}(I' \cup \{\neg a\})$, only differ from $\epsilon^{\equiv}(I')$ in how approximation function for function symbol f is defined. Thus, for any other function symbol $f' \neq f$ in Σ, approximating function for f' in I, $f'^{I'}$, reifies approximating function for f' in both $\epsilon^{\equiv}(I' \cup \{a\})$ and $\epsilon^{\equiv}(I' \cup \{\neg a\})$. Even more it only differs in how approximating function for f is defined on $\vec{x}$ arguments. We only have to show that approximating function f^I reifies an approximating function for f in $\epsilon^{\equiv}(I' \cup \{a\})$ or in $\epsilon^{\equiv}(I' \cup \{\neg a\})$ for the case of $\vec{x}$. Recall that I reifies $\epsilon^{\equiv}(I')$.

Case 1. $f^I(\vec{x}) = \{v\}$. Then $v \in f^{\epsilon^{\equiv}(I')}(\vec{x})$. From the fact that $a \notin I'$ and $\epsilon^{\equiv}$ construction we derive that there is no single atom of the form $[f^{\vec{x}} \doteq v']$ such that $v' \neq v$ and $[f^{\vec{x}} \doteq v']^{I'} = \mathbf{t}$. From $\epsilon^{\equiv}$ construction, it follows that $f^{\epsilon^{\equiv}(I' \cup \{a\})}(\vec{x}) = \{v\}$. Obviously, $f^I(\vec{x})$ reifies $f^{\epsilon^{\equiv}(I' \cup \{a\})}(\vec{x})$.

Case 2. I, $f^I(\vec{x}) = \{v'\}$ so that $v' \neq v$. Then $v' \in f^{\epsilon^{\equiv}(I')}(\vec{x})$.

Case 2.1. $f^{\epsilon^{\equiv}(I')}(\vec{x}) = \{v'\}$. It is easy to see from $\epsilon^{\equiv}$ construction that $f^{\epsilon^{\equiv}(I' \cup \{\neg a\})}(\vec{x}) = \{v'\}$ as well.

Case 2.2. $f^{\epsilon^{\equiv}(I')}(\vec{x}) = \{v'\} \cup S$ where cardinality $|S| \geq 1$. Since $\neg a \notin I'$, we derive that $v \in f^{\epsilon^{\equiv}(I')}(\vec{x})$. It is easy to see that $f^{\epsilon^{\equiv}(I' \cup \{\neg a\})}(\vec{x}) = f^{\epsilon^{\equiv}(I')}(\vec{x}) \setminus \{v\}$. It holds that $v' \in f^{\epsilon^{\equiv}(I' \cup \{\neg a\})}(\vec{x})$. Thus, $f^{I}(\vec{x})$ reifies $f^{\epsilon^{\equiv}(I' \cup \{a\})}(\vec{x})$.

Proof of Property 7. Recall that $\{l\}$ corresponds to an interpretation over $\Sigma^{\equiv}$ where all but one atom is assigned $\mathbb{B}$.

Case 1. l has the form $[f^{\vec{x}} \doteq v]$. $\epsilon^{\equiv}(\{l\}$ results in interpretation where $f^{\epsilon^{\equiv}(\{l\}}(\vec{x}) = \{v\}$ whereas all other approximating functions as well as approximating function for f on different arguments that $\vec{x}$ are mapped to $\mathbb{B}$. By ϵ construction, $\{l\} \in \epsilon(\epsilon^{\equiv}(\{l\}))$. Indeed, $\epsilon(\epsilon^{\equiv}(\{l\}))$ contains l as well as literals of the form $[f, v', \vec{x}^{f} \doteq]$ or all $v' \in \hat{d}^{f}$, where $v' \neq v$.

Case 2. l has the form $\neg[f^{\vec{x}} \doteq v]$ By the construction of $\epsilon^{\equiv}$ and ϵ, it is easy to see that $\{l\} = \epsilon(\epsilon^{\equiv}(\{l\}))$. ◀

We now present a lemma that captures important conditions that help to illustrate the correctness of Proposition 3.

▶ **Lemma 7.** *For any formula φ, a coherent encoding $E = (\Sigma, \Sigma', m, m')$, and a path from $\emptyset$ to a state $l_1 \ldots l_n$ in $\text{ENT}_{\varphi,E}$, every model I of φ is such that $l_i \in m(I)$ if I reifies $m'(\{l_j^{\Delta} | j \leq i\})$.*

Proof. By induction on the length of a path. Since the property trivially holds in the initial state $\emptyset$, we only need to prove that all transition rules of $\text{ENT}_{\varphi,E}$ preserve it.

Consider an edge $M \Rightarrow M'$ where M is a sequence $l_1 \ldots l_k$ such that every model I of φ is such that $l_i \in m(I)$ if I reifies $m'(\{l_j^{\Delta} | j \leq i\})$.

Entailment Propagate: M' is $M\ l_{k+1}$. Take any model I of φ such that I reifies $m'(\{l_j^{\Delta} | j \leq k+1\})$. It is easy to see that $\{l_j^{\Delta} | j \leq k+1\} = \{l_j^{\Delta} | j \leq k\}$. By the inductive hypothesis, since I reifies $m'(\{l_j^{\Delta} | j \leq k\})$, $M \subseteq m(I)$. We only have to illustrate that $l_{k+1} \in m(I)$. This trivially follows from the application condition of *Entailment Propagate*.

Decide: M' is $M\ l_{k+1}^{\Delta}$. Take any model I of φ such that I reifies $m'(\{l_j^{\Delta} | j \leq k+1\})$. By Property 4 (of coherent encoding), interpretation $m'(\{l_j^{\Delta} | j \leq k+1\})$ reifies $m'(\{l_j^{\Delta} | j \leq k\})$. Since reification is a transitive relation we derive that I reifies $m'(\{l_j^{\Delta} | j \leq k\})$. By the inductive hypothesis $M \subseteq m(I)$. We only have to illustrate that $l_{k+1} \in m(I)$. Obviously $l_{k+1} \in \{l_j^{\Delta} | j \leq k+1\}$.

Since I is a model and hence a total interpretation it may only reify consistent interpretations. Hence, $m'(\{l_j^{\Delta} | j \leq k+1\})$ is consistent. Interpretation $m(I)$ is a total over $\Sigma^{\equiv}$, by Property 1. Thus, either $l_{k+1} \in m(I)$ or $\overline{l_{k+1}} \in m(I)$. Assume $\overline{l_{k+1}} \in m(I)$. By Property 1, $I = m'(m(I))$ and reifies $m'(\{l_j^{\Delta} | j \leq k+1\})$. By Property 3, we derive a contradiction. Thus, $l_{k+1} \in m(I)$.

Fail: Obvious.

Backtrack: M has the form $P\ l_i^{\Delta}\ Q$ where Q contains no decision literals. M' is $P\ \overline{l_i}$. Take any model I of φ such that I reifies $m'(\{l_j^{\Delta} | j < i\})$.

By Property 6 two following cases are possible: I reifies $m'(\{l_j^{\Delta} | j < i\} \cup \{l_i\})$ or I reifies $m'(\{l_j^{\Delta} | j < i\} \cup \{\overline{l_i}\})$.

Case 1. I reifies $m'(\{l_j^{\Delta}|j < i\} \cup \{l_i\})$. By inductive hypothesis we derive that $M \subseteq m(I)$. Since I is a model and hence a total interpretation, by Property 1 $m(I)$ is total. We derive a contradiction since M is inconsistent interpretation. Hence, Case 2 must hold.

Case 2. I reifies $m'(\{l_j^{\Delta}|j < i\} \cup \{\overline{l_i}\})$. By Property 4, interpretation $m'(\{l_j^{\Delta}|j < i\} \cup \{\overline{l_i}\})$ reifies $m'(\{l_j^{\Delta}|j < i\})$. Since reification is a transitive relation we derive that I reifies $m'(\{l_j^{\Delta}|j \leq i\})$. By the inductive hypothesis $P \subseteq m(I)$. We only have to illustrate that $\bar{l} \in m(I)$. By Property 4, interpretation $m'(\{l_j^{\Delta}|j < i\} \cup \{\overline{l_i}\})$ reifies $m'(\{\overline{l_i})\})$ (the fact that set $\{l_j^{\Delta}|j < i\} \cup \{\overline{l_i}\}$ of literals is consistent follows from the properties of the *Decide* rules and simple inductive argument). Since reification is a transitive relation we derive that I reifies $m'(\{\overline{l_i}\})$. Since I is a model, it follows that $m'(\{\overline{l_i}\})$ is consistent. By Property 5, $m(I)$ reifies $m(m'(\{\overline{l_i}\}))$. By Property 7 $\{\overline{l_i}\} \subseteq m(m'(\{\overline{l_i}\}))$. By Property 1, $m(I)$ is a total interpretation and hence $\{\overline{l_i}\} \subseteq m(I)$. ◀

Proof of Proposition 3. Part (a) is proven following the arguments for Proposition 1 (a) in the paper by Lierler [6].

(b) Consider any terminal state M other than *Failstate*. From the fact that *Decide* is not applicable, we derive that M assigns all literals over Σ'. Similarly, since neither *Backtrack* nor *Fail* is applicable, M is consistent. By Property 2 of coherent encoding, it follows that $m'(M)$ is either (i) a total interpretation over Σ' or (ii) an inconsistent interpretation.

We now show that (i) holds: $m'(M)$ is a total interpretation. Assume the other case (ii): $m'(M)$ is inconsistent. Take any literal l over Σ' not in M (since M is consistent set of literals such l exists). Interpretation $m'(\{l\})$ is such that $\varphi \models_{m'(M)} m'(\{l\})$ since $m'(M)$ is inconsistent (recall that formula entails any interpretation w.r.t. any inconsistent interpretation). By Property 7 of coherent encoding, $l \in m(m'(\{l\}))$. It follows that *Entailment Propagate* is applicable in M. This contradicts the fact that M is terminal. Consequently, $m'(M)$ is a total interpretation.

We now illustrate that $m'(M)$ is a model of φ. By contradiction. Assume $m'(M)$ is not a model. Since $m'(M)$ is a total interpretation there is no other total interpretation that reifies it. Hence, there is no model that reifies $m'(M)$. It follows that φ entails any interpretation w.r.t. $m'(M)$. Consequently, interpretation $m'(\{l\})$ is such that $\varphi \models_{m'(M)} m'(\{l\})$. Following the argument presented in previous paragraph, we derive that rule *Entailment Propagate* is applicable in M that contradicts the fact that M is terminal. Consequently, $m'(M)$ is a model of φ.

(c) Left-to-right: Since *Failstate* is reachable from $\emptyset$, there is a state M without decision literals so that (i) M is inconsistent, and (ii) there exists a path from $\emptyset$ to M. By Lemma 7, any model I of φ is such that $M \subseteq m(I)$. Since I is a model it is also a total interpretation. By Property 1, $m(I)$ is also a total interpretation. This contradicts the facts that $M \subseteq m(I)$ and M is inconsistent. Indeed, there is an element τ in M such that $\tau^M = \emptyset$, whereas $\tau^{m(I)} = \mathbf{t}$ or $\tau^{m(I)} = \mathbf{f}$.

Right-to-left: From (a) it follows that there is a path from $\emptyset$ to some terminal state. By (b), this state cannot be different from *Failstate*, because φ is unsatisfiable. ◀

Proof of Proposition 6. Sketch: The second claim follows immediately from the proof of the former claim. It is sufficient to illustrate a more general statement, i.e., any interpretation I of φ is such that $\epsilon(I)$ is a model of $F_{\varphi,\Sigma^{\equiv}}$. This is easy to see by following the construction of ϵ and illustrating that every clause in $F_{\varphi,\Sigma^{\equiv}}$ is satisfied by $\epsilon(I)$ for any interpretation I. ◀

Proof of Proposition 4. Sketch: Follows from the properties of *Unit Propagate* and the respective formulas. ◀

6 Conclusions

In this paper we introduced the uniform language constraint CNF which integrates languages from SAT and CP. We also introduced a graph-based framework for a class of algorithms for constraint CNF. We generalized the concept of encoding and identified its essential properties. In the future, we will extend the framework with clause learning, non-finite domains, and constraint-based propagation rules as well as investigate the properties of other non-equality encodings available in CP literature. We will also extend constraint CNF to the case of logic programs so that the algorithms behind answer set solvers and constraint answer set solvers can be captured. This will allow us to formulate algorithms stemming from CP and constraint answer set programming in a uniform fashion to clarify their differences and similarities and facilitate cross-fertilization between the fields. An ultimate goal of this work is to illustrate how advanced solvers stemming from different research sub-communities can be captured as an algorithm for solving search problems stated in constraint CNF. The SAT solver MINISAT [3] is a true success story in model search automated reasoning. *MiniSAT Hack-track* has been an official track since 2009 at the SAT competition – a prime research venue for presenting and comparing state-of-the-art SAT solvers and techniques. The MINISAT authors envisioned such a future for the solver. Their motivation behind the development of the solver was to produce a middle-ware for a SAT solver design. This MINISAT middle-ware incorporates major SAT techniques and also allows a simple integration mechanism for investigating new features. We view constraint CNF as a step in the direction of designing middle-ware that incorporates not only advances in SAT but also other related areas.

Acknowledgments. We would like to thank Marc Denecker, Vladimir Lifschitz, and Mirek Truszczynski for valuable discussions related to this work. The work has been partially supported by FRI (Faculty Research International) of the University of Nebraska at Omaha.

References

1 Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, 1962.
2 Broes De Cat, Bart Bogaerts, Jo Devriendt, and Marc Denecker. Model expansion in the presence of function symbols using constraint programming. In *ICTAI*, pages 1068–1075. IEEE, 2013.
3 Niklas Een and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In *SAT*, 2005.
4 Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. Conflict-driven answer set solving: From theory to practice. *Artif. Intell.*, 187:52–89, 2012.
5 Martin Gebser, Max Ostrowski, and Torsten Schaub. Constraint answer set solving. In Patricia M. Hill and David Scott Warren, editors, *ICLP*, volume 5649 of *LNCS*, pages 235–249. Springer, 2009.
6 Yuliya Lierler. Abstract answer set solvers. In *Proceedings of International Conference on Logic Programming (ICLP)*, pages 377–391. Springer, 2008.
7 Yuliya Lierler. Abstract answer set solvers with backjumping and learning. *Theory and Practice of Logic Programming*, 11:135–169, 2011.
8 João P. Marques Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 131–153. IOS Press, 2009.

9 Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *J. ACM*, 53(6):937–977, 2006.

10 Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, 2009.

11 Naoyuki Tamura, Akiko Taga, Satoshi Kitagawa, and Mutsunori Banbara. Compiling finite linear CSP into SAT. *Constraints*, 14(2):254–272, 2009.

12 Toby Walsh. Sat v csp. In Rina Dechter, editor, *Principles and Practice of Constraint Programming, CP 2000*, volume 1894 of *Lecture Notes in Computer Science*, pages 441–456. Springer Berlin Heidelberg, 2000.

Constraint Propagation and Explanation over Novel Types by Abstract Compilation*

Graeme Gange[1] and Peter J. Stuckey[2]

1 Department of Computing and Information Systems, The University of Melbourne, Melbourne, Australia
gkgange@unimelb.edu.au

2 Data61, CSIRO and Department of Computing and Information Systems, The University of Melbourne, Melbourne, Australia
pstuckey@unimelb.edu.au

Abstract

The appeal of constraint programming (CP) lies in compositionality – the ability to mix and match constraints as needed. However, this flexibility typically does not extend to the types of variables. Solvers usually support only a small set of pre-defined variable types, and extending this is not typically a simple exercise: not only must the solver engine be updated, but then the library of supported constraints must be re-implemented to support the new type.

In this paper, we attempt to ease this second step. We describe a system for automatically deriving a native-code implementation of a global constraint (over novel variable types) from a declarative specification, complete with the ability to explain its propagation, a requirement if we want to make use of modern lazy clause generation CP solvers.

We demonstrate this approach by adding support for *wrapped-integer* variables to `chuffed`, a lazy clause generation CP solver.

1998 ACM Subject Classification F.4.1 Mathematical Logic, I.2.2 Automatic Programming

Keywords and phrases constraint programming, program synthesis, program analysis

Digital Object Identifier 10.4230/OASIcs.ICLP.2016.13

1 Introduction

A large factor in the success of constraint programming (CP) is compositionality – the flexibility to freely mix and match constraints as needed. However, we are reliant on the underlying solver to provide efficient propagator implementations for the constraints of interest. If we require some problem-specific global constraint we must either design and implement bespoke propagation (and, if we want to use modern lazy clause generation solvers [17], explanation) algorithms or decompose our global constraint into supported primitives.

CP solvers typically support only integer, Boolean and occasionally set variables. Suppose we wish to solve problems over some other algebraic structure – a finite semiring, or the two's complement (or *wrapped*) integers. In this case, we need some way to represent variable domains, encode the semantics of operations, and provide implementations of all constraints of interest.

This can be done by representing variables with existing types and emulating constraints by decomposition into existing primitives. However, a decomposition into existing primitives

* This work was supported by Australian Research Council DE160100568.

```
lex_lt([X|_], [Y|_]) :- X < Y.
lex_lt([X|Xs], [Y|Ys]) :-
  X = Y, lex_lt(Xs, Ys).
```
(a)

```
lex_lt1(X1, X2, Y1, Y2) :- X1 < Y1. % c1
lex_lt1(X1, X2, Y1, Y2) :-          % c2
  X1 = Y1, lex_lt2(X2, Y2).
lex_lt2(X2, Y2) :- X2 < Y2.         % c3
```
(b)

Figure 1 Specification of a strict lexicographic order, and concrete instantiation on arrays of length 2.

```
Prop(lex_lt1)($s_0$):
    $s_1 := \mathcal{R}^{\#}(\texttt{v1 < v3})(s_0)$
    $s_2 := \mathcal{R}^{\#}(\texttt{v1 = v3})(s_0)$
    $s_3 := \textsc{Rename}(s_2, [(\texttt{v1} \leftarrow \texttt{v2}), (\texttt{v2} \leftarrow \texttt{v4})])$
    $s_4 := \mathcal{R}^{\#}(\texttt{v1 < v2})(s_3)$
    $s_5 := \textsc{Splice}(s_2, s_4, [(\texttt{v2} \leftarrow \texttt{v1}), (\texttt{v4} \leftarrow \texttt{v2})])$
    $s_6 := \mathcal{R}^{\#}(\texttt{v1 = v3})(s_5)$
    $s_7 := \textsc{Join}([s_1, s_6])$
    return $s_7$
```

```
Expl(lex_lt1)($[s_1, \ldots, s_7], e_0$):
    $e_1 := E_{\mathcal{R}^{\#}}(\texttt{v1 = v3})(e_0, s_5)$
    $e_2 := \text{E}\textsc{Splice}(e_1, s_2, s_4,$
                $[v1 \leftarrow v2, v1 \leftarrow v4])$
    $e_3 := E_{\mathcal{R}^{\#}}(\texttt{v1 < v2})(e_2[2], s_3)$
    $e_4 := \text{E}\textsc{Rename}(e_3, s_2,$
                $[(\texttt{v2} \leftarrow \texttt{v1}), (\texttt{v4} \leftarrow \texttt{v2})])$
    $e_5 := \textsc{Meet}([e_4, e_2[1]])$
    $e_6 := E_{\mathcal{R}^{\#}}(\texttt{v1 = v3})(e_5, s_0)$
    $e_7 := E_{\mathcal{R}^{\#}}(\texttt{v1 < v3})(e_0, s_0)$
    $e_8 := \textsc{Meet}([e_7, e_6])$
    return $e_8$
```

Figure 2 Propagator and explanation computations derived for the constraint in Figure 1.

may be non-obvious and may be quite large. The decomposition may also be quite unwieldy, as customized decompositions must be provided for all global constraints of interest. Decomposition approaches may also sacrifice efficiency and propagation (and explanation) strength.

A more convenient (for the user) way of handling decomposition approaches is as a model transformer; an extended language is defined, supporting the new types of interest, and are compiled down to the core modelling language. This is the approach adopted for finite-extension [5] and option types [15]. Though convenient for modelling, this requires building a parser and compiler, in addition to the expressive limitations of decompositions.

The alternate approach is to integrate the new variable type natively into the solver. Native integration is typically a very substantial undertaking, so is rarely done.

In this paper, we develop a method for dynamically compiling native-code implementations of propagators from declarative specifications. We then use this to construct global propagators for integer variables with two's complement semantics. We have implemented the described approach as a standalone library, which we then integrated into the `chuffed` [6] lazy clause generation constraint programming solver.

The key insight of our approach is that propagation is a form of abstract interpretation, and hence we can use abstract compilation to generate implementations of propagators

▶ **Example 1.** Consider defining strict lexicographic inequality constraint. A possible checker for this constraint is shown in Figure 1(a). If we wish to instantiate a propagator for a particular constraint, we need to unfold the definition using the structure of the constraint. The unfolded definition for `lex_lt([X1,X2],[Y1,Y2])` is shown in Figure 1(b).

We build a propagator by computing approximations of, for each program point, the set of execution states which are reachable from the initial call, and the subset of those states which could succeed.

Figure 2(a) shows the generated propagator for the constraint `lex_lt([X1,X2],[Y1,Y2])`, we will discuss the detailed meaning later in the paper. Both clauses of `lex_lt1` are reachable from any initial state (s_0). Clause `c1` succeeds iff `v1 < v3` holds, so s_1 approximates its success set. For `c2`, s_2 approximates the set of states which may reach the call to `lex_lt2`, which is mapped onto the formal parameters in s_3. At s_4, we have computed the success set for `lex_lt2`. s_5 and s_6 then compute the corresponding success set for `c2`. s_7 combines the succeeding states for `c1` and `c2`, returning newly pruned variable domains.

The explanation procedure given in Figure 2(b) simply retraces the computations performed by the propagator: for each instruction I with predecessor s_{pre} and necessary condition e_{post}, we compute some e_{pre} such that $s_{pre} \sqsubseteq e_{pre}$, and $I^{\#}(e_{pre}) \sqsubseteq e_{post}$. ◀

The contributions of this paper are as follows:

- A high-level declarative language for specifying constraints
- A procedure for partial evaluation of this high-level language down to a simple constraint logic programming language
- A procedure for deriving abstract propagator and explanation algorithms from these constraint definitions
- A method for synthesizing concrete implementations from these abstract propagators and explainers over novel variable types.

In the following section, we give a brief overview of constraint propagation and abstract interpretation. In Section 3, we describe the correspondence between propagation and static analysis, then in Sections 4 and 5, we show how to use this correspondence to derive propagation and explanation algorithms from implementations of checkers. In Sections 6 and 7, we describe integration of these propagators into a solver, and deriving checkers from a more expressive declarative language. Finally, Section 8 gives an example application of this approach, we describe related work in Section 9 then conclude in Section 10.

2 Preliminaries

In this paper, we restrict ourselves to *finite domain* constraint satisfaction and optimization problems (CSPs and COPs). To avoid confusion, we shall denote logical implication with $\Rightarrow$, and the set of functions with $\rightarrow$.

Propagation-based constraint solving

A CSP is defined by a tuple $(\mathcal{V}, \mathcal{D}, \mathcal{C})$ consisting of a set of variables $\mathcal{V}$ where each variable v may take values from a fixed finite set $\mathcal{D}(v)$, and a set of constraints $\mathcal{C}$. A constraint $c \in \mathcal{C}$ has a *scope*, $\mathit{scope}(c)$ which is a set of variables in $\mathcal{V}$. A constraint c with $\mathit{scope}(c) = \{v_1, \ldots, v_n\}$ is a set of assignments mapping each $v_i \in \mathit{scope}(c)$ to a value in $\mathcal{D}(v_i)$. A *solution* to a CSP is an assignment to each $v \in V$ such that every constraint in C is satisfied. In an abuse of notation we say assignment $\theta \in \mathcal{D}$, if $\theta(v) \in \mathcal{D}(v)$ for all $v \in \mathcal{V}$. A domain is *singleton* if it represents a single assignment, e.g. $|\mathcal{D}(v)| = 1, v \in Vars$. We denote the valuation corresponding to a singleton domain $\mathcal{D}$ as $\theta_{\mathcal{D}}$.

A *propagator* f for a constraint c is a decreasing function, from domains to domains which eliminates values which are not part of any solution to c. A propagator is *correct* if it does not exclude any satisfying assignments – that is, $\theta \in c \wedge \theta \in \mathcal{D} \Rightarrow \theta \in f(c)(\mathcal{D})$. A propagator is *checking* if it is exact for singleton domains, i.e. $f(\mathcal{D}) = \mathcal{D}$ for singleton domains iff $\theta_{\mathcal{D}} \in c$.

In a nogood-learning/lazy clause generation [17] solver, inferences/domain reductions are couched in terms of a formal language of *atomic constraints*, which form a complemented, partially ordered set. A common example in finite-domain solvers is the language of integer bounds and (dis-)equalities: $\{\langle x \leq k \rangle, \langle x > k \rangle, \langle x = k \rangle, \langle x \neq k \rangle\}$, for some variable x and integer constant k. Where a solver integrates nogood-learning/lazy clause generation [17] techniques, each inference *inf* resulting from a propagation $f(c)(D)$ is associated with a corresponding *explanation* E. E is a conjunction of atomic constraints such that $D \Rightarrow E$ and $c \wedge E \Rightarrow inf$. The first condition ensures E is true under the current state, and the second ensures $E \Rightarrow inf$ is globally valid in the problem. When a conflict is detected, inferences participating in the conflict are successively replaced by their explanations to derive a valid nogood which eliminates the current branch of the search tree.

Static program analysis by abstract interpretation

The construction of our propagation and explanation algorithms will be based on the machinery of program analysis.

Abstract interpretation [7] is a framework for inferring information about the behaviour of a program by performing computation on an *abstraction* of the program. The domain $\mathcal{A}$ of program states is replaced by an *abstraction* $\mathcal{A}^\#$. The abstract domain $\mathcal{A}^\#$ forms a lattice, equipped with the usual operators $(\sqsubseteq, \sqcup, \sqcap)$. Correspondence between concrete and abstract states is established by a pair (α, γ) of an abstraction and a concretization function, which form a Galois connection.

Each program instruction $\mathcal{T} : \mathcal{A} \to \mathcal{A}$ is similarly replaced with an abstraction $\mathcal{T}^\# : \mathcal{A}^\# \to \mathcal{A}^\#$. Properties of the program are inferred by executing this abstracted program. Rather than directly executing (possibly infinitely many) control paths, abstract interpreters typically store a single approximation of each program point. Where multiple control paths merge (after conditional statements, at loop heads, or function entries), the incoming abstract states are instead combined: $\varphi_p = \bigsqcup_{q \in \mathsf{preds}(p)} \varphi_q$. Thus, φ_p consists of the strongest property (representable in $\mathcal{A}^\#$) which holds in all predecessor states. This avoids the so-called *path-explosion* problem, but sacrifices precision at join points. Starting with all program points (except the entry) unreachable, state transformers are repeatedly evaluated until a fixpoint is reached. If each transformer is a sound overapproximation, any property which holds at the fixpoint also holds in any reachable concrete state. A typical application of this is to infer numerical properties which must hold at each program point. This is a so-called *forward analysis*, as properties at a given program point are derived from its predecessors.

In a backwards analysis, properties of states are derived from their successors. Numerical backwards analyses are typically rarer than forward analyses. In this case, it is important to distinguish *necessary* preconditions, which must hold in *any* predecessor of a given state, from *sufficient* conditions, which guarantee the given property will hold. Inference of *necessary* conditions have been used to infer preconditions from assertions [9].

To perform a backward analysis in the abstract interpretation framework, we must construct state transformers under/over-approximating the pre-image $\mathcal{T}^-$ of program statements. The analysis proceeds in a similar manner to the forward analysis, but proceeds backwards along the flow of execution, replacing the abstract transformer $\mathcal{T}^\#$ with an abstraction $\mathcal{T}^{-\#}$ of the pre-image.

In this paper, we shall require both forms of analysis; forwards to compute reachable states, and backwards to determine which of these satisfy a constraint.

3 Propagation as Static Analysis

Consider some constraint c, and a checker *program* $CH(c)$ which maps valuations θ over *scope*(c) to true/false such that $CH(c)(\theta) = \mathsf{true} \Leftrightarrow c(\theta)$. The semantics of c is exactly the set of assignments θ such that *executing* $CH(c)$ returns `true` (from a logic programming

$$\begin{array}{rcl}
\tau & \to & \texttt{ident} \mid \texttt{const} \mid \texttt{ident}(\tau^*) \\
\alpha & \to & \texttt{ident} \mid \texttt{const} \mid \texttt{ident}(\alpha^*) \\
 & & \mid \alpha \otimes \alpha \mid \ominus \alpha \\
\mathit{def} & \to & \texttt{ident} := \alpha \\
\mathit{guard} & \to & \alpha \ \mathit{op}\ \alpha,\ \mathit{op} \in \{=, \neq, <, \leq\} \\
\mathit{call} & \to & \texttt{ident}(\tau^*) \\
\mathit{stmt} & \to & \mathit{def} \mid \mathit{guard} \mid \mathit{call} \\
\mathit{clause} & \to & \texttt{ident}(\tau^*)\ \texttt{:-}\ \mathit{stmt}^*
\end{array}$$

(a)

$$\begin{array}{rcl}
\tau & \to & \texttt{ident} \mid \texttt{const} \\
\alpha & \to & \texttt{ident}(\tau^*) \mid \tau \otimes \tau \mid \ominus \tau \\
\mathit{def} & \to & \texttt{ident} := \alpha \\
\mathit{guard} & \to & \tau \ \mathit{op}\ \tau,\ \mathit{op} \in \{=, \neq, <, \leq\} \\
\mathit{call} & \to & \texttt{ident}(\tau^*) \\
\mathit{stmt} & \to & \mathit{def} \mid \mathit{guard} \mid \mathit{call} \\
\mathit{clause} & \to & \texttt{ident}(\tau^*)\ \texttt{:-}\ \mathit{stmt}^*
\end{array}$$

(b)

Figure 3 (a) A LP-style specification language $\mathcal{L}$ for constraints, and (b) The simplified intermediate language $\mathcal{L}^-$, having eliminated complex terms, expressions and recursion. $\otimes$ is a binary infix arithmetic operator, $\ominus$ is a unary arithmetic operator.

perspective, this is the set of *answers* of $CH(c)$). Indeed, any backwards reachability analysis (from `true`) on $CH(c)$ computes a sound approximation of c.

If we interpret the solver's domain store $\mathcal{D}$ as an abstraction of assignments, then a *propagator* $P(c)$ is simply an approximation of the answers of $CH(c)$ restricted to $\gamma(\mathcal{D})$. This is, in fact, equivalent to the *contract precondition inference problem* described in [9] – given a transition system (the program) and initial states (the domain), find the strongest properties which eliminate *only* bad states.

Not every analysis is a valid propagator, however. Propagators will be called on a complete assignment to verify that the assignment is a solution. Each propagator must therefore be *checking* to ensure soundness. This is an extremely uncommon property for a general numeric analysis to have – even from a concrete initial state, precision may be lost at join points, and *widening* [8] discards properties to ensure termination in the presence of unbounded loops.

Nevertheless, this gives us the rough skeleton of an approach: given some specification of a constraint and suitable implementations of abstract operations, we shall generate a native-code implementation of an answer-set analysis for the specification.

But first, we must choose the manner of our specifications.

3.1 Programs as Constraints

While this derivation of propagators from programs is *possible* for arbitrary source languages, in practice we must consider both ease of specification (from the user's perspective) and effectiveness of analysis.

For the remainder of this paper, we consider specifications given in a small (C)LP-style language, $\mathcal{L}$, shown in Figure 3(a). The syntactic category τ denotes the usual language of *terms*. α is the syntactic category of arithmetic expressions, which will be eagerly evaluated during execution. We impose two additional syntactic restrictions. First, free variables cannot be introduced in clause bodies. Second, all (possibly indirect) recursion must be structurally decreasing. That is, if some call `p(X')` is reachable from a call `p(X)`, `X'` must be strictly smaller than `X` with respect to some well-founded measure on term *structure* (independent of the values of variables/constants).

This language $\mathcal{L}$ is reasonably expressive, and provides natural formulations for many global constraints, but does not necessarily seem amenable to numeric analysis.

However, the first condition above ensures that all computations are performed on ground values – this will be needed to ensure the propagators correctly reject invalid total assignments. The second condition similarly guarantees that recursion can be statically expanded. When a constraint is instantiated, we can partially evaluate the specification to construct a much simpler acyclic program consisting only of primitive guards, definitions and calls, which we shall use to derive our propagators.

The reduced language $\mathcal{L}^-$ is shown in Figure 3(b), which eliminates structured terms, complex expressions and all functions (except primitive operators and guards).

4 Constructing propagators from programs

Given a program in the intermediate language $\mathcal{L}^-$ described in Section 3, we must construct a program which, for a given input domain, computes an overapproximation of the corresponding concrete inputs which succeed. To do so, we first construct an intermediate representation of the computations performed by the propagator.

Propagator operations

The instructions used in the constructed propagators: postcondition $\mathcal{T}^{\#}(\texttt{stmt})(q)$, precondition $\mathcal{T}^{-\#}(\texttt{stmt})(q)$, relation $\mathcal{R}^{\#}(\texttt{rel})(q)$, disjunction $\textsc{Join}([q_1, \ldots, q_n])$, conjunction $\textsc{Meet}([q_1, \ldots, q_n])$, projection $\textsc{Rename}(q, [y_1 \leftarrow x_1, \ldots, y_1 \leftarrow x_n])$, and partial update $\textsc{Splice}(q, q', [y_1 \leftarrow x_1, \ldots, y_n \leftarrow x_n])$. Each operation computes an approximation of execution states from one or more previous states. $\textsc{Join}$ and $\textsc{Meet}$ respectively compute the least upper bound ($\sqcup$) and greatest lower bound ($\sqcap$) of abstract states under $\mathcal{A}^{\#}$. $\mathcal{T}^{\#}$ and $\mathcal{T}^{-\#}$ are respectively post- and precondition transformers for function applications, and $\mathcal{R}^{\#}$ applies a relation to an existing state. The remaining operations, are used in dealing with predicate calls. $\textsc{Rename}$ maps variables at a call site onto the formal parameters of the callee. $\textsc{Splice}$ copies a given state q, but takes the domains of variables $v_1, \ldots$ from some *other* state q'. This is used to weave the results of a call back into state of the caller.

The execution of some clause c operates on an execution environment E mapping names to constants. Each guard evaluates the current context, and execution fails if the constraint is violated. A definition adds a new binding to the current environment. At each call site, we rename the call parameters and execute the predicate with the resulting environment. For predicates, each clause is simply executed in turn under the current environment, until some clause succeeds. If all clauses fail, the predicate likewise fails.

Analysis of c simply mirrors the program execution. From an abstract state Q_c, we compute approximations of the reachable states after executing each statement A_c. After computing the abstract solutions of predicate calls, we apply inverse state transformers to determine which initial environments correspond to the solutions. For predicates, the analysis is straightforward: compute the solution sets $[A_{c_1}, \ldots, A_{c_k}]$ for each clause $[c_1, \ldots, c_k]$, and compute the abstract join of these, so $A_p = A_{c_1} \sqcup \ldots \sqcup A_{c_k}$. Throughout the analysis, we maintain the property that $Q_i \sqsubseteq A_i$ – every 'solution' is (abstractly) reachable. This is relatively easy to preserve for guards (which are descending) and definitions. As definitions are total functions, executing some definition `x := E` only introduces a new binding `x`. For any state φ, we then have $\exists x.\ \mathcal{T}^{\#}(\texttt{x := E})(\varphi) = \varphi$. Thus, even the trivial pre-image computation $\mathcal{T}^{-\#}(\texttt{x := E})(\varphi) = \exists\ x.\ \varphi$ preserves this invariant. The upshot of this is that we need not explicitly compute $A_i = A_i \sqcap Q_i$, as this is naturally preserved.

We run into some complications at the predicate level, however. As mentioned in Section 2, abstract interpreters perform abstract *join* operations ($\sqcup$) to combine states whenever a program point is reachable along multiple control paths. If `p` is called in several contexts, if we directly retrieve the solutions to `p` at the call-site, we may lose the property that

Query computation	

`p(x1, ...) :- c1; ...; ck`	s = push_state(JOIN(retrieve_callers(p)))
	save_clause(c1, $Q(s \mid$ c1$)$)
	...
	save_clause(c1, $Q(s \mid$ ck$)$)

$Q(s \mid \emptyset)$	s
$Q(s \mid$ `x := E`$, c)$	$Q($push_state$(\mathcal{T}^{\#}($`x := E`$)(s)) \mid c)$
$Q(s \mid$ `x op y`$, c)$	$Q($push_state$(\mathcal{R}^{\#}($`x op y`$)(s)) \mid c)$
$Q(s \mid$ `p(x1, ...)`$, c)$	save_call(p, push_state(RENAME(s, [x1, ...]))); s
Answer computation	
`p(x1, ...) :- c1; ...; ck`	s_1 = A(retrieve_clause(c1) $\mid$ c1) ... s_k = A(retrieve_clause(ck) $\mid$ ck) s = push_state(JOIN)([$s_1, \ldots, s_k$]) save_answer(p, s)
$A(s_0 \mid \emptyset)$	s_0
$A(s_0 \mid$ `x := E`$, c)$	push_state$(\mathcal{T}^{-\#}($`x := E`$)(A(s_0 \mid c)))$
$A(s_0 \mid$ `x op y`$, c)$	push_state$(\mathcal{R}^{\#}($`x op y`$)(A(s_0 \mid c)))$
$A(s_0 \mid$ p$(x_1, \ldots, x_n), c)$	$s_{post} = A(s_0 \mid c)$ s_{proj} = push_state(RENAME(s_{post}, [x1, ...])) s_{ret} = retrieve_answer(p) s_{meet} = push_state(MEET([s_{ret}, s_{proj}]) push_state(SPLICE(s_{post}, s_{meet}, [$x_1, \ldots, x_n$]))

Figure 4 Computing approximations of reachable and satisfying states during checker execution. Reachability computations are performed for predicates in topological order, and answers are computed in the reverse order.

$A_i \sqsubseteq Q_i$. Worse, the loss of precision can interfere with the requirement that the propagator be checking.

▶ **Example 2.** Consider the following program:

```
p(x, y) :- q(x, y).     p(x, y) :- q(y, x).     q(u, v) :- u = v.
```

Consider analysing this program under $\{\mathtt{x} \to 3, \mathtt{y} \to 4\}$. `q` is reachable under two environments: $\{\mathtt{u} \to 3, \mathtt{v} \to 4\}$, and $\{\mathtt{u} \to 4, \mathtt{v} \to 3\}$. Before processing `q`, the calling contexts are combined into $\{\mathtt{u} \to [3,4], \mathtt{v} \to [3,4]\}$. Applying `u = v` here does nothing. Notice that the answer set of `q` is weaker than either call state. When we combine this back into the call site, both calls appear feasible, so we do not detect failure. ◀

To preserve the descending property, we must instead compute the meet of the calling state with the answer set of the predicate. To ensure the propagator is checking, we exploit the fact that *bindings* are functionally defined. We transform the *checker* to ensure all calls to a predicate to have identical argument definitions (in terms of input variables). We can perform this step by traversing the program tracking the definition of each variable, and renaming apart predicate calls with different definitions. In the case of Example 2, `q` becomes two separate predicates `q1` and `q2`.

The algorithm for constructing a propagator from a checker is given in Figure 4. push_state adds a new state to the propagator, and returns the new state's identifier. In addition to the generated instructions, we also need to keep track of three sets of states: states which call some predicate `p`, the final state of each clause `c`, and the answer set of each predicate `p` – we

Instruction	Generated code	Resulting state
$\mathcal{T}^{\#}(\texttt{z := f(x,y)})(c, \sigma)$	$v' := \mathit{emit}(\mathcal{T}^{\#}(f)(\sigma(x), \sigma(y)))$	$(c, \sigma[z \mapsto v'])$
$\mathcal{T}^{-\#}(\texttt{z := f(x,y)})(c, \sigma)$	$b, u', v' :=$ $\quad \mathit{emit}(\mathcal{T}^{-\#}(f)(\sigma(z), \sigma(x), \sigma(y)))$ $c' := c \wedge b$	$(c, \sigma[x \mapsto u', y \mapsto v'] \setminus \{z\})$
$\mathcal{R}^{\#}(\texttt{x rel y})$	$b, u', v' := \mathit{emit}(\mathcal{R}^{\#}(\texttt{rel}, \sigma(x), \sigma(y)))$ $c' := c \wedge b$	$(c', \sigma[x \mapsto u', y \mapsto v'])$
$\textsc{Meet}([(c_A, \sigma_A), (c_B, \sigma_B)])$	$b_x, v_x := \sigma_A(x) \sqcap \sigma_B(x)$ **for** $x \in \sigma_A$ $c' := c_A \wedge c_B \wedge \bigwedge b_x$ $\sigma' := \{x \mapsto v_x \mid x \in \sigma_A\}$	(c', σ')
$\textsc{Join}([(c_A, \sigma_A), (c_B, \sigma_B)])$	$v_x := \begin{pmatrix} \sigma_B(x) \textbf{ if } \neg c_A \\ \sigma_A(x) \textbf{ if } \neg c_B \\ \sigma_A(x) \sqcup \sigma_B(x) \textbf{ else} \end{pmatrix}$ **for** $x \in \sigma_A$ $c' := c_A \vee c_B$ $\sigma' := \{x \mapsto v_x \mid x \in \sigma_A\}$	(c', σ')
$\textsc{Rename}((c, \sigma),\ M)$		$(c, \{y \mapsto \sigma(x)$ $\mid (y \leftarrow x) \in M\})$
$\textsc{Splice}((c, \sigma), (c_{sp}, \sigma_{sp}), \mathrm{M})$		$(c_{sp}, \sigma[y \mapsto \sigma_{sp}(x)$ $\mid (y \leftarrow x) \in M])$

Figure 5 Constructing concrete code implementing an abstract propagator. *emit*() denotes dispatch to an externally-provided transfer function.

use the corresponding save/retrieve functions to keep track of these sets. $Q(s \mid c)$ constructs the computation of final reachable states of clause c starting from state s, and records the context of any predicate calls made.

4.1 Generating Propagator Implementations

The propagator descriptions described above make no assumptions as to the concrete representation of propagator states, other than being elements of a lattice with associated state transformers. For the remainder of the paper, we shall assume states are abstracted by a non-relational ('independent attribute') domain. The concrete representation of a state is then a tuple (c, σ), where c is a Boolean flag indicating whether the state is feasible, and σ is a mapping from variables to the corresponding domain representation.

Under this non-relational representation, generating concrete implementations of these propagators is relatively straightforward. Propagator computation consists of three phases: a prologue, where domain representations are extracted from solver variables, the propagator body, and an epilogue, where we compare the initial and revised domains for each variable, and post any updated domains to the solver. The propagator body simply computes values for the sequence of states appearing in the abstract propagator we constructed. State transformers for operations on individual domain approximations must be externally provided, which we then lift to operations on propagator states.

Rules for state computation are given in Figure 5. We assume machine code is written to a global buffer. In the generated code, Rename and Splice become no-ops; they simply re-bind existing values to new names. $\mathcal{T}^{\#}$, $\mathcal{T}^{-\#}$ and $\mathcal{R}^{\#}$ are similarly straightforward, computing new values for those variables touched by the instruction (using the externally provided implementations), and updating the corresponding bindings. Here *emit* denotes calls to an external code generator, which emits instructions implementing the specified primitive, and returns the location of the resulting values. Join and Meet implement the usual lifting of $\sqcup$ and $\sqcap$ operations to the Cartesian product. We show here code only for binary functions, as well as meet and join; the n-ary operators follow the same pattern.

5 Inferring Explanations

In nogood-learning solvers, we have an additional complication: explanations. Assume a propagation step $f(c)(D)$ infers the atomic constraint at. During conflict analysis, we will need to replace at with some set of antecedents $l_1, \ldots, l_k$ such that $D \Rightarrow l_1 \wedge \ldots \wedge l_k$, and $c \wedge l_1 \wedge \ldots \wedge l_k \Rightarrow at$.

When it comes to dealing with novel variable types, we have two problems: first, how to represent atomic constraints in general, and how to infer explanations for arbitrary constraints, while avoiding imposing too heavy a burden on the solver author.

To this end, we make a pair of perhaps trivial observations. First, a variable domain is always expressible as a conjunction of atomic constraints. Second, the generated propagators always admit some valid explanation consisting of a conjunction of variable domains. This hints at a possible approach - collect explanations using the same domain representation as the propagation algorithm, and have the solver extract the corresponding atomic constraints before returning.

Note that we can't absolve the solver developer from integrating atomic constraints into the solver core; handling propagation, implication and resolution of atomic constraints is still something that needs to be done.[1] However, with this approach they do not need to somehow communicate the semantics of atoms to the synthesis engine, nor provide bindings for the full set of operations (subsumption, disjunction, etc.) on atoms.

In terms of generating the explanation itself, the trivial explanation is always sound, relatively efficient to construct and requires no additional information from the solver, but is of limited value: $\bigwedge\{\mathcal{D}(v) \mid v \in \mathit{scope}(c)\} \Rightarrow at$. We can do much better by taking the correspondence between static analysis and propagation one step further, and observe that explanation is just an analysis of $P(c)$. Recall the computation of $P(c)$, illustrated to the right. From some initial state D, we apply a sequence of state transformers $[T_1, \ldots, T_n]$ computing states $[D_1, \ldots, D_n]$, D_n being the approximate solution set.

For an inference D_{inf}, we wish to find D_{expl} such that $D \sqsubseteq D_{\mathit{expl}}$, and $P(c)(D_{\mathit{expl}}) \sqsubseteq D_{\mathit{inf}}$. We can compute such a state by pushing the condition backwards along the computation of $P(c)$. We first find some state E_{n-1}, with $D_{n-1} \sqsubseteq E_{n-1}$ and $T_n(E_{n-1}) \sqsubseteq D_{\mathit{inf}}$. We continue in this manner, at each step computing E_{i-1} from D_{i-1} and E_i. The final state, E_0 is thus guaranteed to be a valid explanation.

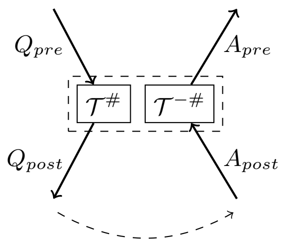


Figure 6 Flow of computation in $P(c)$.

[1] Though the developer may be able to re-use atoms for existing types – encoding option types with pairs of integers [15], or bit-vectors by tuples of Booleans.

$\mathsf{explain}(P, e_0)$	$\mathsf{store\_use}(\mathsf{final\_state}(P), e_0)$ $Ex(P)$
$Ex(s : I, P)$	$Ex(P)$ $e_{post} = \mathsf{push\_state}(\textsc{Meet}(\mathsf{retrieve\_uses}(\mathrm{s})))$ $Ex_I(e_{post}, I)$
$Ex_I(e, \mathcal{T}^{\#}(\mathtt{stmt})(q))$	$\mathsf{store\_use}(q, \mathsf{push\_state}(E_{\mathcal{T}^{\#}}(\mathtt{stmt})(e, q)))$
$Ex_I(e, \mathcal{T}^{-\#}(\mathtt{stmt})(q))$	$\mathsf{store\_use}(q, \mathsf{push\_state}(E_{\mathcal{T}^{-\#}}(\mathtt{stmt})(e, q)))$
$Ex_I(e, \mathcal{R}^{\#}(\mathtt{rel})(q))$	$\mathsf{store\_use}(q, \mathsf{push\_state}(E_{\mathcal{T}^{-\#}}(\mathtt{stmt})(e, q)))$
$Ex_I(e, \textsc{Join}([q_1, \ldots, q_n]))$	$\mathsf{store\_use}(q_1, e); \ldots; \mathsf{store\_use}(q_n, e)$
$Ex_I(e, \textsc{Meet}([q_1, \ldots, q_n]))$	$e' = \mathsf{push\_state}(\text{E}\textsc{Meet}(e, [q_1, \ldots, q_n]));$ $\mathsf{store\_use}(q_1, e'[1])\ ; \ldots; \mathsf{store\_use}(q_n, e'[n]);$
$Ex_I(e, \textsc{Rename}(q, [\mathtt{x_1}, \ldots, \mathtt{x_n}]))$	$\mathsf{store\_use}(q, \mathsf{push\_state}(\text{E}\textsc{Rename}(e, q, [\mathtt{x_1}, \ldots, \mathtt{x_n}])$
$Ex_I(e, \textsc{Splice}(q, q', [\mathtt{x_1}, \ldots, \mathtt{x_n}]))$	$e' = \mathsf{push\_state}(\text{E}\textsc{Splice}(e, q, q', [\mathtt{x_1}, \ldots, \mathtt{x_n}]))$ $\mathsf{store\_use}(q, e'[1]); \mathsf{store\_use}(q', e'[2])$

Figure 7 Constructing an explanation from a propagator. The algorithm walks backwards along the computation, computing a sufficient postcondition for each instruction.

Just as we constructed a propagator from a checker in Section 4, we now define a corresponding translation scheme from propagators to explainers. It is assumed that the explanation algorithm is executed after the propagator, and has access to all the intermediate stages of the propagator. The primitive operations performed during explanation are $E_{\mathcal{T}^{\#}}(\mathtt{stmt})(e, q)$, E$\textsc{Meet}(e, [q_1, \ldots, q_n])$, $E_{\mathcal{T}^{-\#}}(\mathtt{stmt})(e, q)$, E$\textsc{Rename}(e, q, [x_1, \ldots, x_n])$, $E_{\mathcal{R}^{\#}}(\mathtt{rel})(e, q)$, E$\textsc{Splice}(e, q, q', [x_1, \ldots, x_n])$, and $\textsc{Meet}([e_1, \ldots, e_n])$. $\textsc{Meet}$, as in the propagator case, simply conjoins a set of preconditions. All other operations simply push some postcondition back to the instruction's predecessor states (essentially computing an *interpolant* [10]).

The algorithm for translating a propagator into a corresponding 'explainer' is given in Figure 7. The explanation procedure runs backwards along the computations performed by the propagator, constructing a sufficient postcondition for each state of propagator state. In the propagator a given state may be used by multiple successors, particularly states corresponding to predicate heads and call sites. During explanation, each use of that state may result in a different postcondition. We use store_use to record the individual postconditions and conjoin them (using $\textsc{Meet}$) to construct an overall postcondition for the state before extrapolating back to the state's predecessors.

We have several choices in how this abstract explanation algorithm is embodied and used. A single run of the propagator may change domains of several variables. We may either generate a separate explanation for each domain change (which requires running the explanation algorithm several times), or construct a common explanation for all changes (which is cheaper, but yields less general explanations).

Another choice is how to represent preconditions. The most precise approach is to follow the same pattern as for propagation – maintain a full propagator state as the precondition and require externally provided *explanation transformers* for the necessary operations ($E_{\mathcal{T}^{\#}}$ and $E_{\mathcal{T}^{-\#}}$ for functions, $E_{\mathcal{R}^{\#}}$ for guards, and $E_{\sqcap}$ for meet), which turn a postcondition and incoming domains into a set of preconditions. Designing correct, efficient and precise implementations of explanation transformers is challenging, complicated by the fact that we need to deal with variables which are unconstrained in the postcondition (by either having an explicit $\top$ value, a Boolean flag, or pre-computing initial domains for each propagator state).

Domain representation: `t`, **Variable:** `v`, **Atomic constraint:** `a`			
Domain operations		Transformers	
equality	$(\mathtt{t},\mathtt{t}) \to \mathtt{bool}$	$\mathcal{T}^{\#}(\mathtt{fun})$	$\mathtt{list}(\mathtt{t}) \to \mathtt{t}$
conjunction	$(\mathtt{t},\mathtt{t}) \to (\mathtt{bool},\mathtt{t})$	$\mathcal{T}^{-\#}(\mathtt{fun})$	$(\mathtt{t},\mathtt{list}(\mathtt{t})) \to (\mathtt{bool},\mathtt{list}(\mathtt{t}))$
disjunction	$(\mathtt{t},\mathtt{t}) \to \mathtt{t}$	$\mathcal{T}^{\#}(\mathtt{rel})$	$(\mathtt{t},\mathtt{t}) \to (\mathtt{bool},(\mathtt{t},\mathtt{t}))$
Variable hooks		Explanation hooks	
GET-DOMAIN	$\mathtt{v} \to \mathtt{d}$	TO-ATOMS	$(\mathtt{v},\mathtt{d}) \to \mathtt{list}(\mathtt{a})$
SET-DOMAIN	$(\mathtt{v},\mathtt{d}) \to \mathtt{bool}$	SET-DOMAIN$_{expl}$	$(\mathtt{v},\mathtt{d},\mathtt{list}(\mathtt{a})) \to \mathtt{bool}$
		SET-CONFLICT	$\mathtt{list}(\mathtt{a}) \to \mathtt{unit}$

Figure 8 Operations that must be provided for domains, functions and relations in order to execute propagators.

We can instead construct a data-flow based explanation procedure. We track which values could have contributed to the inference of interest, and translate the corresponding domains to atomic constraints as the explanation. For the flow-based explanation, we represent the pre/post-condition as a pair (e_c, ν), where e_c indicates whether we must explain failure, and ν is a mapping from names to Booleans indicating whether the corresponding variable is relevant to the inference. Transformers for this analysis are straightforward. For example, $E_{\mathcal{T}^{\#}}(\mathtt{z\ =\ f(x,\ y)})((e_c,\nu),(c,\sigma))$ returns $(e_c, \nu[x \mapsto \nu(x) \vee \nu(z), y \mapsto \nu(y) \vee \nu(z)] \setminus \{z\})$.

6 Filling in the gaps

For the propagator construction of Section 4, we are missing implementations of three critical elements: the lattice of domain abstractions, state transformers for function and relation symbols, and hooks to communicate with the solver.

The operations needed to implement propagators are given in Figure 8. These fall into two classes: operations on domain abstractions, and communication between the propagators and the underlying solver. A pleasant outcome of this separation is that domain operations are entirely decoupled from the underlying solver – once lattice operations and transformers are defined for a given domain, they may be re-used in other solvers. The only operations which must be defined per solver *and* per variable kind is the extraction and update of domains.

For a classical CP solver, these are the only operations which must be defined. For lazy clause generation, the solver must also provide operations for dealing with atomic constraints. From the propagators' perspective, atomic constraints are entirely opaque. The solver specifies the (maximum) atom size, and each variable indicates the maximum number of atoms required to explain its domain. Before setting domains, we allocate a buffer large enough to fit the largest possible explanation. TO-ATOMS writes atomic constraints to this buffer, returning the end of the explanation so far. SET-CONFLICT and SET-DOMAIN$_{expl}$ will then retrieve the explanation from this buffer.

7 Instantiating Constraints from Specifications

We now return to the problem of transforming high-level specifications into intermediate form. The process must make two transformations: eliminating nested arithmetic expressions, and unfolding predicate bodies. The first is done in the usual manner, introducing fresh variables for sub-terms.

The second amounts to partially evaluating the logic program under the given instantiation. When evaluating a predicate call $\mathtt{p}(T)$ (where $T = [\mathtt{t}_1, \ldots, \mathtt{t}_n]$), we compute 'canonical arguments' T' by replacing each variable appearing in T with the index of its first occurrence, and (recursively) instantiate a copy of `p` with these T'. The instantiation of `p` is a predicate taking one argument for each variable appearing in T'.

Pattern-matching in clause heads is statically resolved. Clauses with non-matching heads or type-mismatches in expressions (e.g. arithmetic expressions instantiated on compound terms) are discarded, as are those containing calls to a predicate with no feasible clauses. The requirement that recursive calls be *structurally decreasing* is so that we may be sure the instantiation process terminates.

▶ **Example 3.** Recall the specification of `lex_lt`, given in Figure 1. Consider instantiating the constraint `lex_lt`$([\mathtt{X}, \mathtt{Y}], [\mathtt{Z}, \mathtt{Z}])$. Numbering variables in order of occurrence, we obtain the canonical arguments $[[\mathtt{V}_1, \mathtt{V}_2], [\mathtt{V}_3, \mathtt{V}_3]]$. Instantiating the first clause body, we get $\mathtt{V}_1 < \mathtt{V}_3$. In the second clause body, we see a recursive call to `lex_lt`, with (instantiated) arguments $[[\mathtt{V}_2], [\mathtt{V}_3]]$.

Instantiating `lex_lt`$([\mathtt{V}_1], [\mathtt{V}_2])$, we again obtain $\mathtt{V}_1 < \mathtt{V}_2$ for the first clause. In the second clause, we reach a recursive call `lex_lt`$([], [])$. Both clauses of `lex_lt` fail due to pattern matching, which causes the second clause of `lex_lt`$([\mathtt{V}_1], [\mathtt{V}_2])$ to fail. This gives us the instantiated checker:

```
lex_lt3(V1,V2,V3) :- V1 < V3. lex_lt3(V1,V2,V3) :- V1 = V3, lex_lt4(V2,V3).
lex_lt4(V1,V2) :- V1 < V2.
```
◀

8 Experimental Evaluation

We have implemented a prototype library `creidhne`[2] implementing this method. The library provides a C++ interface, but is implemented in OCaml using the LLVM compiler framework [14] for code generation.

Two's complement arithmetic

Integer arithmetic in CP operates on a subset of $\mathbb{Z}$. In some applications, particularly model checking, we instead wish to reason under machine arithmetic – the *fixed-width* or *wrapped* integers, which are not typically supported by CP solvers. This domain has received some attention [1, 13], but is not a common inclusion in CP or LCG solvers.

We used `creidhne` to integrate (signed) wrapped integers into `chuffed`,[3] a lazy clause generation CP solver. No modifications were needed to the underlying solver engine. Wrapped integers variables were represented internally using existing integer variables, and existing atomic constraints re-used. Connecting `chuffed` with `creidhne` totalled 300 lines of C++, plus minor changes to the FlatZinc [2] parser to allow string literals in annotations. The lattice operations and state transformers were implemented as code emitters for LLVM, totalling around 350 lines of OCaml.

We tested the synthesized propagators on some error-localization problems using reified 8-bit machine arithmetic. The synthesized propagators appear competitive with the native decompositions. For programmed search, the absence of introduced variables helps noticeably. Note that 32-bit wrapped integers could not be implemented by decomposition.

[2] Available at `http://bitbucket.org/gkgange/creidhne`.
[3] `http://github.com/geoffchu/chuffed`

Table 1 Average time (in seconds) and backtracks on small error-localization problems, using programmed (seq) or activity-driven (act) search. # gives number of instances.

	#	`native`(seq)	`native`(act)	`creidhne`(seq)	`creidhne`(act)
SUMSQUARES	19	16.65 / 193800	0.51 / 4712	0.80 / 21190	0.41 / 4651
TRITYP	100	0.17 / 2707	0.05 / 384	0.10 / 286	0.61 / 42880

9 Related Work

The burden of formulating and implementing propagation algorithms is well recognised, and a number of intermediate languages and compilation approaches have been proposed, although none consider generating explanations.

The approach of [3] represents constraint checkers as finite-state automata augmented with a finite set of counters. A constraint is instantiated by decomposing the automaton into a conjunction of primitive constraints. In [4] constraints were formulated as predicates denoting Boolean formulae of primitive constraints. Inference rules were derived for Boolean operators to determine which (lazily instantiated) primitive constraints could potentially propagate during search.

In [16], the authors propose a propagator specification language for global constraints based on an extension of *indexicals* [19], and define a compiler backend for each supported solver. The indexical-based specifications allow more finer control of propagation, but the universe of types is fixed and propagation rules must be specified by hand.

The method of [12] directly shares our objective of inferring efficient imperative propagators from arbitrary constraints. This approach eagerly pre-computes the result of enforcing domain consistency for all values in the powerset of $\mathcal{D}(c)$, and compiles a lookup table from the results. This computes extremely efficient (and domain-consistent) propagators, but is feasible only for constraints with small domains – the pre-computation time and worst-case memory requirements are $O(|\mathcal{P}D|^{|vars(c)|})$. For global constraints over integer variables with large domains, this approach is impractical.

Several existing works have applied ideas from abstract interpretation to constraint programming. The observation of constraint propagation as a fixpoint procedure was used in [18] to design an abstract interpretation based constraint solver for real variables. In [11], techniques from abstract interpretation were used to support constraints involving loops in a CLP formalism. These constraints were propagated by computing an approximation of the loop under the polyhedra abstract domain, then projecting back onto the problem variables.

10 Conclusion and Further Work

We have presented a system for synthesizing propagators (with explanation) over novel variable types from declarative specifications, and illustrated its effectiveness. There are numerous potential extensions, both in terms of the specification and the synthesis. These include adding support for partial functions, exploiting opportunities for more efficient propagation, and relaxing the restriction on unbounded recursion.

References

1 Sébastien Bardin, Philippe Herrmann, and Florian Perroud. An Alternative to SAT-Based Approaches for Bit-Vectors. In *Tools and Algorithms for the Construction and Analysis of Systems*, number 6015 in LNCS, pages 84–98. Springer Berlin Heidelberg, March 2010.

2 Ralph Becket. Specification of FlatZinc. [Online, accessed 3 March 2015], 2012. `http://www.minizinc.org/downloads/doc-1.6/flatzinc-spec.pdf`.

3 N. Beldiceanu, M. Carlsson, and T. Petit. Deriving filtering algorithms from constraint checkers. In *CP 2014*, volume 3258, pages 107–122, 2004. `doi:10.1007/978-3-540-30201-8_11`.

4 Sebastian Brand and Roland H. C. Yap. Towards 'propagation = logic + control'. In *ICLP 2006*, volume 4079, pages 102–116, 2006. `doi:10.1007/11799573_10`.

5 Rafael Caballero, Peter J. Stuckey, and Antonio Tenorio-Fornes. Two type extensions for the constraint modelling language MiniZinc. *Science of Computer Programming*, 111:156–189, 2016.

6 Geoffrey Chu. *Improving Combinatorial Optimization.* PhD thesis, Department of Computing and Information Systems, University of Melbourne, 2011.

7 Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL'77*, pages 238–252, New York, NY, USA, 1977. `doi:10.1145/512950.512973`.

8 Patrick Cousot and Radhia Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In *PLILP'92*, volume 631, pages 269–295, 1992.

9 Patrick Cousot, Radhia Cousot, and Francesco Logozzo. Precondition Inference from Intermittent Assertions and Application to Contracts on Collections. In *VMCAI 2011*, number 6538 in LNCS, pages 150–168. Springer Berlin Heidelberg, January 2011.

10 William Craig. Three uses of the herbrand-gentzen theorem in relating model theory and proof theory. *Journal of Symbolic Logic*, 22:269–285, 9 1957. `doi:10.2307/2963594`.

11 Tristan Denmat, Arnaud Gotlieb, and Mireille Ducassé. An abstract interpretation based combinator for modelling while loops in constraint programming. In *CP 2013*, volume 4741, pages 241–255, 2007. `doi:10.1007/978-3-540-74970-7_19`.

12 Ian P. Gent, Christopher Jefferson, Steve Linton, Ian Miguel, and Peter Nightingale. Generating custom propagators for arbitrary constraints. *Artif. Intell.*, 211:1–33, 2014. `doi:10.1016/j.artint.2014.03.001`.

13 Arnaud Gotlieb, Michel Leconte, and Bruno Marre. Constraint solving on modular integers. In *ModRef Workshop, associated to CP'2010*, September 2010.

14 C. Lattner and V. Adve. LLVM: a compilation framework for lifelong program analysis transformation. In *CGO 2004*, pages 75–86, March 2004. `doi:10.1109/CGO.2004.1281665`.

15 Christopher Mears, Andreas Schutt, Peter J. Stuckey, Guido Tack, Kim Marriott, and Mark Wallace. Modelling with option types in minizinc. In *CPAIOR 2014*, number 8451 in LNCS, pages 88–103. Springer, 2014. `doi:10.1007/978-3-319-07046-9_7`.

16 Jean-Noël Monette, Pierre Flener, and Justin Pearson. Towards solver-independent propagators. In *CP 2012*, volume 7514, pages 544–560, 2012. `doi:10.1007/978-3-642-33558-7_40`.

17 O. Ohrimenko, P.J. Stuckey, and M. Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, 2009.

18 Marie Pelleau, Antoine Miné, Charlotte Truchet, and Frédéric Benhamou. A Constraint Solver Based on Abstract Domains. In *VMCAI 2013*, number 7737 in LNCS, pages 434–454. Springer Berlin Heidelberg, January 2013.

19 P. Van Hentenryck, Vijay Saraswat, and Yves Deville. Constraint processing in cc(FD). Technical report, Computer Science Department, Brown University, 1992.

A Compositional Typed Higher-Order Logic with Definitions[*]

Ingmar Dasseville[1], Matthias van der Hallen[†1], Bart Bogaerts[‡3], Gerda Janssens[1], and Marc Denecker[1]

1 **KU Leuven – University of Leuven, Celestijnenlaan 200A, Leuven, Belgium**
`ingmar.dasseville@cs.kuleuven.be`

2 **KU Leuven – University of Leuven, Celestijnenlaan 200A, Leuven, Belgium**
`matthias.vanderhallen@cs.kuleuven.be`

3 **Helsinki Institute for Information Technology HIIT, Aalto University, Aalto, Finland; and**
KU Leuven – University of Leuven, Celestijnenlaan 200A, Leuven, Belgium
`bart.bogaerts@cs.kuleuven.be`

4 **KU Leuven – University of Leuven, Celestijnenlaan 200A, Leuven, Belgium**
`gerda.janssens@cs.kuleuven.be`

5 **KU Leuven – University of Leuven, Celestijnenlaan 200A, Leuven, Belgium**
`marc.denecker@cs.kuleuven.be`

Abstract

Expressive KR languages are built by integrating different language constructs, or extending a language with new language constructs. This process is difficult if non-truth-functional or non-monotonic constructs are involved. What is needed is a compositional principle.

This paper presents a compositional principle for defining logics by modular composition of logical constructs, and applies it to build a higher order logic integrating typed lambda calculus and rule sets under a well-founded or stable semantics. Logical constructs are formalized as triples of a syntactical rule, a semantical rule, and a typing rule. The paper describes how syntax, typing and semantics of the logic are composed from the set of its language constructs. The base semantical concept is the *infon*: mappings from structures to values in these structures. Semantical operators of language constructs operate on infons and allow to construct the infons of compound expressions from the infons of its subexpressions. This conforms to Frege's principle of compositionality.

1998 ACM Subject Classification I.2.4 Knowledge Representation Formalisms and Methods

Keywords and phrases Logic, Semantics, Compositionality

Digital Object Identifier 10.4230/OASIcs.ICLP.2016.14

1 Introduction

Expressive knowledge representation languages consist of many different language constructs. New KR languages are often built by adding new (possibly nestable) language constructs to

[*] This research was supported by the project GOA 13/010 Research Fund KU Leuven and projects G.0489.10, G.0357.12 and G.0922.13 of FWO (Research Foundation – Flanders).

[†] Matthias van der Hallen is supported by a Ph.D. fellowship from the Research Foundation – Flanders (FWO – Vlaanderen).

[‡] Bart Bogaerts is supported by the Finnish Center of Excellence in Computational Inference Research (COIN) funded by the Academy of Finland (grant #251170).

existing logics. Principled compositional methods are desired that allow to construct logics from language constructs, or incrementally extend an existing logic with a new construct, while preserving the meaning of the remaining language constructs. This is known as Frege's compositionality principle.

In classical monotone logics it is common practice to extend a logic with new language constructs or connectives by specifying an additional pair of a syntactical and semantical rule. E.g., we can add a cardinality construct to classical first order logic (with finite structures) by defining:

- syntactical rule: $\#(\{x : \varphi\})$ is a (numerical) term if x is a variable and φ a formula;
- semantical rule: $(\#(\{x : \varphi\}))^I = \#(\{d \in \mathcal{D}^I \mid I[x : d] \models \varphi\})$, the cardinality of the set of domain elements that correspond to the set expression. Here, $\mathcal{D}^I$ is the domain of I.

The ease and elegance of this is beautiful. In the context of nonmonotonic languages such as logic programming and extensions such as answer set programming [18, 17, 20] and the logic FO(ID) (classical logic with inductive definitions) [7], the situation is considerably more complex. For example, adding aggregates to these logics required, and still requires a serious effort [15, 25, 10, 22, 21, 9, 11] and resulted in a great diversity of logics.

In this paper, we propose a compositional principle to build logics, and apply it to build a logic $\mathbb{L}$ integrating typed higher order lambda calculus with definitions represented as rule sets under well-founded semantics. The two main contributions of this work are:

- It introduces a compositional principle to build and integrate logics and puts it to the test: by building an expressive logic including rule sets, with aggregates, lambda expressions, higher order rules, rule sets to express definitions by monotone, well-founded and iterated induction, definitions nested in rules, ... The semantical basis is the concept of *infon* which provides a semantical abstraction of the meaning of expressions and is related to intensional objects in intensional logics [14] .
- The logic itself brings together the logics of logic programming and descendants such as answer set programming and FO(ID), and the logic of typed lambda calculus which has become the foundational framework for formal specification languages and functional programming. We illustrate the application of the resulting logic to build simple and elegant theories that express complex knowledge.

2 Related Work

2.1 Logics

Our paper on templates [5] introduced a simpler version of the framework from the current paper, using informal notions. There, the framework was used to construct a logic permitting inductive definitions within the body of other inductive definitions. In that logic, *templates* are (possibly inductive) second order definitions that allow nesting inductive definitions; this nesting is required to build, for instance, templates defining one predicate parameter as the transitive closure of another parameter. In this paper, we present the framework with a more formal basis, using the concept of *infons* as the mathematical object corresponding to the semantics of a language construct, and identify the notion of *Frege's principle of compositionality* as the underlying goal of the framework.

This paper explicitly allows the construction of *higher-order* logics. In the context of meta-programming [1], some logics with higher-order syntax already exist. One such example is HiLog [4], which combines a higher-order syntax with first-order semantics. HiLogs main motivation for this is to introduce a useful degree of higher order while maintaining decidability of the deduction inference. Another example is λprolog [19], which extends

Prolog with (among others) higher-order functions, λ-terms, higher-order unification and polymorphic types. To achieve this, λprolog extends the classical first-order theory of Horn-clauses to the intuitionistic higher-order theory of *Hereditary Harrop* formulas [16].

The algebra of modular system (AMS) [23, 24] is a framework in which arbitrary logics with a model semantics can be combined. The difference with our work is that in AMS, connectives from the different logics cannot be combined arbitrarily. Instead, there is a fixed set of connectives (a "master" logic) that can be used to combine expressions from different logics. Compared to our logic, this has advantages and disadvantages. One advantage is that AMS only requires a two-valued semantics (an infon) to be specified for a given logic, making it more easily applicable to a wide range of logics. A disadvantage is that it does not allow for interactions between the different connectives.

2.2 Infons

The concept of infon in the sense used in this paper is related to intensional objects in Montague's intensional logic [14]. Intensional logic studies the dichotomy between the designation and the meaning of expressions. Intensional objects are represented by lambda expressions and model functions from states to objects similar to our infons. The term "infon" was used by other authors in other areas. In situation semantics [2], infons intuitively represent "quantums of information" [8]. Although such an infon has a different mathematical form than an infon in our theory, it determines a characteristic function from *situations* (which are approximate representations of states, similar to approximate structures) to true, false (or undetermined), which intuitively corresponds to an infon. Situation semantics, the semantics supported by situation theory, provides a foundation for reasoning about real world situations and the derivations made by common sense. In [13], infons are "statements viewed as containers of information" and an (intuitionistic) logic of infons is built for the specific purpose of modelling distributed knowledge authorization.

3 Preliminaries

3.1 Cartesian product, powerset, product, pointwise extension and lifting

The *powerset* operator $\mathcal{P}(\cdot)$ maps a set X to its powerset $\mathcal{P}(X)$. The *power* operator $(\cdot)^{(\cdot)}$ maps pairs (I, Y) of sets to the set Y^I of all functions with domain I and co-domain Y. We denote the function with domain D and co-domain C that maps elements $x \in D$ to the value of a mathematical expression $exp[x]$ in variable x as $\lambda : D \to C : x \mapsto exp[x]$ (using λ as the anonymous function symbol as in lambda calculus). Or, if the co-domain is clear from the context, as $\lambda x \in D : exp[x]$. When $exp[x]$ is Boolean expression, this is also denoted as a set comprehension $\{x \in D \mid exp[x]\}$.

We define the set of truth values $\mathcal{T}wo = \{\mathbf{f}, \mathbf{t}\}$; here $\mathbf{t}$ stands for "true" and $\mathbf{f}$ for "false". For any X, $\mathcal{P}(X)$ is isomorphic to $\mathcal{T}wo^X$, using the mapping from a set to its characteristic function. In the rest of the paper, we will identify $\mathcal{P}(\cdot)$ with $\mathcal{T}wo^{(\cdot)}$.

We frequently use $\langle x_i \rangle_{i \in I}$ to denote the function $\lambda : I \to \{x_i \mid i \in I\} : i \to x_i$. We call this an indexed set (with index set I). Let $\langle V_i \rangle_{i \in I}$ be an indexed set of sets, i.e., each V_i is a set. Its *product set*, denoted $\times_{i \in I} V_i$, is the set of all indexed sets $\langle x_i \rangle_{i \in I}$ such that $x_i \in V_i$ for each $i \in I$. This generalizes Cartesian product $V_1 \times \cdots \times V_n$ (taking $I = \{1, \ldots, n\}$).

Let $\langle \leq_i \rangle_{i \in I}$ be an indexed set of partial order relations $\leq_i$ on sets V_i for each $i \in I$. The *product order* of $\langle \leq_i \rangle_{i \in I}$ is the binary relation

$$\{(\langle v_i \rangle_{i \in I}, \langle w_i \rangle_{i \in I}) \in (\times_{i \in I} V_i)^2 \mid \forall i \in I : v_i \leq_i w_i\}.$$

It is a binary relation on $\times_{i\in I}V_i$. Written differently, it is the Boolean function:

$$\lambda : (\times_{i\in I}V_i)^2 \to \mathcal{T}wo : (\langle v_i\rangle_{i\in I}, \langle w_i\rangle_{i\in I}) \mapsto \wedge_{i\in I}(v_i \leq_i w_i)\,.$$

A special case is if all V_i and $\leq_i$ are the same, i.e., for some V and $\leq$, it holds that $V_i = V$ and $\leq_i = \leq$ for each $i \in I$. Then the product relation $\times_{i\in I} \leq$ will be called the *pointwise extension* of $\leq$ on $V^I = \times_{i\in I}V$. Taking products of orders preserves many good properties of its component orders. It is well-known that the product order is a partial order. The product order of chain complete orders is chain complete order and the product order of complete lattice orders is a complete lattice order.

Let $\langle O_i\rangle_{i\in I}$ be an indexed set of operators $O_i \in X_i^{V_i}$. Then we define the lift operator $\uparrow_{i\in I} O_i$ as the operator in $(\times_{i\in I}X_i)^{(\times_{i\in I}V_i)}$ that maps elements $\langle v_i\rangle_{i\in I}$ to $\langle O_i(v_i)\rangle_{i\in I}$. In another notation, it is the function:

$$\lambda : \times_{i\in I}V_i \to \times_{i\in I}X_i : \langle v_i\rangle_{i\in I} \mapsto \langle O_i(v_i)\rangle_{i\in I}\,.$$

A special case arises when all O_i are the same operator $O : V \to V$. In this case, $\uparrow_{i\in I} O$ is a function in $\times_{i\in I}V = V^I$ mapping $\langle v_i\rangle_{i\in I}$ to $\langle O(v_i)\rangle_{i\in I}$. That is, it is the function $\lambda : V^I \to V^I : f \mapsto O \circ f$. We call this the *lifting* of $O : V \to V$ to the product V^I.

3.2 (Approximation) Fixpoint Theory

A binary relation $\leq$ on set V is a *partial order* if $\leq$ is reflexive, transitive and asymmetric. In that case, we call the mathematical structure $\langle V, \leq\rangle$ a *poset.* $\leq$ is *total* if for every $x, y \in V$, $x \leq y$ or $y \leq x$. The partial order $\leq$ is a *complete lattice order* if for each $X \subseteq V$, there exists a least upperbound $lub(X)$ and a greatest lower bound $glb(X)$. If $\leq$ is a complete lattice order of V, then V has a least element $\bot$ and a greatest element $\top$.

Let $\langle V, \leq\rangle, \langle W, \leq\rangle$ be two posets. An operator $O : V \to W$ is monotone if it is order preserving; i.e. if $x \leq y \in V$ implies $O(x) \leq O(y)$.

Let $\langle V, \leq\rangle$ be complete lattice with least element $\bot$ and greatest element $\top$. Its bilattice is the structure $\langle V^2, \leq_p, \leq\rangle$ with $(v_1, v_2) \leq_p (w_1, w_2)$ if $v_1 \leq w_1, v_2 \geq w_2$ and $(v_1, v_2) \leq (w_1, w_2)$ if $v_1 \leq w_1, v_2 \leq w_2$. The latter is the pointwise extension of $\leq$ to the bilattice. Both orders are known to be lattice orders. $\leq_p$ is called the *precision order.* The least precise element is $(\bot, \top)$ and most precise element is $(\top, \bot)$. An *exact* pair is of the form (v, v). A *consistent* pair (v, w) is one such that $v \leq w$. We say that (v, w) approximates $u \in V$ if $v \leq u \leq w$. The set of values approximated by (v, w) is $[v, w]$. This set is non-empty iff (v, w) is consistent. Exact pairs (V, V) are the maximally consistent pairs and they approximate a singleton $\{X\}$. We view the exact pairs as the embedding of V in V^2. Abusing this, we sometimes write v where (v, v) should be written. Pairs $(v, w) \in V^2$ are written as $\mathfrak{v}$, with $(\mathfrak{v})_1 = v, (\mathfrak{v})_2 = w$.

We define $V^c = \{(v, w) \in V^2 \mid v \leq w\}$. It is the set of consistent pairs. We will call such a pair an approximate value, and we call V^c the *approximate value space* of V. It can be shown that any non-empty set $X \subseteq V^c$ has a greatest lower bound $glb_{\leq_p}(X)$ in V^c, but not every set $X \subseteq V^c$ has a least upperbound in V^c. In particular, the exact elements are exactly the maximally precise elements. Hence, V^c is not a complete lattice. However, if X has an upperbound in V^c, then $lub(X)$ exists. Also, V^c is chain complete: every totally ordered subset $X \subseteq V^c$ has a least upperbound. It follows that each sequence $\langle (v_i, w_i)\rangle_{i<\alpha}$ of increasing precision has a least upperbound $lub(\langle (v_i, w_i)\rangle_{i<\alpha})$, called its limit. This suffices to warrant the existence of a least fixpoint for every $\leq_p$-monotone operator $O : V^c \to V^c$.

▶ **Example 1.** Consider the lattice $\mathcal{T}wo = \{\mathbf{t}, \mathbf{f}\}$ with $\mathbf{f} \leq \mathbf{t}$. The four pairs of its billatice $\mathcal{F}our$ correspond to the standard truth values of four-valued logic. The pairs $(\mathbf{t}, \mathbf{t})$ and $(\mathbf{f}, \mathbf{f})$

are the embeddings of true ($\mathbf{t}$) and false ($\mathbf{f}$) respectively. The pair $(\mathbf{f}, \mathbf{t})$ represents unknown ($\mathbf{u}$) and $(\mathbf{t}, \mathbf{f})$ represents the inconsistent value ($\mathbf{i}$). Here, the set $\mathcal{T}wo^c$ is the set of consistent pairs and is denoted $\mathcal{T}hree$. The precision order is $\mathbf{u} \leq_p \mathbf{t} \leq_p \mathbf{i}, \mathbf{u} \leq_p \mathbf{f} \leq_p \mathbf{i}$ and the product order is $\mathbf{f} \leq \mathbf{u} \leq \mathbf{t}, \mathbf{f} \leq \mathbf{i} \leq \mathbf{t}$.

For any lattice $\langle V, \leq \rangle$ and domain D, the pointwise extension of $\leq$ to V^D is a lattice order, also denoted as $\leq$. The lattice V^D has a bilattice $(V^D)^2$ and approximate value space $(V^D)^c$ which are isomorphic to $(V^2)^D$, respectiely $(V^c)^D$.

▶ **Example 2.** The billattice of $\mathcal{T}wo^D$ and the approximation space $(\mathcal{T}wo^D)^c$ are isomorphic to $\mathcal{F}our^D$, respectively $\mathcal{T}hree^D$ under the pointwise extensions of $\leq_p$ and $\leq$ of $\mathcal{F}our$ and $\mathcal{T}hree$. Elements of $\mathcal{F}our^D$ and $\mathcal{T}hree^D$ correspond to four and three valued sets.

Let D, C be complete lattices.

▶ **Definition 3.** For any function $f : D \to C$, we say that $A : D^c \to C^c$ is an *approximator* of f if (1) ($\leq_p$-monotonicity) A is $\leq_p$-monotone and (2) (exactness) for each $v \in D$, $A(v) \leq_p f(v)$. We call A *exact* if A preserves exactness. The projections of $A(v, w)$ on first and second argument are denoted $A(v, w)_1$ and $A(v, w)_2$.

Approximators of f allow to infer approximate output from approximate input for f. The co-domain of an approximator is equipped with a precision order which can be pointwise extended on $(C^c)^{D^c}$.

▶ **Definition 4.** We say that F is the *ultimate approximator* of f if F is the $\leq_p$-maximally precise approximator of f. We denote F as $\lceil f \rceil$.

One can prove that $\lceil f \rceil(\mathfrak{v}) = glb_{\leq_p}(\{f(v) \mid \mathfrak{v} \leq_p v \in D\})$.

▶ **Example 5.** The ultimate approximators of the standard Boolean functions $\wedge, \neg, \vee, \ldots,$ correspond to the standard 3-valued Boolean extensions known from the Kleene truth assignment. E.g. $\lceil \wedge \rceil$:

$\lceil \wedge \rceil$	$\mathbf{f}$	$\mathbf{u}$	$\mathbf{t}$
$\mathbf{f}$	$\mathbf{f}$	$\mathbf{f}$	$\mathbf{f}$
$\mathbf{u}$	$\mathbf{f}$	$\mathbf{u}$	$\mathbf{u}$
$\mathbf{t}$	$\mathbf{f}$	$\mathbf{u}$	$\mathbf{t}$

Let $\langle V, \leq \rangle$ be complete lattice with least element $\bot$ and greatest element $\top$. With an operator $O : V \to V$, many sorts of fixpoints can be associated: the standard fixpoints $O(x) = x$ and the *grounded* fixpoints of O [3]. For any approximator $A : V^c \to V^c$, more sorts of fixpoints can be defined:

- The A-Kripke-Kleene fixpoint is the $\leq_p$-least fixpoint of A.
- A partial A-stable fixpoint is a pair (x, y) such that
  - $A(x, y) = (x, y)$,
  - (x, y) is prudent, i.e., for all $z \leq y$, $A(z, y)_1 \leq y$ implies $x \leq z$.
  - there is no $z \in [x, y[$ such that $A(x, z)_2 \leq z$.
- The well-founded fixpoint of A is the least precise A-partial stable fixpoint.
- An A-stable fixpoint is an element $v \in L$ such that (v, v) is a partial A-stable fixpoint.

Assume A approximates O. It is well-known that the KK-fixpoint of A approximates all fixpoints of O and all partial stable fixpoints of A, hence also the well-founded fixpoint of A and the (exact) stable fixpoints of A. It can be shown that the three-valued immediate consequence operator of logic programs is an approximator of the two-valued one, and that the above sorts of fixpoints induce the different sorts of semantics of logic programming [6].

With a lattice operator $O : V \to V$, we define the ultimate well-founded fixpoint and the ultimate (partial) stable fixpoints as the well-founded fixpoint and the (partial) stable fixpoints of $\lceil O \rceil$. Compared with other approximators A of O, the ultimate approximator has the most precise KK-fixpoint and well-founded fixpoint, and -somewhat surprisingly- the most (exact) stable fixpoints. That is, the set of exact stable fixpoints of any approximator A of O is a subset of that set of $\lceil O \rceil$. Notice that the ultimate well-founded fixpoint of O is an element of the bilattice, but it may be (and often is) exact.

4 A typed higher order logic $\mathbb{L}$ with (nested) definitions

4.1 Type system

A typed logic $\mathbb{L}$ contains a type system, offering a method to expand arbitrary sets $\mathbb{B}$ of (user-defined) type symbols to a set $\mathbb{T}(\mathbb{B})$ of types, together with a method to expand a type structure $\mathcal{A}$ assigning sets of values to the symbols of $\mathbb{B}$, to an assignment $\bar{\mathcal{A}}$ of sets of values to all types in $\mathbb{T}(\mathbb{B})$. We formalize these concepts.

▶ **Definition 6.** A type vocabulary $\mathbb{B}$ is a (finite) set of type symbols. A type structure $\mathcal{A}$ for $\mathbb{B}$ is an assignment of sets $\tau^{\mathcal{A}}$ to each $\tau \in \mathbb{B}$.

▶ **Definition 7.** A *type constructor* is a pair (tc, Sem_{tc}) of a type constructor symbol tc of some arity $n \geq 0$ and its associated semantic function Sem_{tc} which maps n-tuples of sets to sets such that Sem_{tc} preserves set isomorphism. [1]

Given a set $\mathbb{B}$ of type symbols and a set of type constructor symbols, a set of (finite) type terms τ can be built from them. In general, the set $\mathbb{T}(\mathbb{B})$ of types of a logic theory form a subset of the set of these type terms.

▶ **Definition 8.** A type system consists of a set of type constructors and a function mapping any set $\mathbb{B}$ of type symbols to a set $\mathbb{T}(\mathbb{B})$ of type terms formed from $\mathbb{B}$ and the type constructor symbols such that for any bijective renaming $\theta : \mathbb{B} \to \mathbb{B}'$, $\mathbb{T}(\mathbb{B})$ and $\mathbb{T}(\mathbb{B}')$ are identical modulo the renaming θ. An element of $\mathbb{T}(\mathbb{B})$ is called a *type*. A *compound type* is an element of $\mathbb{T}(\mathbb{B}) \setminus \mathbb{B}$.

For a given type system, it is clear that any type structure $\mathcal{A}$ for $\mathbb{B}$ can be expanded in a unique way to all type terms by iterated application of the semantic functions Sem_{tc}.

▶ **Definition 9.** Given a type system and a type structure $\mathcal{A}$ for a set $\mathbb{B}$ of type symbols, we define $\bar{\mathcal{A}}$ as the unique expansion of $\mathcal{A}$ to $\mathbb{T}(\mathbb{B})$ defined by induction on the structure of type terms and using the semantic type constructor functions Sem_{tc} of type constructors.

By slight abuse of notation, we write $\tau^{\bar{\mathcal{A}}}$ as $\tau^{\mathcal{A}}$.

▶ **Definition 10.** We call a type system *type closed* if for every $\mathbb{B}$, $\mathbb{T}(\mathbb{B})$ is the set of all type terms built over $\mathbb{B}$ and the type constructors of the system.

▶ **Example 11.** The type system of the logic that we will define below is type closed. Its type constructor symbols and corresponding semantic type operators are:

[1] That is, if there exists bijections between S_1 and S'_1, …, S_n and S'_n, then there is a bijection between $Sem_{tc}(S_1, \ldots, S_n)$ and $Sem_{tc}(S'_1, \ldots, S'_n)$.

- the 0-ary Boolean type constructor symbol BOOL with $Sem_{\mathsf{BOOL}} = \mathcal{T}wo$;
- the 0-ary natural number constructor type symbol NAT with $Sem_{NAT} = \mathbb{N}$;
- the n-ary Cartesian product type constructor symbol $\times_n$; we write $\times_n(\tau_1, \ldots, \tau_n)$ as $\tau_1 \times \cdots \times \tau_n$ and $\times_n(\tau, \ldots, \tau)$ as τ^n. The semantic operator $Sem_{\times_n}$ maps tuples of sets $(S_1, \ldots, S_n)$ to the Cartesian product $S_1 \times \cdots \times S_n$;
- the function type constructor $\rightarrow$ with $Sem_{\rightarrow}$ mapping pairs of sets (X, Y) to the function set Y^X.

In typed lambda calculus, Cartesian product is often not used (it can be simulated using higher order functions and *currying*). Here, we keep it in the language to connect easier with FO.

▶ **Example 12.** The type system of typed classical first order logic uses the type constructors corresponding to BOOL, $\times_n$ and $\rightarrow$ in the previous example. $\mathbb{T}(\mathbb{B})$ is the set $\{\tau_1 \times \cdots \times \tau_n \rightarrow \mathsf{BOOL}, \tau_1 \times \cdots \times \tau_n \rightarrow \tau \mid \tau_1, \ldots, \tau_n, \tau \in \mathbb{B}\}$. It consists of first order predicate types $\tau_1 \times \cdots \times \tau_n \rightarrow \mathsf{BOOL}$ and first order function types $\tau_1 \times \cdots \times \tau_n \rightarrow \tau$. The type system of untyped classical first order logic is obtained by fixing $\mathbb{B} = \{U\}$, where U represents the universe of discourse. Clearly, (typed) FO is not type closed.

From now on, we assume a fixed type system. We also assume an infinite supply of type symbols, and for all types τ that can be constructed from this supply and the given type constructor symbols, an infinite supply of symbols σ of type τ. We write $\sigma : \tau$ to denote that τ is the type of σ.

▶ **Definition 13.** A *vocabulary* (or *signature*) Σ is a tuple $\langle \mathbb{B}, Sym \rangle$ with $\mathbb{B}$ a set of type symbols, Sym a set of symbols σ of type $\tau \in \mathbb{T}(\mathbb{B})$.

We write $\mathbb{T}(\Sigma)$ to denote $\mathbb{T}(\mathbb{B})$.

Let Σ be a vocabulary $\langle \mathbb{B}, Sym \rangle$.

▶ **Definition 14.** An *assignment* to Sym in type structure $\mathcal{A}$ for $\mathbb{B}$ is a mapping $A : Sym \rightarrow \{\tau^{\mathcal{A}} \mid \tau \in \mathbb{T}(\Sigma)\}$ such that for each $\sigma : \tau \in Sym$, $\sigma^A \in \tau^{\mathcal{A}}$. That is, the value of σ^A is of type τ in $\mathcal{A}$. The set of assignments to Sym in $\mathcal{A}$ is denoted $Assign^{\mathcal{A}}_{Sym}$.

▶ **Definition 15.** A Σ-*structure* I is a tuple $\langle \mathcal{A}, (\cdot)^I \rangle$ of a type structure $\mathcal{A}$ for $\mathbb{B}$, and $(\cdot)^I$ an *assignment* to all symbols $\sigma \in Sym$ in type structure $\mathcal{A}$. We denote the value of σ as σ^I. The class of all Σ-structures is denoted $\mathbb{S}(\Sigma)$.

We frequently replace $\mathcal{A}$ by I; e.g., we may write τ^I for $\tau^{\mathcal{A}}$.

Let Σ be a vocabulary with type symbols $\mathbb{B}$, I a Σ-structure. Let Sym be a set of symbols with types in $\mathbb{T}(\mathbb{B})$ (it may contain symbols not in Σ). For any assignment $A \in Assign^I_{Sym}$ to Sym in (the type structure of) I, we denote by $I[A]$ the structure that is identical to I except that for every $\sigma \in Sym$, $\sigma^{I[A]} = \sigma^A$. This is a structure of the vocabulary $\Sigma \cup Sym$. As a shorthand notation, let σ be a symbol of type τ and v a value of type τ in I, then $[\sigma : v]$ is the assignment that maps σ to v, and $I[\sigma : v]$ is the updated structure.

▶ **Definition 16.** A Σ-*infon* $\mathfrak{i}$ of type $\tau \in \mathbb{T}(\Sigma)$ is a mapping that associates with each Σ-structure I a value $\mathfrak{i}(I)$ of type τ in I. The class of Σ-infons is denoted $\mathfrak{I}_\Sigma$. Each symbol $\sigma \in \Sigma$ of type τ defines the Σ-infon $\mathfrak{i}_\sigma$ of type τ that associates with each Σ-structure I the value σ^I.

Infons of type τ are similar to intensional objects in Montague's intensional logic [14]. An infon of type BOOL provides an abstract syntax independent representation of a *quantum of information*. It maps a structure representing a possible state of affairs in which the information holds to true, and other structures to false. It will be the case that two sentences are logically equivalent in the standard sense iff they induce the same infon.

4.2 Language constructs

▶ **Definition 17.** A language construct $\mathfrak{C}$ consists of an arity n representing the number of arguments, a typing rule $Type_{\mathfrak{C}}$ specifying the allowable argument types and the corresponding expression type, and a semantic operator $Sem_{\mathfrak{C}}$. A *typing rule* $Type_{\mathfrak{C}}$ is a partial function from n argument types $\tau_1, \dots, \tau_n$ to a type $Type_{\mathfrak{C}}(\tau_1, \dots, \tau_n) = \tau$ that preserves renaming of type symbols; i.e., if θ is a bijective renaming of type symbols, then $Type_{\mathfrak{C}}(\theta(\tau_1), \dots, \theta(\tau_n)) = \theta(\tau)$. If $Type_{\mathfrak{C}}$ is defined for $\tau_1, \dots, \tau_n$, we call $\tau_1, \dots, \tau_n$ an argument type for $\mathfrak{C}$. The semantic operator $Sem_{\mathfrak{C}}$ is a partial mapping defined for all tuples of infons $\mathfrak{i}_1, \dots, \mathfrak{i}_n$ of all argument types $\tau_1, \dots, \tau_n$ for $\mathfrak{C}$ to an infon of the corresponding expression type τ.

A language construct $\mathfrak{C}$ takes a sequence of expressions $e_1, \dots, e_n$ as argument and yields the compound expression $\mathfrak{C}(e_1, \dots, e_n)$. This determines the abstract syntax of expressions. We often specify a concrete syntax for $\mathfrak{C}$ (which often disagrees with the abstract syntax).

Let $\tau_1, \dots, \tau_n$ be an argument type for $\mathfrak{C}$ yielding the expresson type τ. Then for well-typed expressions $e_1, \dots, e_n$ of respectively types $\tau_1, \dots, \tau_n$, the (abstract) compound expression $\mathfrak{C}(e_1, \dots, e_n)$ is well-typed and of type τ. Some language constructs are polymorphic and apply to expressions of many types. Others have unique type for each argument.

▶ **Example 18.** The tupling operator TUP is a polymorphic language construct that maps expressions $e_1, \dots, e_n$ of arbitrary types $\tau_1, \dots, \tau_n$ to the compound expression $TUP(e_1, \dots, e_n)$ of type $\tau_1 \times \dots \times \tau_n$. The concrete syntax is $(e_1, \dots, e_n)$.

The conjunction $\wedge$ maps expressions e_1, e_2 of type BOOL to $\wedge(e_1, e_2)$ of type BOOL. The concrete syntax is $e_1 \wedge e_2$.

The set of language constructs of a logic $\mathbb{L}$ together with a vocabulary Σ uniquely determines the set $Exp^{\mathbb{L}}_{\Sigma}$ of well-typed expressions over Σ, as well as a function $Type_{\mathbb{L}} : Exp^{\mathbb{L}}_{\Sigma} \to \mathbb{T}(\Sigma)$. Formally, consider the set of (finite) labeled trees with nodes labeled by language constructs of $\mathbb{L}$ and symbols of Σ. Within this set, the function $Type_{\mathbb{L}}$ is defined by induction on the structure of expressions

- $Type_{\mathbb{L}}(\sigma) = \tau$ if $\sigma \in \Sigma$ is a symbol of type τ;
- $Type_{\mathbb{L}}(\mathfrak{C}(e_1, \dots, e_n)) = Type_{\mathfrak{C}}(Type_{\mathbb{L}}(e_1), \dots, Type_{\mathbb{L}}(e_n))$.

This mapping $Type_{\mathbb{L}}$ is a partial function, the domain of which is exactly $Exp^{\mathbb{L}}_{\Sigma}$.

Furthermore, the set of language constructs of $\mathbb{L}$ determines for each well-typed expression $e \in Exp^{\mathbb{L}}_{\Sigma}$ of type τ a unique infon $Sem_{\mathbb{L}}(e)$ of that type. The function $Sem_{\mathbb{L}}$ is defined by induction on the structure of expressions by the following equation:

- $Sem_{\mathbb{L}}(\sigma) = \mathfrak{i}_{\sigma}$ if $\sigma \in \Sigma$;
- $Sem_{\mathbb{L}}(\mathfrak{C}(e_1, \dots, e_n)) = Sem_{\mathfrak{C}}(Sem_{\mathbb{L}}(e_1), \dots, Sem_{\mathbb{L}}(e_n))$.

This property warrants a strong form of Frege's compositionality principle.

We call a logic *substitution closed* if every expression of some type may occur at any argument position of that type. E.g., propositional logic and first order logic are substitution closed, but CNF is not due to the syntactical restrictions on the format of CNF formulas.

4.2.1 Simply typed lambda calculus with infon semantics

Below, we introduce a concrete substitution closed logic $\mathbb{L}$ with a type closed type system. We specify the main language constructs.

- $TUP(e_1, \dots, e_n)$:
  - concrete syntax is $(e_1, \dots, e_n)$;
  - typing rule: for arguments of types $\tau_1, \dots, \tau_n$ respectively, the compound expression is of type $\tau_1 \times \dots \times \tau_n$;

- Sem_{TUP} maps finite tuples $\bar{\mathfrak{i}}$ to the infon $\lambda I \in \mathbb{S}(\Sigma) : (\mathfrak{i}_1(I), \dots, \mathfrak{i}_n(I))$.
- $APP(e, e_1)$:
  - concrete syntax $e(e_1)$;
  - typing rule: for arguments of type $\tau_1 \to \tau, \tau_1$, the expression is of type τ;
  - Sem_{APP}: maps well-typed infons $\mathfrak{i}, \mathfrak{i}_1$ to $\lambda I \in \mathbb{S}(\Sigma) : \mathfrak{i}(I)(\mathfrak{i}_1(I))$.
- $Lambda(\bar{\sigma}, e)$: here $\bar{\sigma}$ is a finite sequence $\sigma_1, \dots, \sigma_n$ of symbols (not expressions);
  - concrete syntax $\lambda\sigma_1 \dots \sigma_n : e$; if e is Boolean, then $\{\sigma_1 \dots \sigma_n : e\}$;
  - typing rule: if the symbols $\sigma_1, \dots, \sigma_n$ are of types $\tau_1, \dots, \tau_n$ and the second argument is of type τ, the expression is of type $(\tau_1 \times \dots \times \tau_n) \to \tau$;
  - Sem_{Lambda} maps an $\Sigma \cup \{\sigma_1, \dots, \sigma_n\}$-infon $\mathfrak{i}$ of type τ to the Σ-infon $\lambda I \in \mathbb{S}(\Sigma) : F_I$, where F_I is the function $\lambda\bar{x} \in {\tau_1}^I \times \dots \times {\tau_n}^I : \mathfrak{i}(I[\bar{\sigma} : \bar{x}])$.

Equality, connectives and quantifiers are introduced using interpreted symbols, symbols with a fixed interpretation in each structure.

The logical symbols $\wedge, \vee : \mathsf{BOOL} \times \mathsf{BOOL} \to \mathsf{BOOL}$ and $\neg : \mathsf{BOOL} \to \mathsf{BOOL}$ have the standard Boolean functions as interpretations in every structure.

Quantifiers and equality are polymorphic. We introduce instantiations of them for all types τ. For every type τ, $\forall_\tau, \exists_\tau$ are symbols of type $(\tau \to \mathsf{BOOL}) \to \mathsf{BOOL}$. For concrete syntax, for $\forall_\tau(Lambda(\sigma, e))$ with e a Boolean expression and $\sigma : \tau$, we write $\forall\sigma : e$ (we dropped the underscore from $\forall_\tau$ since τ is the type of σ). It also corresponds to a quantified set comprehension $\forall_\tau(\{\sigma : \varphi\})$. In any structure I, ${\forall_\tau}^I$ is the Boolean function $\lambda X \in (\tau \to \mathsf{BOOL})^I : (X = \tau^I)$ that maps a set X with elements of type τ to $\mathbf{t}$ if X contains all elements of this type in I. Likewise, ${\exists_\tau}^I$ is the Boolean function $\lambda X \in (\tau \to \mathsf{BOOL})^I : (X = \emptyset)$.

Equality is a polymorphic interpreted predicate. For each τ, introduce a symbol $=_\tau$ of type $\tau \times \tau \to \mathsf{BOOL}$. The concrete syntax is $e = e_1$. Its interpretation in an arbitrary structure I is the identity relation of type τ^I.

Likewise, standard aggregate functions such as cardinality and sum are introduced as interpreted higher order Boolean functions. E.g., we introduce the interpreted symbol

$$Card_\tau : ((\tau \to \mathsf{BOOL}) \times NAT) \to \mathsf{BOOL}$$

interpreted in each structure I as the function

$${Card_\tau}^I : ((\tau \to \mathsf{BOOL})^I \times \mathbb{N}) \to \mathcal{T}wo : (S, n) \mapsto (\#(S) = n)).$$

We have chosen here to define $Card_\tau$ as a binary predicate symbol rather than as a unary function, because it is a partial function defined only on finite sets and our logic is not equipped for partial functions.

4.3 The definition construct DEF for higher order and nested definitions

So far, we have defined typed lambda calculus under an infon semantics. In this section, we extend the language with higher order versions of definitions as in the logic FO(ID). There, definitions are conventionally written as finite set of rules $\forall\bar{\sigma}(P(\bar{\sigma}) \leftarrow \varphi)$ where $P : (\bar{\tau} \to \mathsf{BOOL})$ is a predicate symbol, $\bar{\sigma} : \bar{\tau}$ a (sequence of) symbol(s), and φ a Boolean expression. E.g.,

Listing 1 The transitive closure of G.

```
{
∀x ∀y: Reach(x,y)← G(x,y).
∀x ∀z: Reach(x,z)← G(x,y)∧ Reach(y,z).
}
```

In the abstract syntax, a rule $\forall\bar{\sigma}(P(\bar{\sigma}) \leftarrow \varphi)$ will be represented as a pair $(P, \{\bar{\sigma} : \varphi\})$.

In general, an abstract expression of the definition construct DEF is of the form $DEF(\bar{P}, \bar{e})$ where $\bar{P}$ is a finite sequence $(P_1, \dots, P_n)$ $(n > 0)$ of predicate symbols and $\bar{e}$ an equally long sequence of expressions. We write $(P, e) \in \Delta$ to denote that for some $i \leq n$, $P_i = P$ and $e_i = e$. Let $DP(\Delta)$ be $\{P_1, \dots, P_n\}$, the set of *defined symbols* of Δ. It is possible that the same symbol P has multiple rules in Δ (as in the above example). Below, we use the mathematical variable Δ to denote definition expressions.

- For the concrete syntax, $DEF(\bar{P}, \bar{e})$ represents a definition with n rules corresponding to the pairs (P_i, e_i). If e_i is the set comprehension $\{\bar{\sigma} : \varphi\}$, the corresponding rule in concrete syntax is $\forall\bar{\sigma}(P(\bar{\sigma}) \leftarrow \varphi)$.
  Due to the substitution closedness of the logic, new abstract rules are allowed. E.g., $(Reach, G)$ is an abstract representation that is equivalent to the first rule in the *Reach* example, and it is an alternative way to represent the base case of the reachability relation.
- Typing rule: if for each $i \in [1, n]$, P_i, e_i are of the same type $\tau_i \to \mathsf{BOOL}$ then the definition expression is of type BOOL. It follows that the value of a definition in a structure is true or false. Note that defined symbols are predicate symbols.
- Sem_{DEF}: this operator maps tuples $((P_1, \dots, P_n), (\mathfrak{i}_1, \dots, \mathfrak{i}_n))$ where each $\mathfrak{i}_i$ is an infon of type τ_i to an infon $\mathfrak{i}$ of type BOOL. This operator will be applied to the infons $\mathfrak{i}_i$ of the expressions e_i. To define the infon $\mathfrak{i}$ from the input, we construct for each $I \in \mathbb{S}(\Sigma)$ the immediate consequence operator Γ^I_Δ.
  The operator Γ^I_Δ is an operator on $Assign^I_{DP(\Delta)}$, the lattice of $DP(\Delta)$-assignments in I. Note that for a rule $(P, e) \in D$, the value e^I of e in a structure I is exactly the set that this rule produces for P in I. The total produced value for P is then obtained by taking the union of all rules defining P. Formally, for each $P \in DP(\Delta)$, let $INF_P = \{\mathfrak{i}_i \mid P_i = P\}$. That is, INF_P is the set of infons amongst $\mathfrak{i}_1, \dots, \mathfrak{i}_n$ that correspond to rules with P in the head. Then Γ^I_Δ maps an assignment $A \in Assign^I_{DP}$ to an assignment B such that for each $P \in DP$:

  $$P^B = lub_{\leq}(\{\mathfrak{i}_i(I[A]) \mid \mathfrak{i}_i \in INF_P\}$$

  That is, P^B is the union of what each rule of P produces in the structure $I[A]$.
  The operator Γ^I_Δ is well-defined, and indeed, it is the immediate consequence operator of Δ in structure I. This is a lattice operator on the lattice of assignments of the defined symbols $DP(\Delta)$ in I. Consequently, this operator will have an ultimate well-founded fixpoint UWF^I_Δ, the well-founded fixpoint of the ultimate approximator $\lceil \Gamma^I_\Delta \rceil$. This fixpoint may be exact or not. We define the truth value Δ^I of Δ in I as $(I = UWF^I_\Delta)$, that is, $\Delta^I = \mathbf{t}$ if I is the exact ultimate well-founded fixpoint of the operator, and $\Delta^I = \mathbf{f}$ otherwise. The infon $Sem_{\mathbb{L}}(\Delta)$ is the Boolean infon $\lambda I \in \mathbb{S}(\Sigma) : (I = UWF^I_\Delta)$.

The semantic operator $Sem_{\mathbb{L}}$ associates with each expression an infon, and with each theory T a Boolean infon $\mathfrak{i}$. This induces a model semantics, in particular $M \models T$ if $\mathfrak{i}(M) = \mathbf{t}$.

▶ **Theorem 19.** *The logic FO(ID) equipped with the ultimate well-founded semantics for definitions is a fragment of $\mathbb{L}$. That is, any theory T of FO(ID) corresponds syntactically to one T' of $\mathbb{L}$ and T and T' have the same models (taking the ultimate well-founded semantics for definitions).*

4.4 Applications for Higher Order Definitions

Higher order definitions are natural representations for some complex concepts. A standard example is a definition of winning positions in two-player games as can be seen in Listing 2. This definition of win and lose is a monotone second order definition that uses simultaneous definition and has a two-valued well-founded model.

Listing 2 cur is a winning position in a two-player game.

```
{
∀cur ∀Move ∀IsWon: win(cur, Move, IsWon) ← IsWon(cur) ∨
   ∃ nxt : Move(cur,nxt) ∧ lose(nxt,Move,IsWon).
∀cur ∀Move ∀IsWon: lose(cur,Move, IsWon) ← ¬IsWon(cur) ∧
   ∀ nxt : Move(cur,nxt) ⇒ win(nxt,Move,IsWon).
}
```

4.5 Templates

In [5], a subclass of higher order definitions were defined as templates. These templates allow us to define an abstract concept in an isolation, so that it can be reused multiple times. This prevents code duplication and results in more readable specifications. In the same context, we identified applications for nested definitions. An example of this can be seen in Listing 3. In that example a binary higher order predicate tc is defined, such that tc(P,Q) holds iff Q is defined as the transitive closure of P.

Listing 3 This template TC expresses that Q is the transitive closure of P.

```
{
∀Q ∀P: tc(P,Q) ←
   {∀x ∀y: Q(x,y) ← P(x,y) ∨(∃ z: Q(x,z)∧Q(z,y))}.
}
```

Note that using this definition of tc, the definition in Listing 1 can simply be replaced with the atom tc(Reach,G). This demonstrates the abstraction power of these definitions.

4.6 Graph Morphisms

A labeled graph is a tuple of a set of vertices, a set of edges between these vertices, and a labeling function on these vertices. Many applications work with labeled graphs: one example is the graph mining problem [12], which requires the notion of homomorphisms and isomorphisms between graphs. As other applications require these same concepts, these concepts lend themselves to a definition in isolation.

To achieve this, we first define the graph type as an alias for the higher order type $\mathcal{P}(node) \times \mathcal{P}(node \times node) \times \mathcal{P}(node \to label)$, where the components of the triple are called Vertex, Edge and Label respectively. To define when two graphs are homomorph and isomorph, we first define a helper predicate homomorphism. This predicate takes a function and two graphs, and is true when this function represents a homomorphism from the first graph to the second. We then define homomorph and isomorph in terms of the homomorphism predicate. In Listing 4, these higher order predicates are defined using higher order definitions. The higher order arguments of these definitions are either decomposed into the different tuple elements using matching (Line 2) or accepted as a single entity (Line 6).

Listing 4 Defining `homomorph` and `isomorph`.

```
{
homomorphism(F, (V1, Edge1, Label1), (V2, Edge2, Label2)) ←
     (∀ x, y [V1] : Edge1(x, y)  ⟹  Edge2(F(x), F(y))) ∧
     (∀ x : Label1(x) = Label2(F(x)).

homomorph(G1,G2) ←
    ∃ F [G1.Vertex:G2.Vertex] :  homomorphism(F, G1, G2).

isomorph(G1, G2) ←
    (∃ F [G1.Vertex:G2.Vertex], G [G2.Vertex:G1.Vertex] :
      (∀ x [G1.Vertex] : G(F(x)) = x) ∧
      homomorphism(F, G1, G2) ∧ homomorphism(G, G2, G1) ).
}
```

5 Conclusion

We defined a logic integrating typed higher order lambda calculus with definitions. The logic is type closed and substitution closed, allows definitions of higher order predicates and nested definitions. The logic satisfies a strong form of Frege's compositionality principle. The principles that we used allow also to define rules under other semantics (e.g., stable semantics). For future work, one question is how to define standard well-founded semantics for definitions in $\mathbb{L}$ rather than the ultimate well-founded semantics. It is well-known that both semantics often coincide, e.g., always when the standard well-founded model is two-valued, which is frequently the case when rule sets are intended to express definitions of concepts. Nevertheless, standard well-founded semantics is computationally cheaper and seems easier to implement. This provides a good motivation. Another question is how to extend definitions for arbitrary symbols, that is, for functions.

References

1 H. Abramson and H. Rogers. *Meta-programming in logic programming.* MIT Press, 1989.

2 Jon Barwise and John Etchemendy. Information, infons, and inference. *Situation theory and its applications*, 1(22), 1990.

3 Bart Bogaerts. *Groundedness in logics with a fixpoint semantics.* PhD thesis, Informatics Section, Department of Computer Science, Faculty of Engineering Science, June 2015. Denecker, Marc (supervisor), Vennekens, Joost and Van den Bussche, Jan (cosupervisors). URL: `https://lirias.kuleuven.be/handle/123456789/496543`.

4 Weidong Chen, Michael Kifer, and David S Warren. Hilog: A foundation for higher-order logic programming. *The Journal of Logic Programming*, 15(3):187–230, 1993.

5 Ingmar Dasseville, Matthias van der Hallen, Gerda Janssens, and Marc Denecker. Semantics of templates in a compositional framework for building logics. *TPLP*, 15(4-5):681–695, 2015. `doi:10.1017/S1471068415000319`.

6 Marc Denecker, Victor Marek, and Mirosław Truszczyński. Approximations, stable operators, well-founded fixpoints and applications in nonmonotonic reasoning. In Jack Minker, editor, *Logic-Based Artificial Intelligence*, volume 597 of *The Springer International Series in Engineering and Computer Science*, pages 127–144. Springer US, 2000. `doi:10.1007/978-1-4615-1567-8_6`.

7 Marc Denecker and Eugenia Ternovska. A logic of nonmonotone inductive definitions. *ACM Trans. Comput. Log.*, 9(2):14:1–14:52, April 2008. `doi:10.1145/1342991.1342998`.

8 Keith Devlin. *Logic and information*. Cambridge University Press, 1991.

9 Wolfgang Faber, Gerald Pfeifer, and Nicola Leone. Semantics and complexity of recursive aggregates in answer set programming. *Artif. Intell.*, 175(1):278–298, 2011. `doi:10.1016/j.artint.2010.04.002`.

10 Paolo Ferraris. Answer sets for propositional theories. In *Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, pages 119–131, 2005. `doi:10.1007/11546207_10`.

11 Michael Gelfond and Yuanlin Zhang. Vicious circle principle and logic programs with aggregates. *TPLP*, 14(4-5):587–601, 2014. `doi:10.1017/S1471068414000222`.

12 Tias Guns. Declarative pattern mining using constraint programming. *Constraints*, 20(4):492–493, 2015.

13 Yuri Gurevich and Itay Neeman. Logic of infons: The propositional case. *ACM Trans. Comput. Log.*, 12(2):9, 2011. `doi:10.1145/1877714.1877715`.

14 Jerry R Hobbs and Stanley J Rosenschein. Making computational sense of montague's intensional logic. *Artificial Intelligence*, 9(3):287–306, 1977.

15 David B. Kemp and Peter J. Stuckey. Semantics of logic programs with aggregates. In Vijay A. Saraswat and Kazunori Ueda, editors, *ISLP*, pages 387–401. MIT Press, 1991.

16 Javier Leach, Susana Nieva, and Mario Rodríguez-Artalejo. Constraint logic programming with hereditary harrop formula. *CoRR*, cs.PL/0404053, 2004.

17 Vladimir Lifschitz. Answer set planning. In Danny De Schreye, editor, *Logic Programming: The 1999 International Conference, Las Cruces, New Mexico, USA, November 29 – December 4, 1999*, pages 23–37. MIT Press, 1999.

18 Victor Marek and Mirosław Truszczyński. Stable models and an alternative logic programming paradigm. In Krzysztof R. Apt, Victor Marek, Mirosław Truszczyński, and David S. Warren, editors, *The Logic Programming Paradigm: A 25-Year Perspective*, pages 375–398. Springer-Verlag, 1999. URL: `http://arxiv.org/abs/cs.LO/9809032`.

19 Gopalan Nadathur and Dale Miller. An overview of LambdaProlog. In *Fifth International Conference and Symposium on Logic Programming. MIT Press*, 1988.

20 Ilkka Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Ann. Math. Artif. Intell.*, 25(3-4):241–273, 1999. `doi:10.1023/A:1018930122475`.

21 Nikolay Pelov, Marc Denecker, and Maurice Bruynooghe. Well-founded and stable semantics of logic programs with aggregates. *TPLP*, 7(3):301–353, 2007. `doi:10.1017/S1471068406002973`.

22 Tran Cao Son, Enrico Pontelli, and Islam Elkabani. An unfolding-based semantics for logic programming with aggregates. *CoRR*, abs/cs/0605038, 2006. URL: `http://arxiv.org/abs/cs/0605038`.

23 Shahab Tasharrofi and Eugenia Ternovska. A semantic account for modularity in multi-language modelling of search problems. In Cesare Tinelli and Viorica Sofronie-Stokkermans, editors, *Frontiers of Combining Systems, 8th International Symposium, FroCoS 2011, Saarbrücken, Germany, October 5-7, 2011. Proceedings*, volume 6989 of *Lecture Notes in Computer Science*, pages 259–274. Springer, 2011. `doi:10.1007/978-3-642-24364-6_18`.

24 Shahab Tasharrofi and Eugenia Ternovska. Three semantics for modular systems. *CoRR*, abs/1405.1229, 2014. URL: `http://arxiv.org/abs/1405.1229`.

25 Allen Van Gelder. The well-founded semantics of aggregation. In *PODS*, pages 127–138. ACM Press, 1992. `doi:10.1145/137097.137854`.

Inference in Probabilistic Logic Programs Using Lifted Explanations*

Arun Nampally[1] **and C. R. Ramakrishnan**[2]

1 **Computer Science Dept., Stony Brook University, Stony Brook, NY, USA**
`anampally@cs.stonybrook.edu`

2 **Computer Science Dept., Stony Brook University, Stony Brook, NY, USA**
`cram@cs.stonybrook.edu`

Abstract

In this paper, we consider the problem of lifted inference in the context of Prism-like probabilistic logic programming languages. Traditional inference in such languages involves the construction of an explanation graph for the query that treats each instance of a random variable separately. For many programs and queries, we observe that explanations can be summarized into substantially more compact structures introduced in this paper, called "lifted explanation graphs". In contrast to existing lifted inference techniques, our method for constructing lifted explanations naturally generalizes existing methods for constructing explanation graphs. To compute probability of query answers, we solve recurrences generated from the lifted graphs. We show examples where the use of our technique reduces the asymptotic complexity of inference.

1998 ACM Subject Classification D.1.6 Logic Programming, D.3.3 Language Constructs and Features – Constraints,F.3.2 Semantics of Programming Languages – Operational semantics, I.2.3 Deduction and Theorem Proving – Logic programming, Resolution, Uncertain, "fuzzy", and probabilistic reasoning

Keywords and phrases Probabilistic logic programs, Probabilistic inference, Lifted inference, Symbolic evaluation, Constraints

Digital Object Identifier 10.4230/OASIcs.ICLP.2016.15

1 Introduction

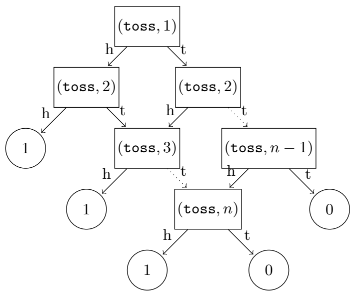
Probabilistic Logic Programming (PLP) provides a declarative programming framework to specify and use combinations of logical and statistical models. A number of programming languages and systems have been proposed and studied under the framework of PLP, e.g. PRISM [12], Problog [4], PITA [11] and Problog2 [5] etc. These languages have similar declarative semantics based on the *distribution semantics* [13]. The inference algorithms used in many of these systems to evaluate the probability of query answers, e.g. PRISM, Problog and PITA, are based on a common notion of *explanation graphs*. These graphs represent explanations, which are sets of *probabilistic choices* that are abduced during query evaluation. Explanation graphs are implemented differently by different systems; e.g. PRISM uses tables to represent them under *mutual exclusion* and *independence* assumptions on explanations; ProbLog and PITA represents them using Binary Decision Diagrams (BDDs).

Inference based on explanation graphs does not scale well to logical/statistical models with large numbers of random processes and variables. In particular, in models containing *families*

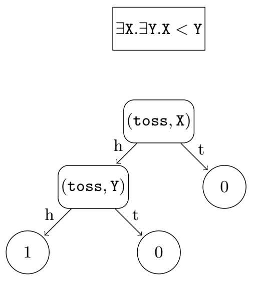
* This work was supported in part by NSF grants IIS 1447549 and CNS 1405641.

```
% Two distinct tosses show "h"
twoheads :-
    X in coins,
    msw(toss, X, h),
    Y in coins,
    {X < Y},
    msw(toss, Y, h).

% Cardinality of coins:
:- population(coins, 100).

% Distribution parameters:
:- set_sw(toss,
categorical([h:0.5, t:0.5])).
```

(a) Simple Px program



(b) Ground expl. Graph



(c) Lifted expl. Graph

Figure 1 Example program and ground explanation graph.

of independent, identically distributed (i.i.d) random variables, outcomes of individual random variables are abduced. However, as developments in the area of *lifted inference* [10, 2, 7] have shown, vast savings in computational effort can be made by exploiting the symmetry in models with populations of i.i.d random variables. The lifted inference algorithms seek to treat a set of i.i.d random variables as single unit and aggregate their behavior to achieve computational speedup. *This paper presents a structure for representing explanation graphs compactly by exploiting the symmetry with respect to i.i.d random variables, and a procedure to build this structure without enumerating each instance of a random process.*

Illustration. The simple example in Fig. 1 shows a program describing a process of tossing a number of i.i.d. coins, and evaluating if at least two of them came up "heads". The example is specified in an extension of the PRISM language, called Px. Explicit random processes of PRISM enables a clearer exposition of our approach. In PRISM and Px, a special predicate of the form `msw(`p`,` i`,` v`)` describes, given a random process p that defines a family of i.i.d. random variables, that v is the value of the i-th random variable in the family. The argument i of `msw` is called the ith *instance* argument. In this paper, we consider Param-Px, a further extension of Px to define parameterized programs. In Param-Px, a built-in predicate, `in` is used to specify membership; e.g. x `in` s means x is member of an enumerable set s. The size of s is specified by a separate `population` directive. The program in Fig. 1 defines a family of random variables generated by `toss`. The instances that index these random variables are drawn from the set `coins`. Finally, predicate `twoheads` is defined to hold if tosses of at least two distinct coins come up "`h`".

State of the Art, and Our Solution. Inference in PRISM, Problog and PITA follows the structure of the derivations for a query. Consider the program in Fig. 1(a) and let the cardinality of the set of coins be n. The query `twoheads` will take $\Theta(n^2)$ time, since it will construct bindings to both `X` and `Y` in the clause defining `twoheads`. However, the size of an explanation graph is $\Theta(n)$, as shown in Fig. 1(b). Computing the probability of the query over this graph will also take $\Theta(n)$ time.

In this paper, we present a technique to construct a symbolic version of an explanation graph, called a *lifted explanation graph* that represents instances symbolically and avoids enumerating the instances of random processes such as `toss`. The lifted explanation graph for query `twoheads` is shown in Fig. 1(c). Unlike traditional explanation graphs where nodes are specific instances of random variables, nodes in the lifted explanation graph may be parameterized by their instance (e.g $(\texttt{toss}, X)$ instead of $(\texttt{toss}, 1)$). A set of constraints on

$$\begin{aligned} f_1(n) &= h_1(1,n) \\ h_1(i,n) &= \begin{cases} g_1(i,n) + (1-\widehat{f}_1)\cdot h_1(i+1,n) & \text{if } i<n \\ g_1(i,n) & \text{if } i=n \end{cases} \\ g_1(i,n) &= \pi \cdot f_2(i,n) \\ \widehat{f}_1 &= \pi \end{aligned}$$

$$\begin{aligned} f_2(i,n) &= \begin{cases} h_2(i+1,n) & \text{if } i<n \\ 0 & \text{otherwise} \end{cases} \\ h_2(j,n) &= \begin{cases} g_2 + (1-\widehat{f}_2)\cdot h_2(j+1,n) & \text{if } j<n \\ g_2 & \text{if } j=n \end{cases} \\ g_2 &= \pi \\ \widehat{f}_2 &= \pi \end{aligned}$$

Figure 2 Recurrences for computing probabilities for Example in Fig. 1.

those variables, specify the allowed groundings.

Note that the graph size is independent of the size of the population. Moreover, the graph can be constructed in time independent of the population size as well. Probability computation is performed by first deriving recurrences based on the graph's structure and then solving the recurrences. The recurrences for probability computation derived from the graph in Fig. 1(c) are shown in Fig. 2. In the figure, the equations with subscript 1 are derived from the root of the graph; those with subscript 2 from the left child of the root; and where π is the probability that `toss` is "`h`". Note that the probability of the query, $f_1(n)$, can be computed in $\Theta(n)$ time from the recurrences.

Contributions. The technical contribution of this paper is two fold.

1. We define a lifted explanation structure, and operations over these structures (see Section 3). We also give method to construct such structures during query evaluation, closely following the techniques used to construct explanation graphs.
2. We define a technique to compute probabilities over such structures by deriving and solving recurrences (see Section 4). We provide examples to illustrate the complexity gains due to our technique over traditional inference.

The rest of the paper begins by defining parameterized Px programs and their semantics (Section 2). After presenting the main technical work, the paper concludes with a discussion of related work. (Section 5).

2 Parameterized Px Programs

The PRISM language follows Prolog's syntax. It adds a binary predicate `msw` to introduce random variables into an otherwise familiar Prolog program. Specifically, in `msw(`s`, `v`)`, s is a "switch" that represents a random process which generates a family of random variables, and v is bound to the value of a variable in that family. The domain and distribution of the switches are specified by `set_sw` directives. Given a switch s, we use D_s to denote the domain of s, and $\pi_s : D_s \to [0,1]$ to denote its probability distribution.

2.1 Px and Inference

The Px language extends the PRISM language in three ways. Firstly, the `msw` switches in Px are ternary, with the addition of an explicit *instance* parameter. This brings the language closer to the formalism presented when describing PRISM's semantics [13]. Secondly, Px aims to compute the distribution semantics without the *mutual exclusion* and *independence* assumptions on explanations imposed by PRISM system. Thirdly, in contrast to PRISM, the switches in Px can be defined with a wide variety of univariate distributions, including continuous distributions (such as Gaussian) and infinite discrete distributions (such as Poisson). However, in this paper, we consider only programs with finite discrete distributions.

Exact inference of Px programs with finite discrete distributions uses explanation graphs with the following structure.

▶ **Definition 1** (Ground Explanation Graph). Let S be the set of ground switches in a Px program P, and D_s be the domain of switch $s \in S$. Let $\mathcal{T}$ be the set of all ground terms over symbols in P. Let "$\prec$" be a total order over $S \times \mathcal{T}$ such that $(s_1, t_1) \prec (s_2, t_2)$ if either $t_1 < t_2$ or $t_1 = t_2$ and $s_1 < s_2$. A *ground explanation tree* over P is a rooted tree γ such that:

- Leaves in γ are labeled 0 or 1.
- Internal nodes in γ are labeled (s, z) where $s \in S$ is a switch, and z is a ground term over symbols in P.
- For node labeled (s, z), there are k outgoing edges to subtrees, where $k = |D_s|$. Each edge is labeled with a unique $v \in D_s$.
- Let $(s_1, z_1), (s_2, z_2), \ldots, (s_k, z_k), c$ be the sequence of node labels in a root-to-leaf path in the tree, where $c \in \{0, 1\}$. Then $(s_i, z_i) \prec (s_j, z_j)$ if $i < j$ for all $i, j \leq k$. As a corollary, node labels along any root to leaf path in the tree are unique.

An *explanation graph* is a DAG representation of a ground explanation tree.

Consider a sequence of alternating node and edge labels in a root-to-leaf path: $(s_1, z_1), v_1, (s_2, z_2), v_2, \ldots, (s_k, z_k), v_k, c$. Each such path enumerates a set of random variable valuations $\{s_1[z_1] = v_1, s_2[z_2] = v_2, \ldots, s_k[z_k] = v_k\}$. When $c = 1$, the set of valuations forms an explanation. An explanation graph thus represents a set of explanations.

Note that explanation trees and graphs resemble decision diagrams. Indeed, explanation graphs are implemented using Binary Decision Diagrams [3] in PITA and Problog; and Multi-Valued Decision Diagrams [15] in Px. The *union* of two sets of explanations can be seen as an "*or*" operation over corresponding explanation graphs. Pair-wise union of explanations in two sets is an "*and*" operation over corresponding explanation graphs.

2.1.1 Inference via Program Transformation

Inference in Px is performed analogous to that in PITA [11]. Concretely, inference is done by translating a Px program to one that explicitly constructs explanation graphs, performing tabled evaluation of the derived program, and computing probability of answers from the explanation graphs. We describe the translation for definite pure programs; programs with built-ins and other constructs can be translated in a similar manner.

For every user-defined atom A of the form $p(t_1, t_2, \ldots, t_n)$, we define $exp(A, E)$ as atom $p(t_1, t_2, \ldots, t_n, E)$ with a new predicate $p/(n+1)$, with E as an added "explanation" argument. For such atoms A, we also define $head(A, E)$ as atom $p'(t_1, t_2, \ldots, t_n, E)$ with a new predicate $p'/(n+1)$. A *goal* G is a conjunction of atoms, where $G = (G_1, G_2)$ for goals G_1 and G_2, or G is an atom A. Function *exp* is extended to goals such that $exp((G_1, G_2)) = ((exp(G_1, E_1), exp(G_2, E_2)), \texttt{and}(E_1, E_2, E))$, where `and` is a predicate in the translated program that combines two explanations using conjunction, and E_1 and E_2 are fresh variables. Function *exp* is also extended to `msw` atoms such that $exp(\texttt{msw}(p, i, v), E)$ is $\texttt{rv}(p, i, v, E)$, where `rv` is a predicate that binds E to an explanation graph with root labeled (p, i) with an edge labeled v leading to a 1 child, and all other edges leading to 0.

Each clause of the form $A \mathrel{:-} G$ in a Px program is translated to a new clause $head(A, E) \mathrel{:-} exp(G, E)$. For each predicate p/n, we define $p(X_1, X_2, \ldots X_n, E)$ to be such that E is the disjunction of all E' for $p'(X_1, X_2, \ldots X_n, E')$. As in PITA, this is done using answer subsumption.

Probability of an answer is determined by first materializing the explanation graph, and then computing the probability over the graph. The probability associated with a node in the graph is computed as the sum of the products of probabilities associated with its children and the corresponding edge probabilities. The probability associated with an explanation graph φ, denoted $prob(\varphi)$ is the probability associated with the root. This can be computed in time linear in the size of the graph by using dynamic programming or tabling.

2.2 Syntax and Semantics of Parameterized Px Programs

Parameterized Px, called Param-Px for short, is a further extension of the Px language. The first feature of this extension is the specification of *populations* and *instances* to specify ranges of instance parameters of `msw`s.

▶ **Definition 2** (Population). A *population* is a named finite set, with a specified cardinality. A population has the following properties:
1. Elements of a population may be atomic, or depth-bounded ground terms.
2. Elements of a population are totally ordered using the default term order.
3. Distinct populations are disjoint.

Populations and their cardinalities are specified in a Param-Px program by `population` facts. For example, the program in Figure 1(a) defines a population named `coins` of size 100. The individual elements of this set are left unspecified. When necessary, `element/2` facts may be used to define distinguished elements of a population. For example, `element(fred, persons)` defines a distinguished element "`fred`" in population persons. In presence of `element` facts, elements of a population are ordered as follows. The order of `element` facts specifies the order among the distinguished elements, and all distinguished elements occur before other unspecified elements in the order.

▶ **Definition 3** (Instance). An *instance* is an element of a population. In a Param-Px program, a built-in predicate `in/2` can be used to draw an instance from a population. All instances of a population can be drawn by backtracking over `in`.

An *instance variable* is one that occurs as the instance argument in a `msw` predicate in a clause of a Param-Px program. In Fig. 1(a), `X in coins` binds `X` to an instance of population `coins` and `X`, `Y` are instance variables.

Constraints. The second extension in Param-Px are atomic constraints, of the form $\{t_1 = t_2\}$, $\{t_1 \neq t_2\}$ and $\{t_1 < t_2\}$, where t_1 and t_2 are variables or constants, to compare instances of a population. We use braces "$\{\cdot\}$" to distinguish the constraints from Prolog built-in comparison operators. In Figure 1(a), `{X \= Y}` is an atomic constraint.

Types. We use populations in a Param-Px program to confer types to program variables. Each variable that occurs in an "`in`" predicate is assigned a unique type. More specifically, X has type p if X `in` p occurs in a program, where p is a population; and X is untyped otherwise. We extend this notion of types to constants and switches as well. A constant c has type p if there is a fact `element(`c`,` p`)`; and c is untyped otherwise. A switch s has type p if there is an `msw(`s`,` X`,` t`)` in the program and X has type p; and s is untyped otherwise.

▶ **Definition 4** (Well-typedness). A Param-Px program is *well-typed* if:
1. For every constraint in the program of the form $\{t_1 = t_2\}$, $\{t_1 \neq t_2\}$ or $\{t_1 < t_2\}$, the types of t_1 and t_2 are identical.
2. Types of arguments of every atom on the r.h.s. of a clause are identical to the types of corresponding parameters of l.h.s. atoms of matching clauses.
3. Every switch in the program has a unique type.

The first two conditions of well-typedness ensure that only instances from the same population are compared in the program. The last condition imposes that instances of random variables generated by switch s are all indexed by elements drawn from the same population. In the rest of the paper, unless otherwise specified, we assume all Param-Px programs under consideration are well-typed.

Semantics of Param-Px Programs. Each Param-Px program can be readily transformed into a non-parameterized "ordinary" Px program. Each `population` fact is used to generate a set of `in/2` facts enumerating the elements of the population. Other constraints are replaced by their counterparts is Prolog: e.g. $\{X < Y\}$ with $X{<}Y$. Finally, each `msw(`s`,`i`,`t`)` is preceded by i `in` p where p is the type of s. The semantics of the original parameterized program is defined by the semantics of the transformed program.

3 Lifted Explanations

In this section we formally define *lifted explanation graphs*. These are a generalization of *ground explanation graphs* defined earlier, and are introduced in order to represent ground explanations compactly. Constraints over instances form a basic building block of lifted explanations and the following constraint language is used for the purpose.

3.1 Constraints on Instances

▶ **Definition 5** (Instance Constraints)**.** Let $\mathcal{V}$ be a set of instance variables, with subranges of integers as domains, such that m is the largest positive integer in the domain of any variable. Atomic constraints on instance variables are of one of the following two forms: $X < aY \pm k$, $X = aY \pm k$, where $X, Y \in \mathcal{V}$, $a \in 0, 1$, where k is a non-negative integer $\leq m + 1$. The language of constraints over bounded integer intervals, denoted by $\mathcal{L}(\mathcal{V}, m)$, is a set of formulae η, where η is a non-empty set of atomic constraints representing their conjunction.

Note that each formula in $\mathcal{L}(\mathcal{V}, m)$ is a convex region in $\mathbb{Z}^{|V|}$, and hence is closed under conjunction and existential quantification.

Let $vars(\eta)$ be the set of instance variables in an instance constraint η. A substitution $\sigma : vars(\eta) \to [1..m]$ that maps each variable to an element in its domain is a *solution* to η if each constraint in η is satisfied by the mapping. The set of all solutions of η is denoted by $[\![\eta]\!]$. The constraint formula η is unsatisfiable if $[\![\eta]\!] = \emptyset$. We say that $\eta \models \eta'$ if every $\sigma \in [\![\eta]\!]$ is a solution to η'.

Note also that instance constraints are a subclass of the well-known integer octagonal constraints [8] and can be represented canonically by difference bound matrices (DBMs) [18, 6], permitting efficient algorithms for conjunction and existential quantification. Given a constraint on n variables, a DBM is a $(n+1) \times (n+1)$ matrix with rows and columns indexed by variables (and a special "zero" row and column). For variables X and Y, the entry in cell (X, Y) of a DBM represents the upper bound on $X - Y$. For variable X, the value at cell $(X, 0)$ is X's upper bound and the value at cell $(0, X)$ is the negation of X's lower bound.

Geometrically, each entry in the DBM representing a η is a "face" of the region representing $[\![\eta]\!]$. Negation of an instance constraint η can be represented by a set of mutually exclusive instance constraints. Geometrically, this can be seen as the set of convex regions obtained by complementing the "faces" of the region representing $[\![\eta]\!]$. Note that when η has n variables, the number of instance constraints in $\neg\eta$ is bounded by the number of faces of $[\![\eta]\!]$, and hence by $O(n^2)$.

Let $\neg\eta$ represent the set of mutually exclusive instance constraints representing the negation of η. Then the disjunction of two instance constraints η and η' can be represented by the set of mutually exclusive instance constraints $(\eta \wedge \neg\eta') \cup (\eta' \wedge \neg\eta) \cup \{\eta \wedge \eta'\}$, where we overload $\wedge$ to represent the element-wise conjunction of an instance constraint with a set of constraints.

An existentially quantified formula of the form $\exists X.\eta$ can be represented by a DBM obtained by removing the rows and columns corresponding to X in the DBM representation of η. We denote this simple procedure to obtain $\exists X.\eta$ from η by $Q(X, \eta)$.

▶ **Definition 6** (Range)**.** Given a constraint formula $\eta \in \mathcal{L}(\mathcal{V}, m)$, and $X \in \mathit{vars}(\eta)$, let $\sigma_X(\eta) = \{v \mid \sigma \in [\![\eta]\!], \sigma(X) = v\}$. Then $\mathit{range}(X, \eta)$ is the interval $[l, u]$, where $l = \mathit{min}(\sigma_X(\eta))$ and $u = \mathit{max}(\sigma_X(\eta))$.

Since the constraint formulas represent convex regions, it follows that each variable's range will be an interval. Note that range of a variable can be readily obtained in constant time from the entries for that variable in the zero row and zero column of the constraint's DBM representation.

3.2 Lifted Explanation Graphs

▶ **Definition 7** (Lifted Explanation Graph)**.** Let S be the set of ground switches in a Param-Px program P, D_s be the domain of switch $s \in S$, m be the sum of the cardinalities of all populations in P and C be the set of distinguished elements of the populations in P. A *lifted explanation graph* over variables $\mathcal{V}$ is a pair $(\Omega : \eta, \psi)$ which satisfies the following conditions

1. $\Omega : \eta$ is the notation for $\exists\Omega.\eta$, where $\eta \in \mathcal{L}(\mathcal{V}, m)$ is either a satisfiable constraint formula, or the single atomic constraint `false` and $\Omega \subseteq \mathit{vars}(\eta)$ is the set of quantified variables in η. When η is `false`, $\Omega = \emptyset$.
2. ψ is a singly rooted DAG which satisfies the following conditions
   - Internal nodes are labeled (s, t) where $s \in S$ and $t \in \mathcal{V} \cup C$.
   - Leaves are labeled either 0 or 1.
   - Each internal node has an outgoing edge for each outcome $\in D_s$.
   - If a node labeled (s, t) has a child labeled (s', t') then $\eta \models t < t'$ or $\eta \models t = t'$ and $(s, c) \prec (s', c)$ for any ground term c (see Def. 1).

In this paper ground explanation graphs (Def. 1), and the DAG components of lifted explanation graphs are represented by textual patterns $(s, t)[\alpha_i : \psi_i]$ where (s, t) is the label of the root and ψ_i is the DAG associated with the edge labeled α_i. Irrelevant parts may denoted "_" to reduce clutter. We define the standard notion of bound and free variables over lifted explanation graphs.

▶ **Definition 8** (Bound and free variables)**.** Given a lifted explanation graph $(\Omega : \eta, \psi)$, a variable $X \in \mathit{vars}(\eta)$, is called a bound variable if $X \in \Omega$, otherwise its called a free variable.

The lifted explanation graph is said to be *well-structured* if every pair of nodes (s, X) and (s', X) with the same bound variable X, have a common ancestor with X as the instance variable. In the rest of the paper, we assume that the lifted explanation graphs are well-structured.

▶ **Definition 9** (Substitution operation). Given a lifted explanation graph $(\Omega : \eta, \psi)$, a variable $X \in vars(\eta)$, the substitution of X in the lifted explanation graph with a value k from its domain, denoted by $(\Omega : \eta, \psi)[k/X]$ is defined as follows: If $\eta[k/X]$ is unsatisfiable, then the result of the substitution is $(\emptyset : \{\texttt{false}\}, 0)$. If $\eta[k/X]$ is satisfiable, then $(\Omega : \eta, \psi)[k/X] = (\Omega \setminus \{X\} : \eta[k/X], \psi[k/X])$. The definition of $\psi[k/X]$ is as follows:

$$\begin{array}{ll|l} ((s,t)[\alpha_i : \psi_i])[k/X] = (s,k)[\alpha_i : \psi_i[k/X]], & \text{if } t = X & 0[k/X] = 0 \\ ((s,t)[\alpha_i : \psi_i])[k/X] = (s,t)[\alpha_i : \psi_i[k/X]], & \text{if } t \neq X & 1[k/X] = 1 \end{array}$$

The definition of substitution operation can be generalized to mappings on sets of variables in the obvious way.

▶ **Lemma 10** (Substitution lemma). *If $(\Omega : \eta, \psi)$ is a lifted explanation graph, and $X \in vars(\eta)$, then $(\Omega : \eta, \psi)[k/X]$ where k is a value in domain of X, is a lifted explanation graph.*

When a substitution $[k/X]$ is applied to a lifted explanation graph, and $\eta[k/X]$ is unsatisfiable, the result is $(\emptyset : \{\texttt{false}\}, 0)$ which is clearly a lifted explanation graph. When $\eta[k/X]$ is satisfiable, the variable is removed from Ω and occurrences of X in ψ are replaced by k. The resultant DAG clearly satisfies the conditions imposed by the Def. 7. Finally we note that a ground explanation graph ϕ (Def. 1) is a trivial lifted explanation graph $(\emptyset : \{\texttt{true}\}, \phi)$. This constitutes the informal proof of Lemma 10.

3.3 Semantics of Lifted Explanation Graphs

The meaning of a lifted explanation graph $(\Omega : \eta, \psi)$ is given by the ground explanation tree represented by it.

▶ **Definition 11** (Grounding). Let $(\Omega : \eta, \psi)$ be a closed lifted explanation graph, i.e., it has no free variables. Then the ground explanation tree represented by $(\Omega : \eta, \psi)$, denoted $Gr((\Omega : \eta, \psi))$, is given by the function $Gr(\Omega, \eta, \psi)$. When $[\![\eta]\!] = \emptyset$, then $Gr(\_, \eta, \_) = 0$. We consider the cases when $[\![\eta]\!] \neq \emptyset$. The grounding of leaves is defined as $Gr(\_, \_, 0) = 0$ and $Gr(\_, \_, 1) = 1$. When the instance argument of the root is a constant, grounding is defined as $Gr(\Omega, \eta, (s,t)[\alpha_i : \psi_i]) = (s,t)[\alpha_i : Gr(\Omega, \eta, \psi_i)]$. When the instance argument is a bound variable, the grounding is defined as $Gr(\Omega, \eta, (s,t)[\alpha_i : \psi_i]) \equiv \bigvee_{c \in range(t,\eta)} (s,c)[\alpha_i : Gr(\Omega \setminus \{t\}, \eta[c/t], \psi_i[c/t])]$.

In the above definition $\psi[c/t]$ represents the tree obtained by replacing every occurrence of t in the tree with c. The disjunct $(s,c)[\alpha_i : Gr(\Omega \setminus \{t\}, \eta[c/t], \psi_i[c/t])]$ in the above definition is denoted $\phi_{(s,c)}$ when the lifted explanation graph is clear from the context.

3.4 Operations on Lifted Explanation Graphs

And/Or Operations. Let $(\Omega : \eta, \psi)$ and $(\Omega' : \eta', \psi')$ be two lifted explanation graphs. We now define "$\wedge$" and "$\vee$" operations on them. The "$\wedge$" and "$\vee$" operations are carried out in two steps. First, the constraint formulas of the inputs are combined. However, the free variables in the operands may have *no known order* among them. Since, an arbitrary order cannot be imposed, the operations are defined in a *relational*, rather than functional form. We use the notation $(\Omega : \eta, \psi) \oplus (\Omega' : \eta', \psi') \rightarrow (\Omega'' : \eta'', \psi'')$ to denote that $(\Omega'' : \eta'', \psi'')$ is *a* result of $(\Omega : \eta, \psi) \oplus (\Omega' : \eta', \psi')$. When an operation returns multiple answers due to ambiguity on the order of free variables, the answers that are inconsistent with the final order are discarded. We assume that the variables in the two lifted explanation graphs are standardized apart such that the bound variables of $(\Omega : \eta, \psi)$ and $(\Omega' : \eta', \psi')$ are all distinct, and different from free variables of $(\Omega : \eta, \psi)$ and $(\Omega' : \eta', \psi')$. Let $\psi = (s,t)[\alpha_i : \psi_i]$ and $\psi' = (s', t')[\alpha'_i : \psi'_i]$.

Combining constraint formulae

$Q(\Omega, \eta) \wedge Q(\Omega', \eta')$ is unsatisfiable. Then the orders among free variables in η and η' are incompatible.

- The $\wedge$ operation is defined as $(\Omega : \eta, \psi) \wedge (\Omega' : \eta', \psi') \rightarrow (\emptyset : \{\texttt{false}\}, 0)$
- The $\vee$ operation simply returns the two inputs as outputs:

$$(\Omega : \eta, \psi) \vee (\Omega' : \eta', \psi') \rightarrow (\Omega : \eta, \psi)$$
$$(\Omega : \eta, \psi) \vee (\Omega' : \eta', \psi') \rightarrow (\Omega' : \eta', \psi')$$

$Q(\Omega, \eta) \wedge Q(\Omega', \eta')$ is satisfiable. The orders among free variables in η and η' are compatible

- The $\wedge$ operation is defined as $(\Omega : \eta, \psi) \wedge (\Omega' : \eta', \psi') \rightarrow (\Omega \cup \Omega' : \eta \wedge \eta', \psi \wedge \psi')$.
- The $\vee$ operation is defined as

$$(\Omega : \eta, \psi) \vee (\Omega' : \eta', \psi') \rightarrow (\Omega \cup \Omega' : \eta \wedge \neg\eta', \psi)$$
$$(\Omega : \eta, \psi) \vee (\Omega' : \eta', \psi') \rightarrow (\Omega \cup \Omega' : \eta' \wedge \neg\eta, \psi')$$
$$(\Omega : \eta, \psi) \vee (\Omega' : \eta', \psi') \rightarrow (\Omega \cup \Omega' : \eta \wedge \eta', \psi \vee \psi')$$

Combining DAGs. Now we describe $\wedge$ and $\vee$ operations on the two DAGs ψ and ψ' in the presence of a single constraint formula. The general form of the operation is $(\Omega : \eta, \psi \oplus \psi')$.

Base cases: The base cases are as follows (symmetric cases are defined analogously).

$(\Omega : \eta, 0 \vee \psi') \rightarrow (\Omega : \eta, \psi')$ | $(\Omega : \eta, 0 \wedge \psi') \rightarrow (\Omega : \eta, 0)$

$(\Omega : \eta, 1 \vee \psi') \rightarrow (\Omega : \eta, 1)$ | $(\Omega : \eta, 1 \wedge \psi') \rightarrow (\Omega : \eta, \psi')$

Recursion: When the base cases do not apply, we try to compare the roots of ψ and ψ'. The root nodes are compared as follows: We say $(s, t) = (s', t')$ if $\eta \models t = t'$ and $s = s'$, else $(s, t) < (s', t')$ (analogously $(s', t') < (s, t)$) if $\eta \models t < t'$ or $\eta \models t = t'$ and $(s, c) \prec (s', c)$ for any ground term c. If neither of these two relations hold, then the roots are not comparable and its denoted as $(s, t) \not\sim (s', t')$.

a. **$(s, t) < (s', t')$:** $(\Omega : \eta, \psi \oplus \psi') \rightarrow (\Omega : \eta, (s, t)[\alpha_i : \psi_i \oplus \psi'])$
b. **$(s', t') < (s, t)$:** $(\Omega : \eta, \psi \oplus \psi') \rightarrow (\Omega : \eta, (s', t')[\alpha'_i : \psi \oplus \psi'_i])$
c. **$(s, t) = (s', t')$:** $(\Omega : \eta, \psi \oplus \psi') \rightarrow (\Omega : \eta, (s, t)[\alpha_i : \psi_i \oplus \psi'_i])$
d. **$(s, t) \not\sim (s', t')$:** Operations depend on whether t, t' are free, bound or constant.

i. t is a free variable or a constant, and t' is a free variable (the symmetric case is analogous).

$$(\Omega : \eta, \psi \oplus \psi') \rightarrow (\Omega : \eta \wedge t < t', \psi \oplus \psi')$$
$$(\Omega : \eta, \psi \oplus \psi') \rightarrow (\Omega : \eta \wedge t = t', \psi \oplus \psi')$$
$$(\Omega : \eta, \psi \oplus \psi') \rightarrow (\Omega : \eta \wedge t' < t, \psi \oplus \psi')$$

ii. t is a free variable or a constant and t' is a bound variable $(\Omega : \eta, \psi \oplus \psi')$ is defined as (the symmetric case is analogous):

$$(\Omega : \eta \wedge t < t', \psi \oplus \psi') \vee (\Omega : \eta \wedge t = t', \psi \oplus \psi') \vee (\Omega : \eta \wedge t' < t, \psi \oplus \psi')$$

Note that in the above definition, all three lifted explanation graphs use the same variable names for bound variable t'. Lifted explanation graphs can be easily standardized apart on the fly, and henceforth we assume that the operation is applied as and when required.

iii. t and t' are bound variables. Let $range(t, \eta) = [l_1, u_1]$ and $range(t', \eta) = [l_2, u_2]$. We can conclude that $range(t, \eta)$ and $range(t', \eta)$ are overlapping, otherwise (s, t) and (s', t') could have been ordered. Without loss of generality, we assume that $l_1 \leq l_2$. The various cases of overlap and the corresponding definition of the $(\Omega : \eta, \psi \oplus \psi')$ is given in the following table.

$l_1 = l_2, u_1 = u2$	$(\Omega \cup \{t''\} : \eta \wedge l_1 - 1 < t'' \wedge t'' - 1 < u_1 \wedge t'' < t \wedge t'' < t', (s, t'')[\alpha_i : (\psi_i[t''/t] \oplus \psi'_i[t''/t']) \vee (\psi_i[t''/t] \oplus \psi') \vee (\psi'_i[t''/t'] \oplus \psi)])$
$l_1 = l_2, u_1 < u_2$	$(\Omega : \eta \wedge t' - 1 < u_1, \psi \oplus \psi') \vee (\Omega : \eta \wedge u_1 < t', \psi \oplus \psi')$
$l_1 = l_2, u_2 < u_1$	$(\Omega : \eta \wedge t = t', \psi \oplus \psi') \vee (\Omega : \eta \wedge u_2 < t, \psi \oplus \psi')$
$l_1 < l_2, u_1 = u_2$	$(\Omega : \eta \wedge t = t', \psi \oplus \psi') \vee (\Omega : \eta \wedge t < l_2, \psi \oplus \psi')$
$l_1 < l_2, u_1 < u_2$	$(\Omega : \eta \wedge u_1 < t', \psi \oplus \psi') \vee (\Omega : \eta \wedge t < l_2 \wedge t' - 1 < u_1, \psi \oplus \psi') \vee (\Omega : \eta \wedge t = t', \psi \oplus \psi')$
$l_1 < l_2, u_2 < u_1$	$(\Omega : \eta \wedge u_2 < t, \psi \oplus \psi') \vee (\Omega : \eta \wedge t < l_2, \psi \oplus \psi') \vee (\Omega : \eta \wedge t = t', \psi \oplus \psi')$

▶ **Lemma 12** (Correctness of "$\wedge$" and "$\vee$" operations). *Let $(\Omega : \eta, \psi)$ and $(\Omega' : \eta', \psi')$ be two lifted explanation graphs with free variables $\{X_1, X_2 \ldots, X_n\}$. Let Σ be the set of all substitutions mapping each X_i to a value in its domain. Then, for every $\sigma \in \Sigma$ and $\oplus \in \{\wedge, \vee\}$, $Gr(((\Omega : \eta, \psi) \oplus (\Omega' : \eta', \psi'))\sigma) = Gr((\Omega : \eta, \psi)\sigma) \oplus Gr((\Omega' : \eta', \psi')\sigma)$*

Quantification

▶ **Definition 13** (Quantification). Let $(\Omega : \eta, \psi)$ be a lifted inference graph and $X \in vars(\eta)$. Then $quantify((\Omega : \eta, \psi), X) = (\Omega \cup \{X\} : \eta, \psi)$.

▶ **Lemma 14** (Correctness of *quantify*). *Let $(\Omega : \eta, \psi)$ be a lifted explanation graph, let σ_{-X} be a substitution mapping all the free variables in $(\Omega : \eta, \psi)$ except X to values in their domains. Let Σ be the set of mappings σ such that σ maps all free variables to values in their domains and is identical to σ_{-X} at all variables except X. Then the following holds $Gr(quantify((\Omega : \eta, \psi), X)\sigma_{-X}) = \bigvee_{\sigma \in \Sigma} Gr((\Omega : \eta, \psi)\sigma)$*

Construction of Lifted Explanation Graphs. Lifted explanation graphs for a query are constructed by transforming the Param-Px program $\mathcal{P}$ into one that explicitly constructs a lifted explanation graph, following a similar procedure to the one outlined in Section 2 for constructing ground explanation graphs. The main difference is the use of existential quantification. Let A :– G be a program clause, and $vars(G) - vars(A)$ be the set of variables in G and not in A. If any of these variables has a type, then it means that the variable used as an instance argument in G is existentially quantified. Such clauses are then translated as $head(A, E_h)$:– $exp(G, E_g), quantify(E_g, V_s, E_h)$, where V_s is the set of typed variables in $vars(G) - vars(A)$. A minor difference is the treatment of constraints: *exp* is extended to atomic constraints φ such that $exp(\varphi, E)$ binds E to $(\emptyset : \{\varphi\}, 1)$.

We order the populations and map the elements of the populations to natural numbers as follows. The population that comes first in the order is mapped to natural numbers in the range $1..m$, where m is the cardinality of this population. Any constants in this population are mapped to natural numbers in the low end of the range. The next population in the order is mapped to natural numbers starting from $m + 1$ and so on. Thus, each typed variable is assigned a domain of contiguous positive values. The rest of the program transformation remains the same, the underlying graphs are constructed using the lifted operators. The lifted explanation graph corresponding to the query in Fig 1(a) is shown in Fig 1(c).

4 Lifted Inference using Lifted Explanations

In this section we describe a technique to compute answer probabilities in a lifted fashion from closed lifted explanation graphs. This technique works on a restricted class of lifted explanation graphs satisfying a property we call the *frontier subsumption property*.

▶ **Definition 15** (Frontier). Given a closed lifted explanation graph $(\Omega : \eta, \psi)$, the frontier of ψ w.r.t $X \in \Omega$ denoted $\mathit{frontier}_X(\psi)$ is the set of non-zero maximal subtrees of ψ, which do not contain a node with X as the instance variable.

Analogous to the set representation of explanations described in Section 2.1, we consider the set representations of lifted explanations, i.e., root-to-leaf paths in the DAGs of lifted explanation graphs that end in a "1" leaf. We consider *term substitutions* that can be applied to lifted explanations. These substitutions replace a variable by a term and further apply standard re-writing rules such as simplification of algebraic expressions. As before, we allow *term mappings* that specify a set of *term substitutions*.

▶ **Definition 16** (Frontier subsumption property). A closed lifted explanation graph $(\Omega : \eta, \psi)$ satisfies the frontier subsumption property w.r.t $X \in \Omega$, if under term mappings $\sigma_1 = \{X \pm k + 1/Y \mid \langle X \pm k < Y \rangle \in \eta\}$ and $\sigma_2 = \{X + 1/X\}$, every tree $\phi \in \mathit{frontier}_X(\psi)$ satisfies the following condition: for every lifted explanation E_2 in ψ, there is a lifted explanation E_1 in ϕ such that $E_1\sigma_1$ is a sub-explanation (i.e., subset) of $E_2\sigma_2$.

A lifted explanation graph is said to satisfy frontier subsumption property, if it is satisfied for each bound variable. This property can be checked in a bottom up fashion for all bound variables in the graph. The tree obtained by replacing all subtrees in $\mathit{frontier}_X(\psi)$ by 1 in ψ is denoted $\widehat{\psi}_X$.

For closed lifted explanation graphs satisfying the above property, the probability of query answers can be computed using the following set of recurrences. With each subtree $\psi = (s,t)[\alpha_i : \psi_i]$ of the DAG of the lifted explanation graph, we associate function $f(\sigma, \psi)$ where σ is a (possibly incomplete) mapping of variables in Ω to values in their domains.

▶ **Definition 17** (Probability recurrences). Given a closed lifted explanation graph $(\Omega : \eta, \psi)$, we define $f(\sigma, \psi)$ (as well as $g(\sigma, \psi)$ and $h(\sigma, \psi)$ wherever applicable) for a partial mapping σ of variables in Ω to values in their domains based on the structure of ψ. As before $\psi = (s,t)[\alpha_i : \psi_i]$

Case 1: ψ is a 0 leaf node. Then $f(\sigma, 0) = 0$

Case 2: ψ is a 1 leaf node. Then $f(\sigma, 1) = \begin{cases} 1, & \text{if } [\![\eta\sigma]\!] \neq \emptyset \\ 0, & \text{otherwise} \end{cases}$

Case 3: $t\sigma$ is a constant. Then $f(\sigma, \psi) = \begin{cases} \sum_{\alpha_i \in D_s} \pi_s(\alpha_i) \cdot f(\sigma, \psi_i), & \text{if } [\![\eta\sigma]\!] \neq \emptyset \\ 0, & \text{otherwise} \end{cases}$

Case 4: $t\sigma \in \Omega$, and $\mathit{range}(t, \eta\sigma) = (l, u)$. Then

$$f(\sigma, \psi) = \begin{cases} h(\sigma[l/t], \psi), & \text{if } [\![\eta\sigma]\!] \neq \emptyset \\ 0, & \text{otherwise} \end{cases}$$

$$h(\sigma[c/t], \psi) = \begin{cases} g(\sigma[c/t], \psi) + ((1 - P(\widehat{\psi}_t)) \times h(\sigma[c+1/t], \psi)), & \text{if } c < u \\ g(\sigma[c/t], \psi), & \text{if } c = u \end{cases}$$

$$g(\sigma, \psi) = \begin{cases} \sum_{\alpha_i \in D_s} \pi_s(\alpha_i) \cdot f(\sigma, \psi_i), & \text{if } [\![\eta\sigma]\!] \neq \emptyset \\ 0, & \text{otherwise} \end{cases}$$

In the above definition $\sigma[c/t]$ refers to a new partial mapping obtained by augmenting σ with the substitution $[c/t]$, $P(\widehat{\psi_t})$ is the sum of the probabilities of all branches leading to a 1 leaf in $\widehat{\psi_t}$. The functions f, g and h defined above can be readily specialized for each ψ. Moreover, the parameter σ can be replaced by the tuple of values actually used by a function. These rewriting steps yield recurrences such as those shown in Fig. 2. Note that $P(\widehat{\psi_t})$ can be computed using recurrences as well (shown as $\widehat{f}$ in Fig. 2).

▶ **Definition 18** (Probability of Lifted Explanation Graph). Let $(\Omega : \eta, \psi)$ be a closed lifted explanation graph. Then, the probability of explanations represented by the graph, $prob((\Omega : \eta, \psi))$, is the value of $f(\{\}, \psi)$.

▶ **Theorem 19** (Correctness of Lifted Inference). *Let $(\Omega : \eta, \psi)$ be a closed lifted explanation graph, and $\phi = Gr(\Omega : \eta, \psi)$ be the corresponding ground explanation graph. Then $prob((\Omega : \eta, \psi)) = prob(\phi)$.*

Given a closed lifted explanation graph, let k be the maximum number of instance variables along any root to leaf path. Then the function $f(\sigma, \psi)$ for the leaf will have to be computed for each mapping of the k variables. Each recurrence equation itself is either of constant size or bounded by the number of children of a node. Using dynamic programming a solution to the recurrence equations can be computed in polynomial time.

▶ **Theorem 20** (Efficiency of Lifted Inference). *Let ψ be a closed lifted inference graph, n be the size of the largest population, and k be the largest number of instance variables along any root of leaf path in ψ. Then, $f(\{\}, \psi)$ can be computed in $O(|\psi| \times n^k)$ time.*

There are two sources of further optimization in the generation and evaluation of recurrences. First, certain recurrences may be transformed into closed form formulae which can be more efficiently evaluated. For instance, the closed form formula for $h(\sigma, \psi)$ for the subtree rooted at the node $(toss, Y)$ in Fig 1(c) can be evaluated in $O(\log(n))$ time while a naive evaluation of the recurrence takes $O(n)$ time. Second, certain functions $f(\sigma, \psi)$ need not be evaluated for every mapping σ because they may be independent of certain variables. For example, leaves are always independent of the mapping σ.

Other Examples. There are a number of simple probabilistic models that cannot be tackled by other lifted inference techniques but can be encoded in Param-Px and solved using our technique. For one such example, consider an urn with n balls, where the color of each ball is given by a distribution. Determining the probability that there are at least two green balls is easy to phrase as a directed first-order graphical model. However, lifted inference over such models can no longer be applied if we need to determine the probability of at least two green or two red balls. The probability computation for one of these events can be viewed as a generalization of noisy-OR probability computation, however dealing with the union requires the handling of intersection of the two events, due to which the $O(\log(N))$ time computation is no longer feasible.

For a more complex example, consider a system of n agents where each agent moves between various states in a stochastic manner. Consider a query to evaluate whether there are at least k agents in a given state s at a given time t. While this model is similar to a *collective graphical model* the aggregate query is different from those considered in [14], where computing probability of observed aggregate counts, parametering learning of individual model, and multiple path reconstruction are considered. Note that we cannot compile a model of this system into a clausal form without knowing the query. This system can be represented as a PRISM/Px program by modeling each agent's evolution as an independent

Hidden Markov Model (HMM). The lifted inference graph for querying the state of an arbitrary agent at time t is of size $O(e \cdot t)$, where e is the size of the transition relation of the HMM. For the "at least k agents" query, note that nodes in the lifted inference graph will be grouped by instances first, and hence the size of the graph (and the number of terms in the recurrences) is $O(k \cdot e \cdot t)$. The time complexity of evaluating the recurrences is $O(n \cdot k \cdot e \cdot t)$ where n is the total number of agents.

5 Related Work and Discussion

First-order graphical models [10, 2] are compact representations of propositional graphical models over populations. The key concepts in this field are that of *parameterized random variables* and *parfactors*. A parameterized random variable stands for a population of i.i.d. propositional random variables (obtained by grounding the logical variables). Parfactors are factors (potential functions) on parameterized random variables. By allowing large number of identical factors to be specified in a first-order fashion, first-order graphical models provide a representation that is independent of the population size. A key problem, then, is to perform *lifted* probabilistic inference over these models, i.e. without grounding the factors unnecessarily. The earliest such technique was *inversion elimination* presented in [10]. When summing out a parameterized random variable (i.e., all its groundings), it is observed that if all the logical variables in a parfactor are contained in the parameterized random variable, it can be summed out without grounding the parfactor.

The idea of *inversion elimination*, though powerful, exploits one of the many forms of symmetry present in first-order graphical models. Another kind of symmetry present in such models is that the values of an intermediate factor may depend on the histogram of propositional random variable outcomes, rather than their exact assignment. This symmetry is exploited by *counting elimination* [2] and elimination by *counting formulas* [7].

In [17] a form of lifted inference that uses constrained CNF theories with positive and negative weight functions over predicates as input was presented. Here the task of probabilistic inference in transformed to one of weighted model counting. To do the latter, the CNF theory is compiled into a structure known as first-order deterministic decomposable negation normal form. The compiled representation allows lifted inference by avoiding grounding of the input theory. This technique is applicable so long as the model can be formulated as a constrained CNF theory.

Another approach to lifted inference for probabilistic logic programs was presented in [1]. The idea is to convert a ProbLog program to parfactor representation and use a modified version of generalized counting first order variable elimination algorithm [16] to perform lifted inference. Problems where the model size is dependent on the query, such as models with temporal aspects, are difficult to solve with the knowledge compilation approach.

In this paper, we presented a technique for lifted inference in probabilistic logic programs using lifted explanation graphs. This technique is a natural generalization of inference techniques based on ground explanation graphs, and follows the two step approach: generation of an explanation graph, and a subsequent traversal to compute probabilities. A more complete description of this technique is in [9]. While the size of the lifted explanation graph is often independent of population, computation of probabilities may take time that is polynomial in the size of the population. A more sophisticated approach to computing probabilities from lifted explanation graph, by generating closed form formulae where possible, will enable efficient inference. Another direction of research would be to generate hints for lifted inference based on program constructs such as aggregation operators. Finally, our future work is focused on performing lifted inference over probabilistic logic programs that represent undirected and discriminative models.

Acknowledgments. We thank Andrey Gorlin for discussions and review of this work.

References

1 Elena Bellodi, Evelina Lamma, Fabrizio Riguzzi, Vítor Santos Costa, and Riccardo Zese. Lifted variable elimination for probabilistic logic programming. *TPLP*, 14(4-5):681–695, 2014.

2 Rodrigo De Salvo Braz, Eyal Amir, and Dan Roth. Lifted first-order probabilistic inference. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence*, pages 1319–1325, 2005.

3 Randal E Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys (CSUR)*, 24(3):293–318, 1992.

4 Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. ProbLog: A probabilistic prolog and its application in link discovery. In *IJCAI*, pages 2462–2467, 2007.

5 Anton Dries, Angelika Kimmig, Wannes Meert, Joris Renkens, Guy Van den Broeck, Jonas Vlasselaer, and Luc De Raedt. Problog2: Probabilistic logic programming. In *ECML PKDD*, pages 312–315, 2015.

6 Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Efficient verification of real-time systems: Compact data structure and state-space reduction. In *IEEE RTSS'97*, pages 14–24, 1997.

7 Brian Milch, Luke S Zettlemoyer, Kristian Kersting, Michael Haimes, and Leslie Pack Kaelbling. Lifted probabilistic inference with counting formulas. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence*, pages 1062–1068, 2008.

8 Antoine Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

9 Arun Nampally and C. R. Ramakrishnan. Inference in probabilistic logic programs using lifted explanations. Technical report, Computer Science Department, Stony Brook University, 2016. URL: `http://www.cs.stonybrook.edu/~px/Papers/NR:lifted_tr_2016/`.

10 David Poole. First-order probabilistic inference. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, volume 3, pages 985–991, 2003.

11 Fabrizio Riguzzi and Terrance Swift. The PITA system: Tabling and answer subsumption for reasoning under uncertainty. *TPLP*, 11(4-5):433–449, 2011.

12 Taisuke Sato and Yoshitaka Kameya. PRISM: a language for symbolic-statistical modeling. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, volume 97, pages 1330–1339, 1997.

13 Taisuke Sato and Yoshitaka Kameya. Parameter learning of logic programs for symbolic-statistical modeling. *Journal of Artificial Intelligence Research*, pages 391–454, 2001.

14 Daniel R Sheldon and Thomas G. Dietterich. Collective graphical models. In *Advances in Neural Information Processing Systems 24*, pages 1161–1169, 2011. URL: `http://papers.nips.cc/paper/4220-collective-graphical-models.pdf`.

15 Arvind Srinivasan, Timothy Kam, Sharad Malik, and Robert K. Brayton. Algorithms for discrete function manipulation. In *International Conference on Computer-Aided Design, ICCAD*, pages 92–95, 1990. `doi:10.1109/ICCAD.1990.129849`.

16 Nima Taghipour, Daan Fierens, Jesse Davis, and Hendrik Blockeel. Lifted variable elimination: Decoupling the operators from the constraint language. *J. Artif. Intell. Res. (JAIR)*, 47:393–439, 2013.

17 Guy Van den Broeck, Nima Taghipour, Wannes Meert, Jesse Davis, and Luc De Raedt. Lifted probabilistic inference by first-order knowledge compilation. In *IJCAI*, pages 2178–2185, 2011.

18 Sergio Yovine. Model-checking timed automata. In *Embedded Systems*, number 1494 in LNCS, pages 114–152, 1998.

On the Expressiveness of Spatial Constraint Systems*

Michell Guzmán[1] and Frank D. Valencia[2]

1 INRIA-LIX École Polytechnique, Paris, France
guzman@lix.polytechnique.fr
2 CNRS-LIX École Polytechnique, Paris, France; and
Pontificia Universidad Javeriana, Cali, Colombia
frank.valencia@lix.polytechnique.fr

Abstract

In this paper we shall report on our progress using spatial constraint system as an abstract representation of modal and epistemic behaviour. First we shall give an introduction as well as the background to our work. Then, we present our preliminary results on the representation of modal behaviour by using spatial constraint systems. Then, we present our ongoing work on the characterization of the epistemic notion of knowledge. Finally, we discuss about the future work of our research.

1998 ACM Subject Classification F.4.1 Mathematical Logic

Keywords and phrases Epistemic logic, Modal logic, Constraint systems, Concurrent constraint programming.

Digital Object Identifier 10.4230/OASIcs.ICLP.2016.16

1 Introduction

Epistemic, mobile and spatial behaviour are common practice in today's distributed systems. The intrinsic *epistemic* nature of these systems arises from social behaviour. Most people are familiar with digital systems where users share their *beliefs*, *opinions* and even intentional *lies* (hoaxes). Also, systems modeling decision behaviour must account for those decisions dependance on the results of interactions with others within some social context. Spatial and mobile behaviour is exhibited by applications and data moving across (possibly nested) spaces defined by, for example, friend circles, groups, and shared folders. We therefore believe that a solid understanding of the notion of *space* and *spatial mobility* as well as the flow of epistemic information is relevant in many models of today's distributed systems.

Constraint systems (cs's) provide the basic domains and operations for the semantic foundations of the family of *formal declarative models* from concurrency theory known as *concurrent constraint programming (ccp)* process calculi [15]. *Spatial constraint systems* [9] (scs) are algebraic structures that extend cs for reasoning about basic spatial and epistemic behaviour such as *extrusion* and *belief*. Both spatial and epistemic assertions can be viewed as specific *modalities*. Other modalities can be used for assertions about time, knowledge and other concepts used in the specification and verification of concurrent systems.

The main goal of this PhD project is the study of the expressiveness of spatial constraint systems in the broader perspective of modal behaviour. In this summary, we shall show that

* This work was partially supported by INRIA-CORDI.

spatial constraint systems are sufficiently robust to capture other modalities and to derive new results for modal logic. We shall also discuss our future work on extending constraint systems to express fundamental epistemic behaviour such as *knowledge* and *distributed knowledge*.

This summary is structured as follows: In Section 2 we give some background. In Section 3 we present our results with applications to modal logic. In Sections 4 and 5 we describe ongoing work and future work for the remaining part of this PhD project. The results in this summary have been recently published as [7, 6].

2 Background

In this section we recall the notion of basic constraint system [3] and the more recent notion of spatial constraint system [9]. We presuppose basic knowledge of order theory and modal logic [1, 14, 5, 2].

The concurrent constraint programming model of computation [15] is parametric in a *constraint system* (cs) specifying the structure and interdependencies of the partial information that computational agents can ask of and post in a *shared store*. This information is represented as *assertions* traditionally referred to as *constraints*.

Constraint systems can be formalized as *complete algebraic lattices* [3][1]. The elements of the lattice, the *constraints*, represent (partial) information. A constraint c can be viewed as an *assertion* (or a *proposition*). The lattice order $\sqsubseteq$ is meant to capture entailment of information: $c \sqsubseteq d$, alternatively written $d \sqsupseteq c$, means that the assertion d represents as much information as c. Thus, we may think of $c \sqsubseteq d$ as saying that d *entails* c or that c can be *derived* from d. The *least upper bound (lub)* operator $\sqcup$ represents join of information; $c \sqcup d$, the least element in the underlying lattice above c and d. Thus, $c \sqcup d$ can be seen as an assertion stating that both c and d hold. The top element represents the lub of all, possibly inconsistent, information, hence it is referred to as *false*. The bottom element *true* represents the empty information.

▶ **Definition 1** (Constraint Systems [3])**.** A constraint system (cs) $\mathbf{C}$ is a complete algebraic lattice $(Con, \sqsubseteq)$. The elements of Con are called *constraints*. The symbols $\sqcup$, *true* and *false* will represent the least upper bound (lub) operation, the bottom, and the top element of $\mathbf{C}$, respectively.

We shall use the following notions and notations from order theory.

▶ **Notation 2** (Lattices and Limit Preservation)**.** *Let $\mathbf{C}$ be a partially ordered set (poset) $(Con, \sqsubseteq)$. We shall use $\bigsqcup S$ to denote the least upper bound (lub) (or* supremum *or* join*) of the elements in S, and $\bigsqcap S$ is the greatest lower bound (glb) (*infimum *or* meet*) of the elements in S. We say that $\mathbf{C}$ is a* complete lattice *iff each subset of Con has a supremum and an infimum in Con. A non-empty set $S \subseteq Con$ is* directed *iff every* finite *subset of S has an upper bound in S. Also, $c \in Con$ is* compact *iff for any directed subset D of Con, $c \sqsubseteq \bigsqcup D$ implies $c \sqsubseteq d$ for some $d \in D$. A complete lattice $\mathbf{C}$ is said to be* algebraic *iff for each $c \in Con$, the set of compact elements below it forms a directed set and the lub of this directed set is c. A self-map on Con is a function $f : Con \to Con$. Let $(Con, \sqsubseteq)$ be a complete lattice. The self-map f on Con* preserves *the supremum of a set $S \subseteq Con$ iff $f(\bigsqcup S) = \bigsqcup\{f(c) \mid c \in S\}$. The preservation of the infimum of a set is defined analogously. We say f preserves finite/infinite suprema iff it preserves the supremum of arbitrary finite/infinite sets. Preservation of finite/infinite infima is defined similarly.*

[1] An alternative syntactic characterization of cs, akin to Scott information systems, is given in [15].

2.1 Spatial Constraint Systems

The authors of [9] extended the notion of cs to account for distributed and multi-agent scenarios where agents have their own space for their local information and performing their computations.

Intuitively, each agent i has a *space* function $[\cdot]_i$ from constraints to constraints. We can think of $[c]_i$ as an assertion stating that c is a piece of information residing *within a space attributed to agent i*. An alternative *epistemic logic* interpretation of $[c]_i$ is an assertion stating that agent i *believes* c or that c holds within the space of agent i (but it may not hold elsewhere). Similarly, $[[c]_j]_i$ is a hierarchical spatial specification stating that c holds within the local space the agent i attributes to agent j. Nesting of spaces can be of any depth. We can think of a constraint of the form $[c]_i \sqcup [d]_j$ as an assertion specifying that c and d hold within two *parallel/neighboring* spaces that belong to agents i and j, respectively.

▶ **Definition 3** (Spatial Constraint System [9])**.** An n-agent *spatial constraint system (n-scs)* $\mathbf{C}$ is a cs $(Con, \sqsubseteq)$ equipped with n self-maps $[\cdot]_1, \ldots, [\cdot]_n$ over its set of constraints Con such that:

(S.1) $[true]_i = true$, and

(S.2) $[c \sqcup d]_i = [c]_i \sqcup [d]_i$ for each $c, d \in Con$.

Axiom S.1 requires $[\cdot]_i$ to be strict map (i.e. bottom preserving). Intuitively, it states that having an empty local space amounts to nothing. Axiom S.2 states that the information in a given space can be distributed. Notice that requiring S.1 and S.2 is equivalent to requiring that each $[\cdot]_i$ preserves *finite suprema*. Also, S.2 implies that $[\cdot]_i$ is monotonic: I.e., if $c \sqsupseteq d$ then $[c]_i \sqsupseteq [d]_i$.

2.2 Extrusion and utterance

We can also equip each agent i with an *extrusion* function $\uparrow_i : Con \to Con$. Intuitively, within a space context $[\cdot]_i$, the assertion $\uparrow_i c$ specifies that c must be posted outside of (or extruded from) agent i's space. This is captured by requiring the *extrusion* axiom $[\uparrow_i c]_i = c$. In other words, we view *extrusion/utterance* as the right inverse of *space/belief* (and thus space/belief as the left inverse of extrusion/utterance).

▶ **Definition 4** (Extrusion)**.** Given an n-scs $(Con, \sqsubseteq, [\cdot]_1, \ldots, [\cdot]_n)$, we say that $\uparrow_i$ is an extrusion function for the space $[\cdot]_i$ iff $\uparrow_i$ is a right inverse of $[\cdot]_i$, i.e., iff $[\uparrow_i c]_i = c$.

2.3 The Extrusion/Right Inverse Problem

A legitimate question is: Given space $[\cdot]_i$ can we derive an extrusion function $\uparrow_i$ for it ? From set theory we know that there is an extrusion function (i.e., a right inverse) $\uparrow_i$ for $[\cdot]_i$ iff $[\cdot]_i$ is *surjective*. Recall that the *pre-image* of $y \in Y$ under $f : X \to Y$ is the set $f^{-1}(y) = \{x \in X \mid y = f(x)\}$. Thus, $\uparrow_i$ can be defined as a function, called *choice* function, that maps each element c to some element from the pre-image of c under $[\cdot]_i$.

3 Preliminary Results

In this part of the summary we shall describe the work we have achieved so far. It is based on the paper [7] recently accepted for publication.

3.1 Modalities in Terms of Space

Modal logics [14] extend classical logic to include operators expressing modalities. Depending on the intended meaning of the modalities, a particular modal logic can be used to reason about space, knowledge, belief or time, among others. Although the notion of spatial constraint system is intended to give an algebraic account of spatial and epistemic assertions, we shall show that it is sufficiently robust to give an algebraic account of more general modal assertions.

The aim of this part of the summary is the study of the *extrusion problem* for a meaningful family of scs's that can be used as semantic structures for modal logics. They are called *Kripke spatial constraint systems* because its elements are *Kripke Structures* (KS's). KS's can be seen as transition systems with some additional structure on their states.

3.2 Constraint Frames and Normal Self Maps

Spatial constraint systems can be used, by building upon ideas from Geometric Logic and Heyting Algebras [16], as semantic structures for modal logic. We shall give an algebraic characterization of the concept of normal modality as maps preserving finite suprema.

First, recall that a *Heyting implication* $c \rightarrow d$ in our setting corresponds to the *weakest constraint* one needs to join c with to derive d: The greatest lower bound (glb) $\bigsqcap\{e \mid e \sqcup c \sqsupseteq d\}$. Similarly, the negation of a constraint c, written $\sim c$, can be seen as the *weakest constraint inconsistent* with c, i.e., the glb $\bigsqcap\{e \mid e \sqcup c \sqsupseteq \mathit{false}\} = c \rightarrow \mathit{false}$.

▶ **Definition 5** (Constraint Frames). A constraint system $(\mathit{Con}, \sqsubseteq)$ is said to be a *constraint frame* iff its joins distribute over arbitrary meets: More precisely, $c \sqcup \bigsqcap S = \bigsqcap\{c \sqcup e \mid e \in S\}$ for every $c \in \mathit{Con}$ and $S \subseteq \mathit{Con}$. Given a constraint frame $(\mathit{Con}, \sqsubseteq)$ and $c, d \in \mathit{Con}$, define Heyting implication $c \rightarrow d$ as $\bigsqcap\{e \in \mathit{Con} \mid c \sqcup e \sqsupseteq d\}$ and Heyting negation $\sim c$ as $c \rightarrow \mathit{false}$.

In modal logics one is often interested in *normal modal* operators. The formulae of a modal logic are those of propositional logic extended with modal operators. Roughly speaking, a modal logic operator m is normal iff (1) the formula $\mathrm{m}(\phi)$ is a theorem (i.e., true in all models for the underlying modal language) whenever the formula ϕ is a theorem, and (2) the implication formula $\mathrm{m}(\phi \Rightarrow \psi) \Rightarrow (\mathrm{m}(\phi) \Rightarrow \mathrm{m}(\psi))$ is a theorem. Thus, using Heyting implication, we can express the normality condition in constraint frames as follows.

▶ **Definition 6** (Normal Maps). Let $(\mathit{Con}, \sqsubseteq)$ be a constraint frame. A self-map m on Con is said to be *normal* if (1) $m(\mathit{true}) = \mathit{true}$ and (2) $m(c \rightarrow d) \rightarrow (m(c) \rightarrow m(d)) = \mathit{true}$ for each $c, d \in \mathit{Con}$.

The next theorem basically states that Condition (2) in Definition 6 is equivalent to the seemingly simpler condition: $m(c \sqcup d) = m(c) \sqcup m(d)$.

▶ **Theorem 7** (Normality & Finite Suprema). *Let* $\mathbf{C}$ *be a constraint frame* $(\mathit{Con}, \sqsubseteq)$ *and let* f *be a self-map on* Con*. Then* f *is normal if and only if* f *preserves finite suprema.*

By applying the above theorem, we can conclude that space functions from constraint frames are indeed normal self-maps, since they preserve finite suprema.

3.3 Extrusion Problem for Kripke Constraint Systems

In this section we will study the extrusion/right inverse problem for a meaningful family of spatial constraint systems (scs's), the Kripke scs. In particular, we shall derive and give a *complete* characterization of normal extrusion functions as well as identify the *weakest* condition on the elements of the scs under which extrusion functions may exist. To illustrate the importance of this study, let us give some intuition first.

Kripke structures (KS) are a fundamental mathematical tool in logic and computer science. They can be seen as transition systems and they are used to give semantics to modal logics. Formally, a KS can be defined as follows.

▶ **Definition 8** (Kripke Structures)**.** An n-agent Kripke Structure (KS) M over a set of atomic propositions Φ is a tuple $(S, \pi, \mathcal{R}_1, \ldots, \mathcal{R}_n)$ where S is a nonempty set of states, $\pi : S \to (\Phi \to \{0,1\})$ is an interpretation associating with each state a truth assignment to the primitive propositions in Φ, and $\mathcal{R}_i$ is a binary relation on S. A *pointed KS* is a pair (M, s) where M is a KS and s, called the *actual world*, is a state of M. We write $s \xrightarrow{i}_M t$ to denote $(s,t) \in \mathcal{R}_i$.

We now define the Kripke scs wrt a set $\mathcal{S}_n(\Phi)$ of pointed KS.

▶ **Definition 9** (Kripke Spatial Constraint Systems [9])**.** Let $\mathcal{S}_n(\Phi)$ be a non-empty set of n-agent Kripke structures over a set of primitive propositions Φ. We define the Kripke n-scs for $\mathcal{S}_n(\Phi)$ as $\mathbf{K}(\mathcal{S}_n(\Phi)) = (Con, \sqsubseteq, [\cdot]_1, \ldots, [\cdot]_n)$ where $Con = \mathcal{P}(\Delta)$, $\sqsubseteq\, =\, \supseteq$, and

$$[c]_i \stackrel{\texttt{def}}{=} \{(M,s) \in \Delta \mid \rhd_i(M,s) \subseteq c\}$$

where Δ is the set of all pointed Kripke structures (M,s) such that $M \in \mathcal{S}_n(\Phi)$ and $\rhd_i(M,s) = \{(M,t) \mid s \xrightarrow{i}_M t\}$ denotes the pointed KS reachable from (M,s).

The structure $\mathbf{K}(\mathcal{S}_n(\Phi)) = (Con, \sqsubseteq, [\cdot]_1, \ldots, [\cdot]_n)$ is a complete algebraic lattice given by a powerset ordered by reversed inclusion $\supseteq$. The join $\sqcup$ is set intersection, the meet $\sqcap$ is set union, the top element *false* is the empty set $\emptyset$, and bottom *true* is the set Δ of all pointed Kripke structures (M,s) with $M \in \mathcal{S}_n(\Phi)$. Notice that $\mathbf{K}(\mathcal{S}_n(\Phi))$ is a frame since meets are unions and joins are intersections so the distributive requirement is satisfied. Furthermore, each $[\cdot]_i$ preserves arbitrary suprema (intersection) and thus, from Theorem 7 it is a normal self-map.

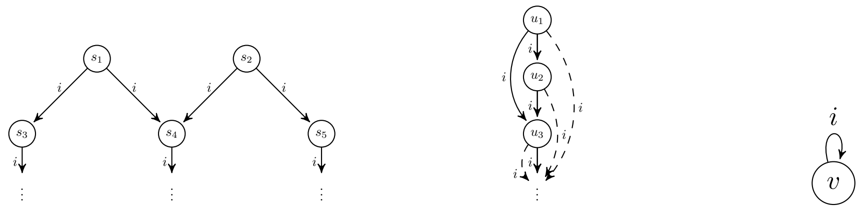
3.4 Existence of Right Inverses

We shall now address the question of whether a given Kripke constraint system can be extended with extrusion functions. We shall identify a sufficient and necessary condition on accessibility relations for the existence of an extrusion function $\uparrow_i$ given the space $[\cdot]_i$.

▶ **Definition 10** (Determinacy and Unique-Determinacy)**.** Let S and $\mathcal{R}$ be the set of states and an accessibility relation of a KS M, respectively. Given $s, t \in S$, we say that s *determines* t wrt $\mathcal{R}$ if $(s,t) \in \mathcal{R}$. We say that s *uniquely determines* t wrt $\mathcal{R}$ if s is the *only state* in S that determines t wrt $\mathcal{R}$. A state $s \in S$ is said to be *determinant* wrt $\mathcal{R}$ if it uniquely determines some state in S wrt $\mathcal{R}$. Furthermore, $\mathcal{R}$ is *determinant-complete* if every state in S is determinant wrt $\mathcal{R}$.

▶ **Example 11.** Figure 1 illustrates some determinant-complete accessibility relations. Figures 1.(i) and 1.(iii) are determinant-complete accessibility relations. Figure 1.(ii) shows a non determinant-complete accessibility relation (the transitive closure of an infinite line structure).

▶ **Notation 12.** *We write* $s \overset{i}{\twoheadrightarrow}_M t$ *for* s uniquely determines t *wrt* $\xrightarrow{i}_M$.



(a) M_1 **(b)** M_2 **(c)** M_3

Figure 1 Accessibility relations for an agent i. In each sub-figure we omit the corresponding KS M_k from the edges and draw $s \xrightarrow{i} t$ whenever $s \xrightarrow{i}_{M_k} t$.

The following theorem provides a complete characterization, in terms of classes of KS, of the existence of right inverses for space functions.

▶ **Theorem 13** (Completeness). *Let $[\cdot]_i$ be a spatial function of a Kripke scs $\mathbf{K}(\mathcal{S})$. Then $[\cdot]_i$ has a right inverse iff for every $M \in \mathcal{S}$ the accessibility relation $\xrightarrow{i}_M$ is determinant-complete.*

Henceforth we use $\mathcal{M}^{\mathsf{D}}$ to denote the class of KS's whose accessibility relations are determinant-complete. It follows from Theorem 13 that $\mathcal{S} = \mathcal{M}^{\mathsf{D}}$ is the largest class for which space functions of a Kripke scs $\mathbf{K}(\mathcal{S})$ have right inverses.

3.5 Right Inverse Constructions

Let $\mathbf{K}(\mathcal{S}) = (Con, \sqsubseteq, [\cdot]_1, \ldots, [\cdot]_n)$ be a Kripke scs. The Axiom of Choice and Theorem 13 tell us that each $[\cdot]_i$ has a right inverse (extrusion function) if and only if $\mathcal{S} \subseteq \mathcal{M}^{\mathsf{D}}$. We are interested, however, in explicit constructions of the right inverses.

Since any Kripke scs space function preserves arbitrary suprema, we obtain the following canonical greatest right-inverse construction. Recall that the pre-image of c under $[\cdot]_i$ is given by the set $[c]_i^{-1} = \{d \mid c = [d]_i\}$.

▶ **Definition 14** (Max Right Inverse). Let a Kripke scs $\mathbf{K}(\mathcal{S}) = (Con, \sqsubseteq, [\cdot]_1, \ldots, [\cdot]_n)$ be defined over $\mathcal{S} \subseteq \mathcal{M}^{\mathsf{D}}$. We define $\uparrow_i^{\mathsf{M}}$ as the following self-map on Con : $\uparrow_i^{\mathsf{M}} : c \mapsto \bigsqcup [c]_i^{-1}$.

Then $\uparrow_i^{\mathsf{M}}$ is a right inverse for $[\cdot]_i$, and from its definition it is clear that $\uparrow_i^{\mathsf{M}}$ is the greatest right inverse of $[\cdot]_i$ wrt $\sqsubseteq$. However, $\uparrow_i^{\mathsf{M}}$ is not necessarily *normal* in the sense of Definition 6. In what follows we shall identify right inverse constructions that are normal.

3.6 Normal Right Inverses

The following central lemma provides distinctive properties of any normal right-inverse.

▶ **Lemma 15.** *Let $\mathbf{K}(\mathcal{S}) = (Con, \sqsubseteq, [\cdot]_1, \ldots, [\cdot]_n)$ be the Kripke scs over $\mathcal{S} \subseteq \mathcal{M}^{D}$. Suppose that f is a normal right-inverse of $[\cdot]_i$. Then for every $M \in \mathcal{S}$, $c \in Con$:*
(i) $\rhd_i(M, s) \subseteq f(c)$ *if* $(M, s) \in c$,
(ii) $\{(M, t)\} \subseteq f(c)$ *if t is multiply determined wrt* $\xrightarrow{i}_M$, *and*
(iii) $true \subseteq f(true)$.

The above property tell us what sets should necessarily be included in every $f(c)$ if f is to be both normal and a right-inverse of $[\cdot]_i$.

In fact, the least self-map f wrt $\subseteq$, i.e., the greatest one wrt the lattice order $\sqsubseteq$, satisfying Conditions 1, 2 and 3 in Lemma 15 is indeed a normal right-inverse. We call such a function the *max normal right-inverse* $\uparrow_i^{\mathtt{MN}}$ and it is given below.

▶ **Definition 16** (Max Normal-Right Inverse). Let $\mathbf{K}(\mathcal{S}) = (Con, \sqsubseteq, [\cdot]_1, \ldots, [\cdot]_n)$ be a Kripke scs over $\mathcal{S} \subseteq \mathcal{M}^{\mathrm{D}}$. We define the *max normal right-inverse* for agent i, $\uparrow_i^{\mathtt{MN}}$ as the following self-map on Con:

$$\uparrow_i^{\mathtt{MN}}(c) \stackrel{\mathtt{def}}{=} \begin{cases} -true \ \text{ if } c = true \\ -\{(M,t) \mid t \text{ is determined wrt } \xrightarrow{i}_M \ \& \\ \forall s : s \overset{i}{\twoheadrightarrow}_M t, (M,s) \in c\} \end{cases} \tag{1}$$

Notice that $\uparrow_i^{\mathtt{MN}}(c)$ excludes indetermined states (i.e. a state t such that for every $s \in S, (s,t) \notin \mathcal{R}$.) if $c \neq true$. It turns out that we can add them and obtain a more succinct normal right-inverse:

▶ **Definition 17** (Normal Right-Inverse). Let $\mathbf{K}(\mathcal{S}) = (Con, \sqsubseteq, [\cdot]_1, \ldots, [\cdot]_n)$ be a Kripke scs over $\mathcal{S} \subseteq \mathcal{M}^{\mathrm{D}}$. Define $\uparrow_i^{\mathtt{N}} : Con \to Con$ as $\uparrow_i^{\mathtt{N}}(c) \stackrel{\mathtt{def}}{=} \{(M,t) \mid \forall s : s \overset{i}{\twoheadrightarrow}_M t, (M,s) \in c\}$.

Clearly $\uparrow_i^{\mathtt{N}}(c)$ includes every (M,t) such that t is indetermined wrt $\xrightarrow{i}_M$.

3.7 Applications

In this section we will illustrate and briefly discuss the results obtained in the previous section in the context of modal logic.

We can interpret modal formulae as constraints in a given Kripke scs $\mathbf{C} = \mathbf{K}(\mathcal{S}_n(\Phi))$.

▶ **Definition 18** (Kripke Constraint Interpretation). Let $\mathbf{C}$ be a Kripke scs $\mathbf{K}(\mathcal{S}_n(\Phi))$. Given a modal formula ϕ in the modal language $\mathcal{L}_n(\Phi)$, its interpretation in the Kripke scs $\mathbf{C}$ is the constraint $\mathbf{C}[\![\phi]\!]$ inductively defined as follows: $\mathbf{C}[\![p]\!] = \{(M,s) \mid \pi_M(s)(p) = 1\}$, $\mathbf{C}[\![\phi \wedge \psi]\!] = \mathbf{C}[\![\phi]\!] \sqcup \mathbf{C}[\![\psi]\!]$, $\mathbf{C}[\![\neg\phi]\!] = \sim \mathbf{C}[\![\phi]\!]$, $\mathbf{C}[\![\Box_i\phi]\!] = [\ \mathbf{C}[\![\phi]\!]\ ]_i$.

To illustrate our results in the previous sections, we fix a modal language $\mathcal{L}_n(\Phi)$ (whose formulae are) interpreted in an arbitrary Kripke scs $\mathbf{C} = \mathbf{K}(\mathcal{S}_n(\Phi))$. Suppose we wish to extend it with modalities $\Box_i^{-1}$, called reverse modalities also interpreted over the same set of KS's $\mathcal{S}_n(\Phi)$ and satisfying some minimal requirement. The language is given by the following grammar.

▶ **Definition 19** (Modal Language with Reverse Modalities). Let Φ be a set of primitive propositions. The modal language $\mathcal{L}_n^{+r}(\Phi)$ is given by the following grammar: $\phi, \psi, \ldots := p \mid \phi \wedge \psi \mid \neg\phi \mid \Box_i\phi \mid \Box_i^{-1}\phi$ where $p \in \Phi$ and $i \in \{1, \ldots, n\}$.

The minimal semantic requirement for each $\Box_i^{-1}$ is that:

$$\Box_i\Box_i^{-1}\phi \ \Leftrightarrow \ \phi \quad \text{valid in } \mathcal{S}_n(\Phi). \tag{2}$$

We then say that $\Box_i^{-1}$ is a *right-inverse* modality for $\Box_i$.

Since $\mathbf{C}[\![\Box_i\phi]\!] = [\ \mathbf{C}[\![\phi]\!]\ ]_i$, we can derive semantic interpretations for $\Box_i^{-1}\phi$ by using a right inverse $\uparrow_i$ for $[\cdot]_i$ in Definition 18. Assuming that such a right inverse exists, we can interpret the reverse modality in $\mathbf{C}$ as

$$\mathbf{C}[\![\Box_i^{-1}\phi]\!] = \uparrow_i(\ \mathbf{C}[\![\phi]\!]\ ). \tag{3}$$

We can choose $\uparrow_i$ in Equation (3) from the set $\{\uparrow_i^{\mathtt{N}}, \uparrow_i^{\mathtt{MN}}, \uparrow_i^{\mathtt{M}}\}$ of right-inverses given in Section 3.5.

3.7.1 Temporal Operators

We conclude this section with a brief discussion on some right-inverse linear-time modalities. Let us suppose that $n = 2$ in our modal language $\mathcal{L}_n(\Phi)$ under consideration (thus interpreted in Kripke scs $\mathbf{C} = \mathbf{K}(\mathcal{S}_2(\Phi))$. Assume further that the intended meaning of the two modalities $\Box_1$ and $\Box_2$ are the *next* operator ($\bigcirc$) and the *henceforth/always* operator ($\Box$), respectively, in a *linear-time* temporal logic. To obtain the intended meaning we take $\mathcal{S}_2(\Phi)$ to be the largest set such that: If $M \in \mathcal{S}_2(\Phi)$, M is a 2-agent KS where $\xrightarrow{1}_M$ is isomorphic to the successor relation on the natural numbers and $\xrightarrow{2}_M$ is the reflexive and transitive closure of $\xrightarrow{1}_M$. The relation $\xrightarrow{1}_M$ is intended to capture the linear flow of time. Intuitively, $s \xrightarrow{1}_M t$ means t is the only next state for s. Similarly, $s \xrightarrow{2}_M t$ for $s \neq t$ is intended to capture the fact that t is one of the infinitely many future states for s.

Let us first consider the next operator $\Box_1 = \bigcirc$. Notice that $\xrightarrow{1}_M$ is determinant-complete. If we apply Equation (3) with $\uparrow_1 = \uparrow_1^{\mathtt{M}}$, we obtain $\Box_1^{-1} = \ominus$, a past modality known in the literature as *strong* previous operator [13]. If we take $\uparrow_i$ to be the normal right inverse $\uparrow_i^{\mathtt{N}}$, we obtain $\Box_1^{-1} = \tilde{\ominus}$, the past modality known as *weak* previous operator [13]. Notice that the only difference between the two operators is that, if s is an indetermined/initial state wrt $\xrightarrow{1}_M$ then $(M, s) \not\models \ominus\, \phi$ and $(M, s) \models \tilde{\ominus}\, \phi$ for any ϕ. It follows that $\ominus$ is not a normal operator, since $\ominus \top$ is not valid in $\mathcal{S}_2(\Phi)$ but $\top$ is.

Let us now consider the always operator $\Box_2 = \Box$. Notice that $\xrightarrow{2}_M$ is not determinant-complete: Take any increasing chain $s_0 \xrightarrow{1}_M s_1 \xrightarrow{1}_M \ldots$ The state s_1 is not determinant because for every s_j such that $s_1 \xrightarrow{2}_M s_j$ we also have $s_0 \xrightarrow{2}_M s_j$. Theorem 13 tells us that there is no right-inverse $\uparrow_2$ of $[\cdot]_i$ that can give us an operator $\Box_2^{-1}$ satisfying Equation (2).

4 Ongoing Work

4.1 Knowledge in Terms of Space

In this section we show our current work on using spatial constraint systems to express the epistemic concept of knowledge by using the following notion of global information:

▶ **Definition 20** (Global Information). Let $\mathcal{C}$ be an n-scs with space functions $[\cdot]_1, \ldots, [\cdot]_n$ and G be a non-empty subset of $\{1, \ldots, n\}$. *Group-spaces* $[\cdot]_G$ and *global* information $[\![\cdot]\!]_G$ of G in $\mathcal{C}$ are defined as:

$$[c]_G \stackrel{\mathtt{def}}{=} \bigsqcup_{i \in G} [c]_i \quad \text{and} \quad [\![c]\!]_G \stackrel{\mathtt{def}}{=} \bigsqcup_{j=0}^{\infty} [c]_G^j \tag{4}$$

where $[c]_G^0 \stackrel{\mathtt{def}}{=} c$ and $[c]_G^{k+1} \stackrel{\mathtt{def}}{=} [[c]_G^k]_G$.

The constraint $[c]_G$ means that c holds in the spaces of agents in G. The constraint $[\![c]\!]_G$ entails $[[\ldots[c]_{i_m}\ldots]_{i_2}]_{i_1}$ for any $i_1, i_2, \ldots, i_m \in G$. Thus, it realizes the intuition that c holds *globally* wrt G: c holds in each nested space involving only the agents in G. In particular, if G is the set of all agents, $[\![c]\!]_G$ means that c holds *everywhere*. From the epistemic point of view $[\![c]\!]_G$ is related to the notion of common-knowledge of c [5].

4.2 Knowledge Constraint System

In [9] the authors extended the notion of spatial constraint system to account for *knowledge*. In this summary we shall refer to the extended notion in [9] as *S4 constraint systems* since it is meant to capture the epistemic logic for knowledge S4. Roughly speaking, one may wish to use $[c]_i$ to represent not only some information c that agent i has but rather a *fact* that he knows. The domain theoretical nature of constraint systems allows for a rather simple and elegant characterization of knowledge by requiring space functions to be *Kuratowski closure operators* [10]: i.e., monotone, extensive and idempotent maps preserving bottom and lubs.

▶ **Definition 21** (Knowledge Constraint System [9]). An n-agent *S4 constraint system (n-s4cs)* $\mathbf{C}$ is an n-scs whose space functions $[\cdot]_1, \ldots, [\cdot]_n$ are also *closure operators.* Thus, in addition to S.1 and S.2 in Definition 3, each $[\cdot]_i$ also satisfies:
(EP.1) $[c]_i \sqsupseteq c$ and
(EP.2) $[[c]_i]_i = [c]_i$.

Intuitively, in an n-s4cs, $[c]_i$ states that the agent i has knowledge of c in its store $[\cdot]_i$. The axiom EP.1 says that if agent i knows c then c must hold, hence $[c]_i$ has at least as much information as c. The epistemic principle that an agent i is aware of its own knowledge (*the agent knows what he knows*) is realized by EP.2. Also, the epistemic assumption that agents are *idealized reasoners* follows from the monotonicity of space functions, i.e., for a consequence c of d ($d \sqsupseteq c$), then if d is known to agent i, so is c, $[d]_i \sqsupseteq [c]_i$.

In [9] the authors use the notion of *Kuratowski closure operators* $[c]_i$ to capture knowledge. In what follows we show an alternative interpretation of knowledge as the global construct $[\![c]\!]_G$ in Definition 20.

4.3 Knowledge as Global Information

Let $\mathcal{C} = (Con, \sqsubseteq, [\cdot]_1, \ldots, [\cdot]_n)$ be a *spatial* constraint system. From Definition 20 we obtain the following equation:

$$[\![c]\!]_{\{i\}} = c \sqcup [c]_i \sqcup [c]_i^2 \sqcup [c]_i^3 \sqcup \ldots = \bigsqcup_{j=0}^{\infty} [c]_i^j \qquad (5)$$

For simplicity, we shall use $[\![\cdot]\!]_i$ as an abbreviation of $[\![\cdot]\!]_{\{i\}}$. We shall demonstrate that $[\![c]\!]_i$ can also be used to represent the knowledge of c by agent i.

We will show that the global function $[\![c]\!]_i$ is in fact a Kuratowski closure operator and thus satisfies the epistemic axioms EP.1 and EP.2 above: It is easy to see that $[\![c]\!]_i$ satisfies $[\![c]\!]_i \sqsupseteq c$ (EP.1). Under certain natural assumptions we shall see that it also satisfies $[\![[\![c]\!]_i]\!]_i = [\![c]\!]_i$ (EP.2). Furthermore, we can combine knowledge with our belief interpretation of space functions: clearly, $[\![c]\!]_i \sqsupseteq [c]_i$ holds for any c. This reflects the epistemic principle that *whatever is known is also believed* [8].

We now show that any *spatial constraint system* with continuous space functions (i.e. functions preserving lubs of any directed set) $[\cdot]_1, \ldots, [\cdot]_n$ induces an s4cs with space functions $[\![\cdot]\!]_1, \cdots [\![\cdot]\!]_n$.

▶ **Definition 22.** Given an scs $\mathcal{C} = (Con, \sqsubseteq, [\cdot]_1, \ldots, [\cdot]_n)$, we use $\mathcal{C}^*$ to denote the tuple $(Con, \sqsubseteq, [\![\cdot]\!]_1, \ldots, [\![\cdot]\!]_n)$.

One can show that $\mathcal{C}^*$ is also a spatial constraint system. Besides, it is an s4cs as stated next.

▶ **Theorem 23.** *Let $\mathcal{C} = (Con, \sqsubseteq, [\cdot]_1, \ldots, [\cdot]_n)$ be a spatial constraint system. If $[\cdot]_1, \ldots, [\cdot]_n$ are continuous functions, then $\mathcal{C}^*$ is an n-agent s4cs.*

We shall now prove that S4 can also be captured using the global interpretation of space.

From now on $\mathbf{C}$ denotes the Kripke constraint system $\mathbf{K}(\mathcal{M})$ (Definition 9), where $\mathcal{M}$ represent a set of non-empty set of n-agent Kripke strutures. Notice that constraints in $\mathbf{C}$, and consequently also in $\mathbf{C}^*$, are sets of *unrestricted* (pointed) Kripke structures. Although $\mathbf{C}$ is not an S4cs, from the above theorem, its induced scs $\mathbf{C}^*$ is. Also, we can give in $\mathbf{C}^*$ a sound and complete compositional interpretation of S4 formulae.

The compositional interpretation of modal formulae in our constraint system $\mathbf{C}^*$ is similar to the one introduced in 18 except for the interpretation of the $\Box_i\phi$ modality.

Notice that $\Box_i\phi$ is interpreted in terms of the global operation. Since $\mathbf{C}^*$ is a power-set ordered by reversed inclusion, the lub is given by set intersection. Thus, from Equation (5)

$$\mathbf{C}^*[\![\Box_i\phi]\!] = [\![\ \mathbf{C}^*[\![\phi]\!]\ ]\!]_i = \bigsqcup_{j=0}^{\omega}[\mathbf{C}^*[\![\phi]\!]]_i^j = \bigcap_{j=0}^{\omega}[\mathbf{C}^*[\![\phi]\!]]_i^j \tag{6}$$

In particular, from Theorem 23 and Axiom EP.2, $\mathbf{C}^*[\![\Box_i\phi]\!] = \mathbf{C}^*[\![\Box_i(\Box_i\phi)]\!]$ follows as an S4-knowledge modality; i.e., if agent i knows ϕ he knows that he knows it.

We conclude this section with the following theorem stating the correctness wrt validity of the interpretation of knowledge as as global operator.

▶ **Theorem 24.** $\mathbf{C}^*[\![\phi]\!] = true$ *if and only if* ϕ *is S4-valid.*

5 Future Work

As future work we are planning to specify the epistemic notion of *Distributed Knowledge* (DK) [5] as well as a computational notion of *process* in our algebraic structures.

5.1 Distributed Knowledge in Terms of Space

Informally, DK says that, if a given agent i has $c \to d$ in his space and an agent j has c in her space, then if we were to communicate with each other we could have d in their space though individually neither i nor j has d. This could be an important concept for distributed systems, e.g. to predict unwanted behavior in a system upon potential communication among agents.

Using [5] and our notion of Heyting implication in Definition 5 we could extend scs with DK as follows.

▶ **Definition 25.** Let $\mathcal{C} = (Con, \sqsubseteq, [\cdot]_1, \ldots, [\cdot]_n)$. Let $G \subseteq \{1, 2, \ldots, n\}$ be a non-empty subset of agents. Distributed knowledge of G is a self-map $\mathcal{D}_G : Con \to Con$ satisfying the next axioms:

1. $\mathcal{D}_G(true) = true$
2. $\mathcal{D}_G(c \sqcup d) = \mathcal{D}_G(c) \sqcup \mathcal{D}_G(d)$
3. $\mathcal{D}_G(c) = [c]_i$ *if* $G = \{i\}$
4. $\mathcal{D}_{G'}(c) \sqsupseteq \mathcal{D}_G(d)$ *if* $G' \subseteq G$

Intuitively $\mathcal{D}_G(c)$ means that G has DK of c. The first condition says that any G has DK of *true*. The second condition says that, if G has DK of two pieces of information c and d, then G has DK of their join. The third condition tells us that an agent has DK of what he knows. Finally, the fourth condition says that the larger the subgroup, the greater its DK.

In previous paragraphs we argued that if agent i has $c \to d$ and an agent j has c then they would have DK of d ($\mathcal{D}_{\{i,j\}}(d)$). Indeed, from the above axioms and the properties of space, one can prove that $[c \to d]_i \sqcup [c]_j \sqsupseteq \mathcal{D}_{\{i,j\}}(d)$.

As future work we would like to give an explicit spatial construction that characterizes $\mathcal{D}_G(c)$.

5.2 Processes as Constraint Systems

Concurrent constraint programming (ccp) calculi are a well-known family of process algebras from concurrency theory [15, 12, 4, 11]. Computational processes from ccp can be seen as closure operators over an underlying constraint system $\mathcal{C} = (Con, \sqsubseteq)$. A closure operator f over $\mathcal{C} = (Con, \sqsubseteq)$ is a monotonic self map on Con such that $f(c) \sqsupseteq c$ and $f(f(c)) = f(c)$.

It is well known that closure operators form themselves a complete lattice. Thus, ccp processes can be interpreted as elements of the cs $\mathcal{C}^+ = (Con^+, \sqsubseteq)$ where Con^+ is the set of closure operators over Con ordered wrt $\sqsubseteq$ (recall that $f \sqsubseteq g$ iff $f(c) \sqsubseteq g(c)$ for every $c \in Con$.)

We plan to use the space and extrusion functions from spatial constraint systems to give a declarative semantics to the corresponding spatial, time and extrusion constructs in ccp-based process algebras. More importantly, we plan to use the notion of distributed knowledge to derive a corresponding notion in ccp-process algebras. To our knowledge this will be the first time that distributed knowledge is used in the context of process calculi.

References

1 Samson Abramsky and Achim Jung. Domain theory. *Handbook of logic in computer science*, pages 1–77, 1994.

2 Patrick Blackburn, Maarten De Rijke, and Yde Venema. *Modal Logic*. Cambridge University Press, 1st edition, 2002.

3 Frank S. Boer, Alessandra Di Pierro, and Catuscia Palamidessi. Nondeterminism and infinite computations in constraint programming. *Theoretical Computer Science*, pages 37–78, 1995.

4 Alessandra Di Pierro, Catuscia Palamidessi, and Frank S. Boer. An algebraic perspective of constraint logic programming. *Journal of Logic and Computation*, pages 1–38, 1997.

5 Ronald Fagin, Joseph Y Halpern, Yoram Moses, and Moshe Y Vardi. *Reasoning about knowledge*. MIT press Cambridge, 4th edition, 1995.

6 M. Guzman, S. Haar, S. Perchy, C. Rueda, and F. Valencia. Belief, knowledge, lies and other utterances in an algebra for space and extrusion. *Journal of Logical and Algebraic Methods in Programming*, 2016.

7 M. Guzman, S. Perchy, C. Rueda, and F. Valencia. Deriving extrusion on constraint systems from concurrent constraint programming process calculi. In *ICTAC 2016*, 2016.

8 Jaakko Hintikka. *Knowledge and belief*. Cornell Univeristy Press, 1962.

9 Sophia Knight, Catuscia Palamidessi, Prakash Panangaden, and Frank D Valencia. Spatial and epistemic modalities in constraint-based process calculi. In *CONCUR 2012*, pages 317–332. Springer, 2012.

10 John Charles Chenoweth McKinsey and Alfred Tarski. The algebra of topology. *Annals of mathematics*, pages 141–191, 1944.

11 Nax P Mendler, Prakash Panangaden, Philip J Scott, and RAG Seely. A logical view of concurrent constraint programming. *Nordic Journal of Computing*, pages 181–220, 1995.

12 Prakash Panangaden, Vijay Saraswat, Philip J Scott, and RAG Seely. A hyperdoctrinal view of concurrent constraint programming. In *Workshop of Semantics: Foundations and Applications, REX*, pages 457–476. Springer, 1993.

13 Amir Pnueli and Zohar Manna. *The temporal logic of reactive and concurrent systems*. Springer, 1992.

14 Sally Popkorn. *First steps in modal logic.* Cambridge University Press, 1st edition, 1994.

15 Vijay A Saraswat, Martin Rinard, and Prakash Panangaden. Semantic foundations of concurrent constraint programming. In *POPL'91*, pages 333–352, 1991.

16 Steven Vickers. *Topology via logic.* Cambridge University Press, 1st edition, 1996.

Tabled CLP for Reasoning Over Stream Data*

Joaquín Arias

IMDEA Software Institute, Spain
`joaquin.arias@imdea.org`; and
Technical University of Madrid, Madrid, Spain
`joaquin.arias.herrero@alumnos.upm.es`

Abstract

The interest in reasoning over stream data is growing as quickly as the amount of data generated. Our intention is to change the way stream data is analyzed. This is an important problem because we constantly have new sensors collecting information, new events from electronic devices and/or from customers and we want to reason about this information. For example, information about traffic jams and costumer order could be used to define a deliverer route. When there is a new order or a new traffic jam, we usually restart from scratch in order to recompute the route. However, if we have several deliveries and we analyze the information from thousands of sensors, we would like to reduce the computation requirements, e.g. reusing results from the previous computation. Nowadays, most of the applications that analyze stream data are specialized for specific problems (using complex algorithms and heuristics) and combine a computation language with a query language. As a result, when the problems become more complex (in e.g. reasoning requirements), in order to modify the application complex and error prone coding is required.

We propose a framework based on a high-level language rooted in logic and constraints that will be able to provide customized services to different problems. The framework will discard wrong solutions in early stages and will reuse previous results that are still consistent with the current data set. The use of a constraint logic programming language will make it easier to translate the problem requirements into the code and will minimize the amount of re-engineering needed to comply with the requirements when they change.

1998 ACM Subject Classification D.3.2 Constraint and logic languages, I.2.8 Graph and tree search strategies, H.2.8 Data mining

Keywords and phrases logic, languages, tabling, constraints, graph, analysis, reasoning

Digital Object Identifier 10.4230/OASIcs.ICLP.2016.17

1 Introduction and Problem Description

In recent years, wired and wireless sensors, social media and the Internet of Things generate data (stream data) which is expanding in three fronts: velocity (speed of data generation), variety (types of data) and volume (amount of data). As a result the demand for analysis and reasoning over stream data (stream data mining) has exploded [17].

The main property of stream data is that the sets of data change due to insertion, modification and/or deletion of data. In most cases, the subset of changed data is substantially smaller than the complete amount of data which is analyzed. The objective of stream data mining is to find relations and associations between the values of categorical variables in big sets of data (millions of items or more), which are dynamically updated.

* Work partially funded by Comunidad de Madrid project S2013/ICE-2731 *N-Greens Software* and MINECO Projects TIN2012-39391-C04-03 *StrongSoft* and TIN2015-67522-C3-1-R *TRACES*.

Datalog, a high level language based on logic, has demonstrated its efficiency in stream reasoning (systems like Deductive Applications Language System (DeALS) [9] developed in UCLA, StreamLog [33] and Yedalog [6] by Google, are based on Datalog), and in machine learning where the queries are executed in parallel over big databases distributed in different clusters.

Our system is based on Prolog because it is more expressive (Datalog is semantically a subset of Prolog) and its native search strategy is top-down instead of bottom-up. This means that the search is guided by the query reducing the search tree. Our system also provides two extensions: constraints logic programming, which discards search options in early stages reducing the search tree, and tabling, an execution strategy which avoids entering loops in some cases and reuses previous results. As a result these extensions not only increase the performance of Prolog but also its expressiveness as we show in Sec. 4.

When data changes, most current approaches have to recompute the analysis over the complete data set. With our system, the combination of tabling and constraints will minimize or override the recomputation overhead by computing only the subset of data affected by the modified data, because:

- The tabling engine will invalidate results that are inconsistent with the current data set in order to reuse previous results in such a way that we can ensure they are correct.
- New constraints solvers will make it possible to define restriction to prune the search tree during the data analysis. A pruned search tree reduces the number of accesses to databases and/or tables.

2 Background and Overview of the existing literature

In this section we will describe the framework (TCLP) which make it possible to integrate constraints solvers in the tabling engine; the data model that will be used to represent data; the constraints needed to deal with the dynamic nature of the data; and a brief state of the art.

2.1 TCLP: Tabling + Constraints

Constraint Logic Programming (CLP) [12] extends Logic Programming (LP) with variables which can belong to arbitrary constraint domains and the ability to incrementally solve the equations involving these variables. CLP brings additional expressive power to LP, since constraints can very concisely capture complex relationships between variables. Also, shifting from "generate-and-test" to "constrain-and-generate" patterns reduces the search tree and therefore improves performance, even if constraint solving is in general more expensive than unification.

Tabling [26, 30] is an execution strategy for logic programs which suspends repeated calls which would cause infinite loops. Answers from other, non-looping branches, are used to resume suspended calls which can in turn generate more answers. Only new answers are saved, and evaluation finishes when no new answers can be generated. Tabled evaluation always terminates for calls / programs with the bounded term depth property and can improve efficiency for programs which repeat computations, as it automatically implements a variant of dynamic programming. Tabling has been successfully applied in a variety of contexts, including deductive databases, program analysis, semantic Web reasoning, and model checking [31, 8, 34, 20].

The combination of CLP and tabling [28, 23, 7, 4], called TCLP, brings several advantages. It improves termination properties and increases speed in a range of programs. It has been

```
person {                              triple(s01, type,  person)
  name: "John Doe"                    triple(s01, name,  "John Doe")
  email: "jdoe@gmail.com"             triple(s01, email, "jdoe@gmail.com")
}
```

Figure 1 A person model in Protocol Buffers (left) and Prolog syntax (right).

applied in several areas, including constraint databases [13, 28], verification of timed automata and infinite systems [3], and abstract interpretation [27].

2.2 Graph Databases

Graph Databases are increasingly used to store data and most of the current data, such as linked data on the Web and social network data, are graph-structured [32]. A graph database is essentially a collection of nodes and edges. There are different graph data models but we will limit our research to the directed labeled graphs where the edges are directed and identified with a label.

The Resource Description Framework (RDF) [21] is a standard model for data interchange on the Web. RDF referees an edge as a "triple" <subject> <predicate> <object> and allows structured and semi-structured data to be mixed, exposed, and shared across different applications. As a result, it facilitates the integration of data from different sources. The RDF model theory also formalizes the notion of inference in RDF and provides a basis for computing deductive closure of RDF graphs.

The OWL Web Ontology Language [16], based on the RDF framework, was designed to represent rich and complex knowledge about things, groups of things, and relations between things. OWL documents, known as ontologies, define concept type hierarchies in such a way that a property defined for a more general concept is also defined for the concept subsumed by the more general concept. It is also possible to define various hierarchical relations.

OWL is a computational logic-based language that can be exploited to verify the consistency of the database knowledge or to make explicit an implicit knowledge. As a result, since Prolog is also a logic-based language, there are several RDF-APIs in Prolog which provide an interface to RDF databases and engine interfaces based on Prolog like F-OWL [34] to reasoning over OWL ontologies. The RDF triples can be easily translated into Prolog i.e. using facts of the form triple(Subject, Predicate, Object).

Other languages, like Protocol Buffers [19] based on name-value pairs, which Google uses as a common representation of data, can also be modelled as a directed labeled graph. Fig 1 shows the model of a person with a name and an email in protocol buffer test format (left) and in Prolog syntax (right).

The data model based on directed labeled graphs combined with the unification of Prolog, makes it easy to read, write, match and transform the data. Additionally, Sec. 4 shows that TCLP will increase the performance and termination properties of Prolog in most of the reasoning problems over graph databases because they can be solved in terms of reachability, connectivity, and distance in graphs.

2.3 Stream Time Constraints

The analysis of stream data has to deal with the unbounded nature of the data. First, it is not possible to store all the generated data, therefore several techniques have been developed to process the data and to store only the relevant information. Second, the queries have to

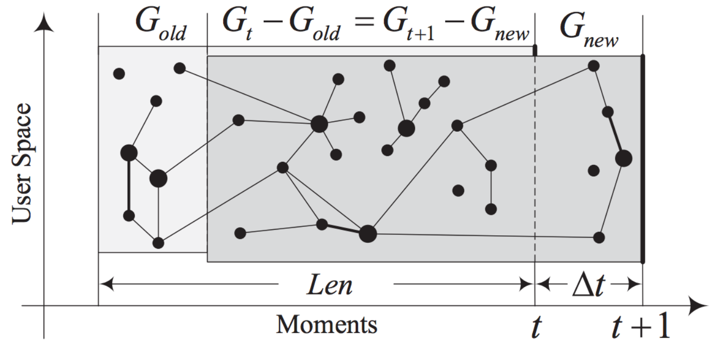


Figure 2 Sliding time window from time t to t+1. G_t will be updated by deleting subgraph G_{old} and adding subgraph G_{new}. Example from [15].

be re-evaluated periodically and in one pass because the source data is not stored for further evaluation.

Usually the reasoning is performed over a snapshot of a finite amount of data (a window). A window is defined by its size (a fixed time interval or a number of data items) and, since the queries are repeated in the time, it is also defined by a slide distance (the time between two consecutive queries). In many applications, the time interval size of the window is larger than the slide distance, so the set of data that is modified (due to addition or deletion) is smaller than the set of data contained in the window. Our intention is to design a system that updates the results recomputing the part of the modified data instead of recomputing the query over the complete data set. Similar work presented in [15] applied an incremental tracking framework (see Fig 2) to the event evolution tracking task in social streams, showing much better efficiency than other approaches.

Constraint logic programming provides arithmetical constraint solvers that can deal with the window definition in a natural manner (i.e. interval constraint solvers), and the operations required to deal with temporal reasoning [1] can be evaluated by the constraint solver.

2.4 State of the Art

In recent years several new logic languages, most of which are based on Datalog, have been developed to reason over stream data. Two of them are: Yedalog [6] developed by Google, an extension of Datalog that seamlessly mixes data-parallel pipelines and computation in a single language, and adds features for working with data structured as nested records; and LogiQL [11] developed by LogicBlox, a unified and declarative language based on Datalog with advanced incremental maintenance (changes are computed in an incremental fashion) and live programming facilities (changes to application code are quickly compiled and "hot-swapped" into the running program). There is more research done in this direction and some of its results are described in the surveys [17, 32].

3 Goal of the Research

Our goal is to extend the functionality of Prolog (logic programming language) to provide a full high level programming language which can be used to reason over stream data, reusing previous results instead of recomputing them from scratch when new data arrives.

We intend to make the stream analysis a native capability of our system by using the

```
dist(X, Y, D) :-
    dist(X, Z, D1),
    edge(Z, Y, D2),
    D is D1 + D2.
dist(X, Y, D) :-
    edge(X, Y, D).
```

```
dist(X, Y, D) :-
    D1 #> 0, D2 #> 0,
    D  #= D1 + D2,
    dist(X, Z, D1),
    edge(Z, Y, D2).
dist(X, Y, D) :-
    edge(X, Y, D).
```

Figure 3 Versions of distance in a graph: Prolog / tabling (left) and CLP / TCLP (right). The symbols #> and #= are (in)equalities in CLP.

monotonicity of logic programming and by introducing the revision of previous inferences when facts are removed, which is a form of non-monotonicity.

We envision advantages in several fronts: complex queries and non-trivial reasoning will be easier to express thanks to the higher-level of logic programming and constraints; fewer computations will be necessary thanks to the automatic reuse of previous inferences brought by tabling (which in a certain sense performs dynamic programming in an automatic way); queries and associated actions (if any) can be programmed using the same syntax.

4 Current Status of the Research and Results Accomplished

During my first year of PhD I have been designing and implementing the TCLP framework which eases the integration of additional constraint solvers in an existing tabling module in Ciao Prolog[1].

The main goal of the TCLP framework is to make the addition of constraint solvers easier. In order to achieve this goal, we determined the services that a constraint solver should provide to the tabling engine. The constraint solver can freely implement them and has been designed to cover many different implementations.

To validate our design we have interfaced: one solver for difference constraints, previously written in C, existing classical solvers (CLP(Q/R)), and a new solver for constraints over finite lattices. We have found the integration to be easy – certainly easier than with other designs, given the capabilities that our system provides. We evaluate the performance of our framework in several benchmarks using the aforementioned constraint solvers. All the development work and evaluation was done in Ciao Prolog and is described in [2].

In order to highlight some of the advantages of TCLP versus Prolog, CLP and tabling with respect to declarativeness and logical reading, we compare the behavior of these paradigms and strategies using different versions of a program to compute distances between nodes in a graph. Each version is adapted to a different paradigm, but trying to stay as close as possible to the original code, so that the additional expressiveness can ultimately be attributed to the semantics of the programming language and not to differences in the code itself.

The code in Fig. 3, left, is the Prolog / tabling version of the program dist/3 to find nodes in a graph within a distance K from each other. Fig. 3, right, is the CLP / TCLP version of the same code. In order to find the nodes X and Y within a maximum distance K from each other we use the queries ?- dist(X,Y,D), D < K. and ?- D #< K, dist(X,Y,D). in

[1] A robust, mature, next-generation Prolog system. Stable versions of Ciao Prolog are available at `http://www.ciao-lang.org`.

Table 1 Run time (ms) for dist/3. A '–' means no termination.

	Prolog	CLP	Tabling	TCLP	
Left recursion	–	–	144	45	Without cycles
Right recursion	1917	200	291	184	
Left recursion	–	–	–	420	With cycles
Right recursion	–	4261	–	1027	

Prolog / tabling and CLP / TCLP, respectively. To evaluate the performance, we use a graph of 25 nodes without cycles (with 584 edges) or with cycles (with 785 edges).

Table 1 shows the termination properties and speed of dist/3 in the four paradigms. It highlights that TCLP terminates in all the cases and it is also the fastest one. Additionally, it shows, in line with the experience on tabling, that left-recursive implementations are usually faster and preferable.

These results are relevant because most of the reasoning problems over graph databases are solved in terms of reachability, connectivity and distances in graphs. In fact, this example is a typical query for the analysis of social networks [25].

5 Open Issues and Expected Achievements

Constraint solver over ontologies. The idea of answer subsumption (which only stores an answer if it is more general than the previous answers according to a defined partial order) was presented in [25]. The paper also analyzes its application in social network analysis. From our point of view, the TCLP framework will increase this performance because it can be used not only to check answer subsumption, but also to avoid the execution of queries where the concepts are more particular (they are entailed in terms of the ontology hierarchy) than the concepts of a previous query. Moreover, the constraint solver can be used to state the relationships defined in the ontology as constraint before the analysis starts. These relations can propagate and prune the search space reducing the computation and eventually avoiding accesses to databases.

Temporal constraint solver. The analysis should be done over a finite window of time, therefore a constraint solver is needed to deal with the operation required by the temporal reasoning tasks [1]. Moreover, the integration of the solver with the TCLP framework will increase its benefits because some of its operations will explode the stored results stored.

Stream-TCLP. In order to apply our framework to stream data, the answers must be returned as soon as they are available. Instead of the local scheduling which tries to find all the answers before returning them, the tabling engine should use an incremental answering strategy similar to batch scheduling [10], JET mechanism [22] or swapping evaluation [5].

Dynamic tabling. A more complex technique - similar to incremental tabling [24] - has to be defined in order to: invalidate knowledge inferred by data which is updated / removed; update the knowledge when the temporal window slides; and remove previous tabled results to make place for more recent results.

Stream recursive aggregates. Some research has been done in the field of aggregates (see [14, 18, 29]) regarding the Prolog program semantic in tabled execution and with recursive queries. And since most of the queries are defined in terms of aggregates as min, sum or count, it is relevant to take into consideration this research problem which is unclear and related with non-monotonic properties.

References

1 James F Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.

2 J. Arias and M. Carro. Description and Evaluation of a Generic Design to Integrate CLP and Tabled Execution. In *18th Int'l. ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'16)*. ACM Press, September 2016.

3 Witold Charatonik, Supratik Mukhopadhyay, and Andreas Podelski. Constraint-based infinite model checking and tabulation for stratified clp. In Peter J. Stuckey, editor, *ICLP*, volume 2401 of *Lecture Notes in Computer Science*, pages 115–129. Springer, 2002.

4 P. Chico de Guzmán, M. Carro, M. Hermenegildo, and P. Stuckey. A General Implementation Framework for Tabled CLP. In Tom Schrijvers and Peter Thiemann, editors, *FLOPS'12*, number 7294 in LNCS, pages 104–119. Springer Verlag, May 2012.

5 P. Chico de Guzmán, M. Carro, and David S. Warren. Swapping Evaluation: A Memory-Scalable Solution for Answer-On-Demand Tabling. *Theory and Practice of Logic Programming, 26th Int'l. Conference on Logic Programming (ICLP'10) Special Issue*, 10 (4–6):401–416, July 2010.

6 Brian Chin, Daniel von Dincklage, Vuk Ercegovac, Peter Hawkins, Mark S Miller, Franz Och, Christopher Olston, and Fernando Pereira. Yedalog: Exploring knowledge at scale. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 32. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

7 Baoqiu Cui and David Scott Warren. A System for Tabled Constraint Logic Programming. In *Computational Logic*, pages 478–492, 2000.

8 S. Dawson, C. R. Ramakrishnan, and D. S. Warren. Practical Program Analysis Using General Purpose Logic Programming Systems – A Case Study. In *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation*, pages 117–126, New York, USA, 1996. ACM Press.

9 Deductive Application Language System. http://wis.cs.ucla.edu/deals/.

10 Juliana Freire, Terrance Swift, and David Scott Warren. Beyond Depth-First Strategies: Improving Tabled Logic Programs through Alternative Scheduling. *Journal of Functional and Logic Programming*, 1998(3), 1998.

11 Todd J Green, Dan Olteanu, and Geoffrey Washburn. Live programming in the LogicBlox system: a MetaLogiQL approach. *Proceedings of the VLDB Endowment*, 8(12):1782–1791, 2015.

12 J. Jaffar and M.J. Maher. Constraint LP: A Survey. *JLP*, 19/20:503–581, 1994.

13 Paris C. Kanellakis, Gabriel M. Kuper, and Peter Z. Revesz. Constraint Query Languages. *J. Comput. Syst. Sci.*, 51(1):26–52, 1995.

14 David B Kemp and Peter J Stuckey. Semantics of logic programs with aggregates. In *ISLP*, volume 91, pages 387–401. Citeseer, 1991.

15 Pei Lee, Laks VS Lakshmanan, and Evangelos E Milios. Incremental cluster evolution tracking from highly dynamic network data. In *2014 IEEE 30th International Conference on Data Engineering*, pages 3–14. IEEE, 2014.

16 OWL Web Ontology Language Guide. http://www.w3.org/TR/owl-guide/.

17 Emanuele Panigati, Fabio A Schreiber, and Carlo Zaniolo. Data streams and data stream management systems and languages. In *Data Management in Pervasive Systems*, pages 93–111. Springer International Publishing, 2015.

18 Nikolay Pelov, Marc Denecker, and Maurice Bruynooghe. Well-Founded and Stable Semantics of Logic Programs with Aggregates. *TPLP*, 7(3):301–353, 2007. `doi:10.1017/S1471068406002973`.

19 Protocol Buffers. `https://developers.google.com/protocol-buffers/`.

20 Y.S. Ramakrishna, C.R. Ramakrishnan, I.V. Ramakrishnan, S.A. Smolka, T. Swift, and D.S. Warren. Efficient Model Checking Using Tabled Resolution. In *CAV*, volume 1254 of *LNCS*, pages 143–154. Springer Verlag, 1997.

21 Resource Description Framework (RDF). `https://www.w3.org/RDF/`.

22 Konstantinos F. Sagonas and Peter J. Stuckey. Just Enough Tabling. In *Principles and Practice of Declarative Programming*, pages 78–89. ACM, August 2004.

23 Tom Schrijvers, Bart Demoen, and David Scott Warren. TCHR: a Framework for Tabled CLP. *TPLP*, 8(4):491–526, 2008.

24 Terrance Swift. Incremental tabling in support of knowledge representation and reasoning. *Theory and Practice of Logic Programming*, 14(4-5):553–567, 2014.

25 Terrance Swift and David Scott Warren. Tabling with answer subsumption: Implementation, applications and performance. In Tomi Janhunen and Ilkka Niemelä, editors, *JELIA*, volume 6341 of *Lecture Notes in Computer Science*, pages 300–312. Springer, 2010. `doi:10.1007/978-3-642-15675-5`.

26 H. Tamaki and M. Sato. OLD Resol. with Tabulation. In *ICLP*, pages 84–98. LNCS, 1986.

27 David Toman. Constraint Databases and Program Analysis Using Abstract Interpretation. In *CDTA*, volume 1191 of *LNCS*, pages 246–262, 1997.

28 David Toman. Memoing Evaluation for Constraint Extensions of Datalog. *Constraints*, 2(3/4):337–359, 1997. `doi:10.1023/A:1009799613661`.

29 Alexander Vandenbroucke, Maciej Pirog, Benoit Desouter, and Tom Schrijvers. Tabling with Sound Answer Subsumption. *Theory and Practice of Logic Programming, 32th Int'l. Conference on Logic Programming (ICLP'16)*, 16, October 2016.

30 D. S. Warren. Memoing for Logic Programs. *CACM*, 35(3):93–111, 1992.

31 R. Warren, M. Hermenegildo, and S. K. Debray. On the Practicality of Global Flow Analysis of Logic Programs. In *JICSLP*, pages 684–699. MIT Press, August 1988.

32 Peter T Wood. Query languages for graph databases. *ACM SIGMOD Record*, 41(1):50–60, 2012.

33 Carlo Zaniolo. A logic-based language for data streams. In *SEBD*, pages 59–66, 2012.

34 Youyong Zou, Tim Finin, and Harry Chen. F-OWL: An Inference Engine for Semantic Web. In *Formal Approaches to Agent-Based Systems*, volume 3228 of *Lecture Notes in Computer Science*, pages 238–248. Springer Verlag, January 2005.

Testing of Concurrent Programs

Miguel Isabel[*]

Complutense University of Madrid, Madrid, Spain
`miguelis@ucm.es`

Abstract

Testing concurrent systems requires exploring all possible non-deterministic interleavings that the concurrent execution may have, as any of the interleavings may reveal erroneous behaviour. This introduces a new problem: the well-known state space problem, which is often computationally intractable. In the present thesis, this issue will be addressed through: (1) the development of new *Partial-Order Reduction Techniques* and (2) the combination of static analysis and testing (*property-based testing*) in order to reduce the combinatorial explosion. As a preliminary result, we have performed an experimental evaluation on the SYCO tool, a CLP-based testing framework for actor-based concurrency, where these techniques have been implemented. Finally, our experiments prove the effectiveness and applicability of the proposed techniques.

1998 ACM Subject Classification F.1.1 Models of Computation

Keywords and phrases Property-based Testing, Partial Order Reduction, Deadlock-Guided Testing, Deadlock Detection, Systematic Testing

Digital Object Identifier 10.4230/OASIcs.ICLP.2016.18

1 Introduction

Due to increasing performance demands, application complexity and multi-core parallelism, concurrency is omnipresent in today's software applications. It is widely recognized that concurrent programs are difficult to develop, debug, test and analyze. This is even more so in the context of concurrent *imperative* languages that use a global memory (so called heap) to which the different tasks can have access. These accesses introduce additional hazards not present in sequential programs such as race conditions, data races, deadlocks, and livelocks. Therefore, software validation techniques urge especially in the context of concurrent programming.

Testing is the most widely-used methodology for software validation. However, due to the non-deterministic interleaving of tasks, traditional testing for concurrent programs is not as effective as for sequential programs. In order to ensure that all behaviors of the program are tested, the testing process, in principle, must systematically explore all possible ways in which the tasks can interleave. This is known as *systematic testing* [1] in the context of concurrent programs. Such full systematic exploration of all task interleavings produces the well known state explosion problem and is often computationally intractable (see, e.g., [2] and its references).

We consider actor systems [3], a model of concurrent programming that has been regaining popularity lately and that is being used in many systems (such as Go, ActorFoundry, Asynchronous Agents, Charm++, E, ABS, Erlang, and Scala). The Actor Model is having extensive influence on commercial practice. For example, Twitter has used actors for scalability, also, Microsoft has used the actor model in the development of its asynchronous

[*] Supervised by Elvira Albert & Miguel Gómez-Zamalloa.

agents library. Actor programs consist of computing entities called actors, each with its own local state and thread of control, that communicate by exchanging messages asynchronously.

An actor configuration consists of the local state of the actors and a set of pending tasks. In response to receiving a message, an actor can update its local state, send messages, or create new actors. In the computation of an actor system, there are two non-deterministic choices: first which actor is selected, and then which task of its pending tasks is scheduled. The actor model is characterized by inherent concurrency of computation within and among actors, dynamic creation of actors, and interaction only through direct asynchronous message passing with no restriction on message arrival order. The interaction using non-preemptive asynchronous communication (i.e., the execution of a task cannot be interrupted by another one), together with the fact that there is no shared memory among different actors, facilitates in general the application of formal methods.

In particular, for the sake of systematic testing, one can assume [1] that the evaluation of all statements of a task takes place serially (without interleaving with any other task) until it releases the processor (gets to a return instruction).

Compared to multi-threaded systems, this reduces a lot the state explosion problem. However, a naive exploration of the search space to reach all possible system configurations still does not scale. The challenge of systematic testing of concurrent programs in general is to avoid as much as possible the exploration of redundant paths which lead to the same configuration and paths which are not leading to the satisfaction of some property.

2 Goals of the Research

The focus of this thesis project is the development and application of new techniques for actor-based systems testing, which allow to carry out the validation process efficiently (reducing the combinatorial explosion) and, therefore, applicable to large systems; and the adaptation of these techniques to other concurrency models and widely-used languages. In order to reduce the state space explored, we are going to address the problem from different angles:

- guiding the execution towards paths satisfying some property, and pruning the uninteresting ones (*property-based testing*),
- avoiding the exploration of redundant paths which lead to the same configuration (*Partial-Order Reduction techniques* [4]),
- applying these techniques in the context of symbolic execution, and
- developing a CLP-based framework incorporating all these new techniques.

3 State of the Art

The main goal of testing is bug detection. There are different kinds of bugs that one can aim at catching. In concurrent programs, *deadlocks* are one of the most common programming errors and, thus, a main goal of verification and testing tools is, respectively, proving deadlock freedom and *deadlock detection*. Therefore, one of the properties we could be interested in is deadlock detection, guiding the execution only towards those paths that might lead to deadlock, and prune those that we know certainly cannot lead to deadlock.

Static analysis and testing are two different ways of detecting deadlocks that often complement each other and thus it seems quite natural to combine them. Static analysis evaluates an application by examining its code but without executing it. In contrast, testing consists of executing the application for concrete input values. Since a deadlock can manifest only on specific sequences of task interleavings, in order to apply testing for deadlock detection, the testing process must systematically explore all task interleavings.

The primary advantage of *systematic testing* [1, 5] for deadlock detection is that it can provide the detailed deadlock trace with all information that the user needs in order to fix the problem. However, there is an important shortcoming, as we said before, although recent research tries to avoid redundant exploration as much as possible [5, 6, 7, 8], the search space of systematic testing (even without redundancies) can be huge. This is a threat to the application of testing in concurrent programming.

Partial-order reduction (POR) [9] is a general theory that helps mitigate this combinatorial explosion by formally identifying equivalence classes of redundant explorations. Early POR algorithms were based on different static analyses to detect and avoid exploring redundant derivations. The state-of-the-art POR algorithm [10], called DPOR (Dynamic POR), improves over those approaches by dynamically detecting and avoiding the exploration of redundant derivations on-the-fly. Since the invention of DPOR, there have been several works [1] proposing improvements, variants and extensions in different contexts to the original DPOR algorithm.

The most notable one is [2] which proposes an improved DPOR algorithm which further reduces redundant computations ensuring that only one derivation per equivalence class is generated. Some of these works [1] have addressed the application of POR to the context of actor systems from different perspectives. The most recent one [2] presents the TransDPOR algorithm, which extends DPOR to take advantage of a specific property in the dependency relations in pure actor systems, namely transitivity, to explore fewer configurations than DPOR.

4 Current Status of the Research

The first way of reducing the combinatorial explosion that we have explored is by means of *property-based testing* and, in particular, guiding the testing process towards those paths leading to deadlock. Static analysis [15, 16, 17, 18] and testing [20, 21, 22, 23] are two different ways of detecting deadlocks that often complement each other, and, thus it seems quite natural to combine them.

In Integrated Formal Methods 2016, we have presented a seamless combination of static analysis and testing for effective deadlock detection (*deadlock-guided testing*) [11] that works as follows: an existing static deadlock analysis [12] is first used to obtain *abstract* descriptions of potential deadlock cycles. Now, given an abstract deadlock cycle, we guide the systematic execution towards paths that might contain a representative of that abstract deadlock cycle, by discarding paths that are guaranteed not to contain such a representative. The main idea is as follows: (1) From the abstract deadlock cycle, we generate *deadlock-cycle constraints*, which must hold in all states of derivations leading to the given deadlock cycle. (2) We extend the execution semantics to support deadlock-cycle constraints, with the aim of stopping derivations as soon as cycle-constraints are not satisfied, which do not lead to a deadlock of the given cycle. So those executions are stopped as soon as they are guaranteed not to lead to a state satisfying the deadlock-state constraints.

5 Experiments & Preliminary Results

We have implemented the SYCO tool [13], a testing tool for *concurrent objects* which is available at http://costa.ls.fi.upm.es/syco. The whole testing framework for this actor-based language has been implemented by means of CLP. It consists of two basic parts: first, the imperative program is compiled into an equivalent CLP program and, second, systematic testing is performed on the CLP program by relying only on CLP's evaluation mechanisms.

In our approach, the whole testing process is formulated using CLP only, and without the need of defining specific operators to handle the different features. This, on the one hand, has the advantage of providing a clean and uniform formalization. And, more importantly, since systematic testing is performed on an equivalent CLP program, we can often obtain the desired degree of coverage by using existing evaluation strategies on the CLP side. This gives us flexibility and parametricity w.r.t. the adequacy criteria. SYCO is based on aPET [14], a Partial-Evaluation based TCG tool by symbolic execution.

The experiments have been performed using as benchmarks: (1) classical concurrency patterns containing deadlocks and (2) deadlock free versions of them, for which deadlock analyzers give false positives. We have compared the results obtained using a systematic testing setting and a deadlock-guided testing setting. Regarding the first set of benchmarks, we achieve significant gains w.r.t systematic testing and, thus, this proves the applicability, effectiveness and impact of *deadlock-guided testing*. Finally, for the examples that are deadlock free, we are also able to prove deadlock freedom for most cases where static analysis reports false positives.

6 Open Issues & Future Work

The techniques developed so far address dynamic testing, but our approaches would be applicable also in static testing, where the execution is performed on constraint variables rather than on concrete values. These possible extensions will require the use of termination criteria which provide the desired degree of coverage. Our CLP-based framework will facilitate the application of these extensions.

The development of improvements in precision of *Partial Order Reduction* techniques and the study of their applicability to concurrent languages containing blocking synchronization instructions also remains as future work in the thesis project.

Up to now, we have only studied the combination of deadlock analysis and testing in order to reduce the combinatorial explosion. However, other types of analysis could be used in this approach, for instance: resource analysis, if we want to guide the exploration towards those paths which are consuming above a threshold; termination analysis, towards paths that do terminate but the analysis is not able to prove it; and starving analysis, which could guide the testing process towards paths leading to a starving situation.

References

1 K. Sen and G. Agha. Automated Systematic Testing of Open Distributed Programs. In *Proc. FASE'06*, Lecture Notes in Computer Science 3922, pages 339–356. Springer, 2006.

2 S. Tasharofi, R. K. Karmani, S. Lauterburg, A. Legay, D. Marinov, and G. Agha. Trans-DPOR: A Novel Dynamic Partial-Order Reduction Technique for Testing Actor Programs. In FMOODS/FORTE, volume 7273 of Lecture Notes in Computer Science, pages 219–234. Springer, 2012.

3 G.A. Agha. Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press, Cambridge, MA, 1986.

4 Patrice Godefroid. Partial-Order Methods for the Verification of Concurrent Systems. An Approach to the State-Explosion Problem, volume 1032 of Lecture Notes in Computer Science. Springer, 1996.

5 M. Christakis, A. Gotovos, and K. F. Sagonas. Systematic Testing for Detecting Concurrency Errors in Erlang Programs. In ICST'13, pages 154–163. IEEE, 2013.

6 P. Abdulla, S. Aronis, B. Jonsson, and K. F. Sagonas. Optimal Dynamic Partial Order Reduction. In *Proc. of POPL'14*, pages 373–384. ACM, 2014.

7 E. Albert, P. Arenas and M. Gómez-Zamalloa. Actor- and Task-Selection Strategies for Pruning Redundant State-Exploration in Testing. In *FORTE'14*, Pages 49–65, Springer.

8 C. Flanagan and P. Godefroid. Dynamic Partial-Order Reduction for Model Checking Software. In Proceedings of POPL'05, pages 110–121. ACM, 2005.

9 Patrice Godefroid. Using Partial Orders to Improve Automatic Verification Methods. In Proceedings of CAV, volume 531 of LCNS, pages 176-185. Springer, 1991.

10 Cormac Flanagan and Patrice Godefroid. Dynamic Partial-Order Reduction for Model Checking Software. In Proceedings of POPL'05, pages 110–121. ACM, 2005.

11 E. Albert, M. Gómez-Zamalloa, M. Isabel. Combining Static Analysis and Testing for Deadlock Detection. In Proceedings of iFM'16, pages 409–424.

12 A. Flores-Montoya, E. Albert, and S. Genaim. May-Happen-in-Parallel based Deadlock Analysis for Concurrent Objects. In *FORTE'13*, Lecture Notes in Computer Science 7892. 2013.

13 E. Albert, M. Gómez-Zamalloa. M. Isabel. SYCO: a Systematic Testing Tool for Concurrent Objects. In Proceedings of CC'16, pages 269–270.

14 E. Albert, P. Arenas, M.Gómez-Zamalloa, P. Y. H. Wong: aPET: a test case generation tool for concurrent objects. In Proceedings of ESEC/SIGSOFT FSE'13, pages 595–598.

15 E. Giachino, C.A. Grazia, C. Laneve, M. Lienhardt, and P. Wong. Deadlock Analysis of Concurrent Objects – Theory and Practice, 2013.

16 S. P. Masticola and B. G. Ryder. A Model of Ada Programs for Static Deadlock Detection in Polynomial Time. In *Parallel and Distributed Debugging*. ACM, 1991.

17 M. Naik, C. Park, K. Sen, and D. Gay. Effective Static Deadlock Detection. In Proceedings of ICSE, pages 386-396. IEEE, 2009.

18 R. Agarwal, L. Wang and S. D. Stoller. Detecting Potential Deadlocks with Static Analysis and Run-Time Monitoring. In *HVC*, Lecture Notes in Computer Science 3875. Springer, 2006.

19 P. Joshi, M. Naik, K. Sen, and Gay D. An Effective Dynamic Analysis for Detecting Generalized Deadlocks. In Proceedings of FSE'10, pages 327–336. ACM, 2010.

20 P. Joshi, C. Park, K. Sen, and M. Naik. A Randomized Dynamic Program Analysis Technique for Detecting Real Deadlocks. In Proceedings of PLDI'09. ACM, 2009.

21 A. Kheradmand, B. Kasikci, and G. Candea. Lockout: Efficient Testing for Deadlock Bugs. Technical report, 2013.

22 S. Savage, M. Burrows, G. Nelson, P. Sobalvarro and T. E. Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM TCS*, 1997.

23 K. Havelund, Using Runtime Analysis to Guide Model Checking of Java Programs, In Proceedings of the 7th International SPIN Workshop, Springer-Verlag, 2000.

24 Javier Esparza. Model Checking Using Net Unfoldings. Sci. Comput. Program., pages 151–195, 1994.

Controlled Natural Languages for Knowledge Representation and Reasoning[*]

Tiantian Gao

Dept. of Computer Science, Stony Brook University, Stony Brook, NY, USA
tiagao@cs.stonybrook.edu

Abstract

Controlled natural languages (CNLs) are effective languages for knowledge representation and reasoning. They are designed based on certain natural languages with restricted lexicon and grammar. CNLs are unambiguous and simple as opposed to their base languages. They preserve the expressiveness and coherence of natural languages. In this paper, it mainly focuses on a class of CNLs, called machine-oriented CNLs, which have well-defined semantics that can be deterministically translated into formal languages to do logical reasoning. Although a number of machine-oriented CNLs emerged and have been used in many application domains for problem solving and question answering, there are still many limitations: First, CNLs cannot handle inconsistencies in the knowledge base. Second, CNLs are not powerful enough to identify different variations of a sentence and therefore might not return the expected inference results. Third, CNLs do not have a good mechanism for defeasible reasoning. This paper addresses these three problems and proposes a research plan for solving these problems. It also shows the current state of research: a paraconsistent logical framework from which six principles that guide the user to encode CNL sentences were created. Experiment results show this paraconsistent logical framework and these six principles can consistently and effectively solve word puzzles with injections of inconsistencies.

1998 ACM Subject Classification I.2.1 Applications and Expert Systems

Keywords and phrases Controlled Natural Languages, Paraconsistent Logics, Defeasible Reasoning

Digital Object Identifier 10.4230/OASIcs.ICLP.2016.19

1 Introduction

Controlled natural languages (CNLs) are effective languages for knowledge representation and reasoning. According to [27], "A controlled natural language is a constructed language that is based on a certain natural language, being more restrictive concerning lexicon, syntax, and/or semantics while preserving most of its natural properties". Unlike the languages that develop naturally, constructed languages are the languages whose lexicon and syntax are designed with intent. A CNL is constructed on the basis of an existing natural language, such as English, French, or German. Words in the lexicon of a CNL mainly come from its base language. They may or may not be used in the same manner as in the base language. Some words are used with fewer senses or reserved as key-words for specific purposes. CNLs have a well-defined syntax to form phrases, sentences and texts. The syntax of a CNL is generally simpler than that of the source language. Sentences are interpreted in a deterministic way. CNLs are more accurate than natural languages, because the language is more restrictive,

[*] This work was partially supported by Michael Kifer and Paul Fodor.

but not all CNLs have formal semantics. Those that have formal semantics can be processed by computers for knowledge representation, machine translation, and logical reasoning. Although a CNL may deviate from its base language in the lexicon, syntax, and/or semantics, it still preserves most of the natural properties of the base language, so the reader would correctly comprehend the CNL with little effort.

CNLs generally fall into two categories: human-oriented CNLs and machine-oriented CNLs. Human-oriented CNLs are designed to make the texts easier for readers to understand [40]. Examples include Basic English [36], Special English [35], and Simplified Technical English [23]. Machine-oriented CNLs, as opposed to human-oriented ones, have formal semantics which can be understood and processed by computers for the purpose of knowledge representation and logical reasoning. Examples include Attempto Controlled English (ACE) [17], Processable English (PENG) [39], and Computer-processable Language (CPL) [14].

Typically, there is a big learning curve for domain experts in the fields such as law, business and medical to represent the domain-specific knowledge in computer languages. CNLs are superior to other ways of knowledge representation in that they require little knowledge for users to understand the syntax and semantics of the underlying knowledge representation framework. Users can encode the knowledge base in English by following a restricted grammar and then make inferences. For example, in the ERGO project (Effective Representation of Guidelines with Ontologies)[1], ACE was used to author pediatric guideline recommendations. As a result, the clinical practice guidelines can be automatically translated into rules which can be incorporated into decision support systems to facilitate clinicians. In [41], PENG was used to solve word puzzles. The original Jobs Puzzle was rewritten in CNL sentences with the addition of some implicit background knowledge. These CNL sentences are then transformed into a program in Answer Set Programming (ASP) [22] paradigm to compute the answer. In [13], CPL was used to encode the AP (advanced high-school) level examination questions. The CNLs questions are machine-understandable such that they can be processed by its inference system for question-answering.

There are limitations to the aforementioned CNLs. First, they cannot conduct reasoning in the presence of inconsistencies. In practical cases, it is very likely that the knowledge base is constructed from different sources, thus the occurrences of inconsistencies are quite likely. However, current reasoning systems for the aforementioned CNLs do not accept inconsistencies. Because of this, occurrences of inconsistencies in one source will break the whole system and inhibit reasoning. But, in many cases, inconsistencies in one source may not affect the others. Thus, it is necessary to know which piece of information is inconsistent. Besides, it is also desirable to derive things from the information which is consistent.

Second, current CNLs have limited power to identify variations of a sentence. Although CNLs have restricted grammar and pre-defined interpretation rules, users still have multiple choices to express a sentence. Consider the case of question-answering, users may also compose questions in different words as opposed the ones used in the knowledge base or have different ways of writing the same sentence in CNL. In the aspect of knowledge representation, it is desirable to map the variations of a sentence to the same logical form. Otherwise, users might not be able to get the inference results as expected. For instance, in ACE, phrases like “Mary’s father” and “the father of Mary” are represented as the same form. However, given the sentences “Mary’s gender is female. If Mary is a female then Mary is a doctor”, ACE will not derive the conclusion that “Mary is a doctor”.

Third, it is common that CNL sentences are not created equally. Different sentences imply different degrees of priorities. Consider the sentences “Every bird flies. Penguins do

[1] `http://gem.med.yale.edu/ergo/`

not fly". The first sentence states the default case: a bird flies. The second sentence indicates a higher priority than the first one – in that if something is a penguin, it will refute the conclusion drawn from the first one. This type of reasoning is called defeasible reasoning [34]. Defeasible statements are common in texts. Although reference [42] provides a mechanism to denote defaults and exceptions in CNL, it is still very limited when handling priorities in complex cases. Details will be discussed in the next section.

In the following: Section 2 gives an overview of existing CNLs. Section 3 presents the goal of the research. Section 4 shows the current state of research, to be specific, a powerful paraconsistent logical framework and six principles derived the logic for encoding CNL sentences. Section 5 gives a brief summary of the experiment results show the aforementioned logical framework and principles can consistently and effectively solve word puzzles. Section 6 discusses the open issues and the expected achievements in the future. Section 7 concludes the paper.

2 Background

ACE is the first CNL that can be translated to first-order logic. ACE is a subset of English defined by a restricted grammar along with interpretation rules that control the semantic analysis of grammatically correct ACE sentences. ACE uses discourse resolution structure (DRS) [25] as the logical structure to represent the semantics of a set of ACE sentences. ACE is supported by a language processor, Attempto Parsing Engine (APE), and a reasoner, RACE [19]. APE is an online language processor that allows users to compose ACE sentences as input and generates their semantics in DRS and first-order logic clauses as output. RACE is a CNL reasoner that supports theorem proving, consistency checking, and question answering. RACE is implemented in Prolog. It is an extension of Satchmo [30], which is a theorem prover based on the model generation paradigm. Satchmo executes the clauses by forward reasoning and generates a minimal finite model of clauses. RACE extends Satchmo by giving a justification for every proof, finding all minimal unsatisfiable subsets of clauses if the axioms are not consistent,

PENG was developed by Rolf Schwitter at Macquarie University. It was partly inspired by ACE. PENG is a subset of English with restricted grammar and use DRS as semantic representation. Unlike ACE, PENG does not require users to learn the grammar of the language. Instead, it designs a predictive editor that informs users of the look-ahead information that guides users to proceed based on the structure of the current sentence. The original implementation of PENG's reasoner is based on a theorem-prover Otter [32] and a model builder MACE [33]. The reasoner supports consistency checking, informativity checking, and question answering. In later extensions, PENG translates CNL sentences into ASP programs and embeds Clingo [21] as its underlying reasoner for question answering. Besides, it extends the grammar to support defeasible reasoning [42] by introducing defaults and exceptions. A default statement is identified by the keyword *normally*. There are two types exceptions: strong exception and weak exception, where strong exceptions, identified by the word "not", can refute the default conclusion and weak exceptions, identified by the keyword "abnormally" make the default conclusion inapplicable without refuting it. There are a few limitations to this approach. First, the way it represents weak exceptions is more close to the English translation of the intended ASP rule. It is very hard for users to correctly represent weak exceptions in CNL without knowing the underlying ASP rules. Second, the design of defaults and exceptions only generates two levels of priorities where exceptions have higher priorities than defaults and therefore refuting the defaults. However, in real cases, it is very common, especially in the fields of law and financial regulations, that there are more than two levels of priorities among sentences. A sentence can refute some sentences while the sentence itself can be refuted by others as well.

CPL was developed by Peter Clark at University of Texas. The vocabulary of CPL is based on a pre-defined Component Library (CLib) ontology [8]. CPL accepts three types of sentences: ground facts, rules, and questions. The semantics of CPL are represented by KM (Knowledge Machine) [15] sentences. KM is a powerful frame-based knowledge representation language. It represents first-order logic clauses in LISP-like syntax. The CPL interpreter translates a CPL sentence into KM sentences in three steps. First, the interpreter uses a bottom-up, broad coverage chart parser, called SAPIR [24], to parse a CPL sentence and then generates a logical form (LF). Second, an initial logic generator is used to transform the LF into ground logical assertions (KM sentences) by applying a set of simple, syntactic rewrite rules. Third, subsequent post-processing is performed based on the logical assertions generated in Step 2, including word sense disambiguation, semantic role labelling, and structural re-organization.

BioQuery-CNL [16] is a CNL designed for representing biomedical documents. The expressive power of BioQuery-CNL is superior to existing semantic web query languages, like SPARQL. For instance, users can write simple English phrases such as “gene-gene relation chain” to indicate transitive closures. Both the biomedical knowledge base and user queries are encoded by ASP programs, which can be fed into ASP solvers for making inferences. Between the querying interface and the underlying knowledge base, there is an intermediate layer, the rule layer, which stores definitions of auxiliary concepts derived from the knowledge base. These auxiliary definitions help connect ASP queries to the underlying knowledge base.

NL2KR [43] is a platform that can translate natural languages into knowledge representation formalisms. It consists of two sub-parts: NL2KR-L and NL2KR-T, where NL2KR-L is the training phase of the system and NL2KR-T is the translation system. Both parts embed a Combinatory Categorial Grammar (CCG) [28] parser, where each word is associated with a syntactic category and semantic representation in the form of λ-expressions. The purpose of NL2KR-L is to learn the semantic meaning of each word in the lexicon based on a training set. Given the sentences and their semantic representations, Inverse-λ [6] is used to extract the semantic meaning of a word within the given context. When Inverse-λ is not enough to extract the meaning of the words, Generalization [7] is used to guess the meaning of the words. Ambiguity is solved by a Parameter Learning module which learns the weights of all possible meanings to a word and chooses the most probable one. After the training phase, words in the lexicon are augmented with new meanings extracted from the training set. In the translation phase, sentences are translated by a CCG parser. Same as NL2KR-L, Generalization is used to determine the meaning of unknown words. Experimental results show that NL2KR achieves high accuracy when applied to GeoQuery and Jobs datasets for question-answering.

In addition to CNL systems, current advances in ASP provide ways to solve more complicated knowledge representation problems in CNL. For instance, CR-Prolog [5] extends ASP with consistency-restoring rules (cr-rules), which can be used to specify exceptions. Once inconsistencies arise in the knowledge base, cr-rules are used to override the conclusions derived from default statements. This logical framework captures the characteristics of defeasible reasoning in natural languages. Another extension of ASP is EZCSP [2], which is designed to encode numerical information and reason about it efficiently. This feature can be applied to represent numerical information in natural languages and achieve high performance in reasoning. In addition, there is ASP{f} [3, 4], which augments EZCSP and can handle defaults and exceptions in ASP as well.

3 Goal of the Research

The first goal is to develop a paraconsistent logic that handles inconsistencies in CNL reasoning. Although there is a list of paraconsistent logics, e.g., [37], [10], and [9], they deal with inconsistencies from the philosophical or mathematical point of view. Other paraconsistent logics, such as [11] and [26], were developed for definite logic programs and cannot be easily applied to solving more complex CNL reasoning problems.

There is a list of desired properties the intended paraconsistent logical framework is supposed to have: First, the logic intends to identify the most likely cause of inconsistencies. Consider the knowledge base consisting of the following sentences: 1) Every actor is male, 2) Mary is a female, and 3) Mary is an actor. Apparently, the knowledge base is not consistent since Mary is a female but she is also an actor. There could be two explanations: one where Mary is not an actor and the other where Mary is not a female. Given that Mary is a female name, the former explanation is more reasonable than the latter one. To achieve this goal, it is required that the logic can select the most preferred models by taking into account some background knowledge. Second, in some cases, contrapositive inference is used in CNL reasoning but this is not always the case. Therefore, the logic intends to provide a mechanism to allow/inhibit contrapositive inference. Third, since if + *premise* + then + *conclusion* statements are used to derive new facts, it is necessary that the logical framework has a mechanism to decide whether or not to derive conclusions from inconsistent premises. Last, as closed world assumption [29] is used in databases, this is also useful for CNL reasoning. Therefore, the underlying logic should be able to ensure complete knowledge of information.

The second goal is to standardize logical representations of CNLs, such that they have more power to identify different variations of the same sentence and map them to the same logical form. As is discussed in the introduction, although simple forms of paraphrases of sentences can be identified by CNLs, they are still very limited. For instance, the sentence "Mary has a dog" and "Mary owns a dog" will be translated into two different logical forms in ACE. As a result, users may not get the expected answers when they compose question in a way that uses different terms as in the knowledge base. First, a list of standardized relations is required to be defined to achieve this goal. Second, methods should be proposed to extract the relations from CNL sentences by consulting their syntactic or semantic properties. Although there is a list of tools such as StanfordIE and Ollie for relation extractions, the number of pre-defined relations are very small. Although the structures of CNL sentences are more restricted as opposed to the ones StanfordIE [1] and Ollie [31] work on, the standardization intends to normalize all possible relations in logical representations instead of a few pre-defined relations such as *location*, *founded_by*, etc.

The third goal is to enable defeasible reasoning in CNLs. The previous section shows the limitations of [42] in handling defaults. That is, there can be only two levels of priorities among CNL sentences. To allow more than two levels of priorities for CNL sentences, to the best my knowledge, Logic Programming with Defaults and Argumentation Theories (LPDA) [44] can be considered as a good candidate framework with desirable features.

LPDA is based on the three-valued well-founded semantics [38]. It is a unifying defeasible reasoning framework that uses defaults and exceptions with prioritized rules, and argumentation theories. LPDA has two types of rules: strict and defeasible, where strict rules generate non-defeasible conclusions and defeasible rules generate defeasible conclusions that can be defeated by some exceptions. Each LPAD program is accompanied by an argumentation theory that specifies when a defeasible rule is defeated. A rule is defeated if it is refuted, rebutted, or disqualified. Generally, a rule is refuted if there is another rule that draws an incompatible conclusion with higher priority. A rule is rebutted if there is another rule that draws an incompatible conclusion and there is no way to resolve the conflict based on the relative priorities. A rule is disqualified if it is cancelled, self-defeated, etc. Based on LPDA, defeasible statements in CNL can be encoded by defeasible rules and their priorities can be specified in argumentation theories.

Another challenge is identification of the priorities among CNL sentences. This can be done either explicitly by user specifications or implicitly detected by some background knowledge or natural language understanding methodologies.

4 Current State of Research

Reference [20] shows the current state of research. A new kind of paraconsistent logic was developed to deal with inconsistencies in word puzzles, more generally, for translating CNL sentences into logic. The logical framework is based on the well-known type of paraconsistent logics, Annotated Predicate Calculus (APC) [26], but has a new kind of non-monotonic semantics, called *consistency preferred stable models*. The language is a logic programming subset of APC, denoted as APC_{LP}. APC_{LP} can be isomorphically embedded in ASP extended with a model preference framework, such as the Clingo [21] with its Asprin extension [12]. It was proved in [20] that this embedding is one-to-one and preserves the semantics.

Along with the logical framework, six principles were proposed to guide users to encode CNL sentences in APC_{LP}. Each of the principles will be briefly described in the following: Principle 1 guides users to encode an if + *premise* + then + *conclusion* sentence that can perform contrapositive inference. Principle 2 describes the way to encode if + *premise* + then + *conclusion* sentences such that it can allow/inhibit derivations of conclusions from inconsistent premises. Principle 3 addresses the encoding of polar facts in CNL. For instance, a person must be either a male or a female, but not both or unknown. When inconsistency is possible, this principle ensures this requirement. Besides, if one of them is inconsistent then the other is too. Inconsistent information is not created equal, as people have different degrees of confidence in different pieces of information based on common sense knowledge. For example, there is more confidence in that someone whom people barely know is a person compared to the information about this person's marital situation (e.g., whether a husband exists). Principle 4 allows users to specify the degrees of confidence. As a result, when there are multiple explanations for the cause of inconsistencies, Principle 4 will select the most reasonable one by consulting the degrees specified. Principle 5 behaves like the closed world assumption. It guides users to encode CNL sentences ensure complete knowledge of information. Principle 6 captures the cardinality constraints in the presence of inconsistencies in CNL sentences, e.g., a person holds exactly one job.

5 Preliminary results

The paraconsistent logical framework and the proposed principle mentioned in the previous section have been applied to solve word puzzles, such as Jobs Puzzle [45] and Zebra Puzzle[2] with inconsistencies. Experiment results show that in the cases where there is no inconsistency, APC_{LP} can correctly compute the answer. In the cases of inconsistencies, APC_{LP} can find

[2] `https://en.wikipedia.org/wiki/Zebra_Puzzle`

the most likely cause of inconsistencies within the puzzle and give reasonable inference results. More detailed information can be found in [20].

6 Open Issues and Expected Achievements

The first issue is that current CNLs have limited power to recognize variations of a sentence and therefore might not always map sentences that express the same meaning to the same logical form. As the next step, it is intended to extend ACE to overcome this issue. ACE parser translates CNL sentences into DRS with pre-defined predicates [18] to represent the semantics of a sentence. This form of representation is simple and well-structured. It is intended to do post-processing based on the semantic representation in order to extract semantic relations standardized in ontologies, such as DBpedia[3] and Wikidata[4]. The second issue is to perform defeasible reasoning in CNLs. In order to detect the refutation relations between two sentences, it is expected to do the following: First, extend ACE to incorporate some background knowledge for primitive detection of sentence priorities. Second, design a user interface that allows users to make corrections. In addition, it is expected to extend current DRS representation to accommodate defeasible information, which will be eventually translated to an LPDA program for defeasible reasoning.

7 Conclusion

In this paper, it first gives an overview of the development of CNLs and discusses the limitations of current CNLs in the aspect of knowledge representation and reasoning. Then, it gives an outline of the research plan for solving these problems. This includes designing a paraconsistent logical framework for knowledge representation, empowering current CNLs to recognize variations of a sentence and perform defeasible reasoning. Next, it shows the current state of research – a powerful paraconsistent logical framework along with six principles derived from that for encoding CNL sentences. In addition, it shows the application of the current work to solving word puzzles with inconsistencies. Finally, it addresses some open issues and presents the plans for future achievements.

References

1 Gabor Angeli, Melvin Jose Johnson Premkumar, and Christopher D. Manning. Leveraging linguistic structure for open domain information extraction. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing, ACL 2015, July 26-31, 2015, Beijing, China, Volume 1: Long Papers*, pages 344–354. Association for Computational Linguistics, The Association for Computer Linguistics, 2015.

2 Marcello Balduccini. Representing constraint satisfaction problems in answer set programming. In *ICLP09 Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP09)(Jul 2009)*, 2009.

3 Marcello Balduccini. A “conservative” approach to extending answer set programming with non-herbrand functions. In *Correct Reasoning*, pages 24–39. Springer, 2012.

[3] `http://wiki.dbpedia.org/`
[4] `https://www.wikidata.org/wiki/Wikidata:Main_Page`

4 Marcello Balduccini. Asp with non-herbrand partial functions: A language and system for practical use. *Theory and Practice of Logic Programming*, 13(4-5):547–561, 2013.

5 Marcello Balduccini and Michael Gelfond. Logic programs with consistency-restoring rules. In *International Symposium on Logical Formalization of Commonsense Reasoning, AAAI 2003 Spring Symposium Series*, volume 102, 2003.

6 Chitta Baral, Juraj Dzifcak, Marcos Alvarez Gonzalez, and Aaron Gottesman. Typed answer set programming lambda calculus theories and correctness of inverse lambda algorithms with respect to them. *TPLP*, 12(4-5):775–791, 2012.

7 Chitta Baral, Juraj Dzifcak, Marcos Alvarez Gonzalez, and Jiayu Zhou. Using inverse lambda and generalization to translate english to formal languages. *CoRR*, abs/1108.3843, 2011.

8 Ken Barker, Bruce W. Porter, and Peter Clark. A library of generic concepts for composing knowledge bases. In *Proceedings of the First International Conference on Knowledge Capture (K-CAP 2001), October 21-23, 2001, Victoria, BC, Canada*, pages 14–21. ACM, 2001.

9 Nuel D Belnap Jr. A useful four-valued logic. In *Modern uses of multiple-valued logic*, pages 5–37. Springer, 1977.

10 Jean-Yves Béziau, Walter Alexandre Carnielli, and Dov M Gabbay. *Handbook of paraconsistency.* College Publications, 2007.

11 Howard A Blair and VS Subrahmanian. Paraconsistent logic programming. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 340–360. Springer, 1987.

12 Gerhard Brewka, James P. Delgrande, Javier Romero, and Torsten Schaub. asprin: Customizing answer set preferences without a headache. In Blai Bonet and Sven Koenig, editors, *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA.*, pages 1467–1474. AAAI Press, 2015.

13 Peter Clark, Shaw Yi Chaw, Ken Barker, Vinay K. Chaudhri, Philip Harrison, James Fan, Bonnie E. John, Bruce W. Porter, Aaron Spaulding, John A. Thompson, and Peter Z. Yeh. Capturing and answering questions posed to a knowledge-based system. In Derek H. Sleeman and Ken Barker, editors, *Proceedings of the 4th International Conference on Knowledge Capture (K-CAP 2007), October 28-31, 2007, Whistler, BC, Canada*, pages 63–70. ACM, 2007.

14 Peter Clark, Philip Harrison, Thomas Jenkins, John A. Thompson, and Richard H. Wojcik. Acquiring and using world knowledge using a restricted subset of english. In Ingrid Russell and Zdravko Markov, editors, *Proceedings of the Eighteenth International Florida Artificial Intelligence Research Society Conference, Clearwater Beach, Florida, USA*, pages 506–511. AAAI Press, 2005.

15 Peter Clark, Bruce Porter, and Boeing Phantom Works. Km?the knowledge machine 2.0: Users manual. *Department of Computer Science, University of Texas at Austin*, 2:5, 2004.

16 Esra Erdem, Halit Erdogan, and Umut Öztok. BIOQUERY-ASP: querying biomedical ontologies using answer set programming. In Stefano Bragaglia, Carlos Viegas Damásio, Marco Montali, Alun D. Preece, Charles J. Petrie, Mark Proctor, and Umberto Straccia, editors, *Proceedings of the 5th International RuleML2011@BRF Challenge, co-located with the 5th International Rule Symposium, Fort Lauderdale, Florida, USA, November 3-5, 2011*, volume 799 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2011.

17 Norbert E. Fuchs, Kaarel Kaljurand, and Tobias Kuhn. Attempto controlled english for knowledge representation. In Cristina Baroglio, Piero A. Bonatti, Jan Maluszynski, Massimo Marchiori, Axel Polleres, and Sebastian Schaffert, editors, *Reasoning Web, 4th International Summer School 2008, Venice, Italy, September 7-11, 2008, Tutorial Lectures*, volume 5224 of *Lecture Notes in Computer Science*, pages 104–124. Springer, 2008.

18 Norbert E. Fuchs, Kaarel Kaljurand, and Tobias Kuhn. Discourse Representation Structures for ACE 6.6. Technical Report ifi-2010.0010, Department of Informatics, University of Zurich, Zurich, Switzerland, 2010.

19 Norbert E. Fuchs and Uta Schwertel. Reasoning in attempto controlled english. In François Bry, Nicola Henze, and Jan Maluszynski, editors, *Principles and Practice of Semantic Web Reasoning, International Workshop, PPSWR 2003, Mumbai, India, December 8, 2003, Proceedings*, volume 2901 of *Lecture Notes in Computer Science*, pages 174–188. Springer, 2003.

20 Tiantian Gao, Paul Fodor, and Michael Kifer. Paraconsistency and word puzzles. *CoRR*, abs/1608.01338, 2016.

21 Martin Gebser, Benjamin Kaufmann, Roland Kaminski, Max Ostrowski, Torsten Schaub, and Marius Thomas Schneider. Potassco: The potsdam answer set solving collection. *AI Commun.*, 24(2):107–124, 2011.

22 Michael Gelfond and Yulia Kahl. *Knowledge representation, reasoning, and the design of intelligent agents: The answer-set programming approach.* Cambridge University Press, 2014.

23 ASD Simplified Technical English Maintenance Group. *ASD-STE 100: Simplified Technical English : International Specification for the Preparation of Maintenance Documentation in a Controlled Language.* Aerospace and Defence Industries Association of Europe, 2007.

24 Philip Harrison and Michael Maxwell. A new implementation of gpsg. In *Proc. 6th Canadian Conf on AI*, pages 78–83, 1986.

25 Hans Kamp and Uwe Reyle. *From discourse to logic: Introduction to modeltheoretic semantics of natural language, formal logic and discourse representation theory*, volume 42. Springer Science & Business Media, 2013.

26 Michael Kifer and Eliezer L. Lozinskii. A logic for reasoning with inconsistency. *J. Autom. Reasoning*, 9(2):179–215, 1992.

27 Tobias Kuhn. A survey and classification of controlled natural languages. *Computational Linguistics*, 40(1):121–170, 2014.

28 Tom Kwiatkowski, Luke S. Zettlemoyer, Sharon Goldwater, and Mark Steedman. Inducing probabilistic CCG grammars from logical form with higher-order unification. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing, EMNLP 2010, 9-11 October 2010, MIT Stata Center, Massachusetts, USA, A meeting of SIGDAT, a Special Interest Group of the ACL*, pages 1223–1233. ACL, 2010.

29 Vladimir Lifschitz. Closed-world databases and circumscription. *Artif. Intell.*, 27(2):229–235, 1985.

30 Rainer Manthey and François Bry. Satchmo: a theorem prover implemented in prolog. In *International Conference on Automated Deduction*, pages 415–434. Springer, 1988.

31 Mausam, Michael Schmitz, Stephen Soderland, Robert Bart, and Oren Etzioni. Open language learning for information extraction. In Jun'ichi Tsujii, James Henderson, and Marius Pasca, editors, *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning, EMNLP-CoNLL 2012, July 12-14, 2012, Jeju Island, Korea*, pages 523–534. ACL, 2012.

32 William McCune. *Otter 3.0 reference manual and guide*, volume 9700. Argonne National Laboratory Argonne, IL, 1994.

33 William McCune. Mace4 reference manual and guide. *arXiv preprint cs/0310055*, 2003.

34 Donald Nute. Defeasible logic, handbook of logic in artificial intelligence and logic programming (vol. 3): nonmonotonic reasoning and uncertain reasoning, 1994.

35 Voice of America (Organization). *VOA Special English word book: a list of words used in Special English programs on radio, television, and the Internet.* Voice of America, 2007.

36 Charles Kay Ogden. *Basic English: A general introduction with rules and grammar*. Number 29 in Psyche miniatures., General series. K. Paul, Trench, Trubner, 1944.

37 Graham Priest, Koji Tanaka, and Zach Weber. *Paraconsistent logic*. M´unchen, 1989.

38 Teodor C. Przymusinski. Well-founded and stationary models of logic programs. *Ann. Math. Artif. Intell.*, 12(3-4):141–187, 1994.

39 Rolf Schwitter. English as a formal specification language. In *13th International Workshop on Database and Expert Systems Applications (DEXA 2002), 2-6 September 2002, Aix-en-Provence, France*, pages 228–232. IEEE Computer Society, 2002.

40 Rolf Schwitter. Controlled natural languages for knowledge representation. In Chu-Ren Huang and Dan Jurafsky, editors, *COLING 2010, 23rd International Conference on Computational Linguistics, Posters Volume, 23-27 August 2010, Beijing, China*, pages 1113–1121. Chinese Information Processing Society of China, 2010.

41 Rolf Schwitter. The jobs puzzle: Taking on the challenge via controlled natural language processing. *TPLP*, 13(4-5):487–501, 2013.

42 Rolf Schwitter. Working with defaults in a controlled natural language. In *Australasian Language Technology Association Workshop 2013*, page 106, 2013.

43 Nguyen H Vo, Arindam Mitra, and Chitta Baral. The nl2kr platform for building natural language translation systems. In *Association for Computational Linguistics (ACL)*, 2015.

44 Hui Wan, Benjamin N. Grosof, Michael Kifer, Paul Fodor, and Senlin Liang. Logic programming with defaults and argumentation theories. In Patricia M. Hill and David Scott Warren, editors, *Logic Programming, 25th International Conference, ICLP 2009, Pasadena, CA, USA, July 14-17, 2009. Proceedings*, volume 5649 of *Lecture Notes in Computer Science*, pages 432–448. Springer, 2009.

45 L. Wos. *Automated reasoning: introduction and applications*. McGraw-Hill, 1992.

The Functional Perspective on Advanced Logic Programming

Alexander Vandenbroucke

KU Leuven, Heverlee, Belgium
alexander.vandenbroucke@kuleuven.be

Abstract

The basics of logic programming, as embodied by Prolog, are generally well-known in the programming language community. However, more advanced techniques, such as tabling, answer subsumption and probabilistic logic programming fail to attract the attention of a larger audience. The cause for the community's seemingly limited interest lies with the presentation of these features: the literature frequently focuses on implementations and examples that do little to aid the understanding of non-experts in the field. The key point is that many of these advanced logic programming features can be characterised in more generally known, more accessible terms. In my research I try to reconcile these advanced concepts from logic programming (Tabling, Answer subsumption and probabilistic programming) with concepts from functional programming (effects, monads and applicative functors).

1998 ACM Subject Classification D.1.6 Logic Programming, D.3.1 Formal Definitions and Theory

Keywords and phrases Tabling, Answer Subsumption, Effect Handlers, Functional Programming, Logic Programming, Probabilistic Programming

Digital Object Identifier 10.4230/OASIcs.ICLP.2016.20

1 Introduction

Logic programming is – or has the potential to be – one of the most declarative programming paradigms. In fact, the essentials of logic programming, are generally well-known in the programming language community, and almost every computer scientist has had some exposure to Prolog.

Unfortunately, more advanced features, or more recent advances in logic programming fail to attract the attention of a larger audience beyond the logic programming community. For instance, several Prolog systems, such as XSB [19], Yap [16], B-Prolog [26] and most recently SWI-Prolog [24], support a more advanced form of resolution, SLG-resolution, also called *tabling*. Sadly, this very useful technique is completely unfamiliar to most programming language researchers that are not active in logic programming. This sometimes leads to the technique being reinvented in some very specific setting, for example for parsing.

Similarly, probabilistic logic programming extends regular logical programming to the realm of probabilistic computation, while still retaining the basic logical semantics. For example, the ProbLog system [2] is a simple syntactic extension of Prolog, where Prolog clauses can be annotated with probabilities. Such a system admits declarative specification of many probabilistic problems. ProbLog additionally supports many powerful probabilistic inference modes. However, the larger probabilistic programming community remains ignorant of these features.

The cause for this apparently limited interest from the community lies with the presentation of these features: the literature frequently focuses on implementations and examples that do little to aid the understanding of non-experts in the field. The key point is that many of these advanced logic programming features can be characterised in more generally known, more accessible terms. For example, the behaviour of logic programs is often formalised by fixed points of functions. In particular, Van Emden's concise and elegant fixed point semantics for Prolog, is a prime example of this approach.

The benefits of adopting a more general, abstract presentation are mutual and twofold:

1. By recasting (advanced features of) logic programming in a more general light, a fair comparison with similar functional systems becomes possible.
   For instance, functional logic programming systems claim to be more expressive than their logical counterparts. In a general framework, objective verification of such claims is possible, and moreover cross-pollination can proceed in a natural way.
2. The functional programming community has amassed a wealth of techniques that deal with and use non-standard control-flow. These now become readily available to the logic programmer. Recent examples of this are *delimited control* [3] and *effect handlers* [7, 15]. Here the benefits are clearly mutual: Delimited control is applied to capture tabling in a functional context. As a side-effect tabling is reduced to its essence, which in turn enables a very compact (logic programming) implementation.

Currently, my research focuses on two main areas: probabilistic (functional & logic) programming languages, and formalising tabling with answer subsumption for logic programs, which is a more advanced version of tabling.

2 Background

2.1 Probabilistic Programming Languages

Probabilities are an indispensable tool for dealing with uncertainty in real-world scenarios. They allow us to quantify missing information and thereby reason with incomplete knowledge. This key insight is the root of many advances in artificial intelligence: from machine learning and data mining, to natural language processing (NLP), information retrieval (IR) and automated reasoning. Traditionally, probabilistic models and their inference routines are tightly coupled in a single implementation, necessitating their re-implementation when the same inference technique is used for a different model. Universal probabilistic programming languages instead provide a generic platform to express probabilistic models and their inference routines. For example, consider a simple ProbLog program that models a fair coin:

```
coin(c).
0.5 :: heads(X) :- coin(X).
```

The result of the query `heads(X)` is a probability distribution which is true (with $X = c$) with probability 0.5.

Obviously, a single lingua franca for probabilistic programming enables much more efficient communication and reuse of algorithms. However, in practice the probabilistic programming landscape is highly fragmented due to the sheer number of incompatible probabilistic programming languages. Often these languages belong to completely different paradigms, from Object Oriented Programming (Microsoft's Infer.NET [11]); Logic Programming (ProbLog [2], PRISM [8]); Functional Programming (Church [5], Anglican [25]); and hybrid systems (Factorie [10], Figaro [13]).

Thus, while probabilistic programming was originally intended to unify AI-discourse on the subject, the lack of provisions for interoperability between the systems has only served to divide it further.

Clearly, what is needed is a single theory or framework that explains the relative capabilities of the different systems. When two systems are equivalent (that is, they possess the same capabilities), it should be possible to translate one system into the other and vice-versa.

Recently there has been much interest, from both functional and logical communities in using *monads* to model the semantics of probabilistic programming languages [4, 14]. Monads are a concept from category-theory, an abstract branch of mathematics. Initially, Moggi [12] proposed them as a way to structure compositional denotational semantics of programs. This compositionality has proven incredibly useful for implementing side-effects in pure functional programming languages such as Haskell [22]. Monads (and other similar category-theoretic structures) may be precisely the tool that is needed to unite the disparate branches of probabilistic programming.

2.2 Tabling with Answer Subsumption

2.2.1 Tabling

Tabling [23, 19] is a well-known and extensively studied extension of standard Prolog. The main benefit of tabling is that it brings the behaviour of many logic programs in line with their standard logical semantics. In more practical terms, it frees the Prolog programmer from worrying about more operational concerns such as clause and goal ordering. Additionally, it can dramatically speed-up the execution of a program, in exchange for higher memory consumption. Tabling has been implemented in various Prolog systems such as XSB [19], Yap [16], B-Prolog [26] and SWI-Prolog [24].

Consider the following program defining a graph containing three nodes arranged in a cycle. The edges are modelled by the `e/2`-predicate, while `p(X,Y)` holds if there is a path between `X` and `Y`.

```
:- table p/2.
e(1,2).
e(2,3).
e(3,1).

p(X,Y) :- p(X,Z),e(Z,Y).
p(X,Y) :- e(X,Y).
```

The `:-table p/2`-directive indicates that tabled resolution should be used when evaluating `p/2`. Under normal Prolog execution, the order of the clauses would cause an infinite loop, while with tabled execution the program produces all possible combinations and then *terminates*. Note that termination cannot be achieved with regular execution, even if we permute the program's clauses and bodies, since the graph contains a cycle. The technique is called tabling, because answers are stored in a data structure, called a *table* while the program is executed. In this fashion the Prolog system can keep track of the answers it has already seen.

2.2.2 Answer Subsumption and Tabling Modes

Some Prolog systems support an extension of tabling that we call *Answer subsumption*, using Swift and Warren's nomenclature [18], often implemented as a set of *tabling modes* [6]. Answer subsumption, specifies how answers should be aggregated in the table. Subsumption refers to the fact that the original answers are replaced by their aggregates, that is, they are subsumed.

Consider the following program where we use answer subsumption to compute the length of the shortest path in a graph.

```
:- table p(index,index,min).

e(1,2).
e(2,3).
e(3,1).

p(X,Y,1) :- e(X,Y).
p(X,Y,D) :- p(X,Z,D1),p(Z,Y,D2), D is D1 + D2.
```

The directive `:-table p(index,index,min)` specifies the tabling mode of each argument of the `p/3` predicate: the first two arguments serve as indexes into the table, while the final argument uses the `min` mode indicating that only the smallest answer must be retained. This means that if the table contains an answer `p(X,Y,D)` for any `X` and `Y` after the program has been executed, then `D` must be the length of the shortest path from `X` to `Y`. Instead of table modes, XSB uses lattice and partial order answer subsumption modes, which allow the user to specify an arbitrary predicate (subject to some mild conditions) to aggregate answers.

Using answer subsumption can yield very compact and efficient programs for optimisation problems, especially those that are instances of Dynamic Programming [6].

Unfortunately, none of the existing implementations that we are aware of are generally sound. Consider the following pure logic program:

```
p(0). p(1).
p(2) :- p(X), X = 1.
p(3) :- p(X), X = 0.
```

The query `?-p(X)` has the finite set of answers `p(0)`,`p(1)`,`p(2)`,`p(3)`, the largest of which is `p(3)`. However, XSB, Yap and B-Prolog all yield different (invalid) solutions when answer subsumption is used to obtain the maximal value. Both XSB and B-Prolog yield `X = 2`, with a maximum aggregation and max table mode respectively. Yap (also with max table mode) yields `X = 0; X = 1; X = 2`, every solution except the right one.

The problem is exacerbated by the fact that none of the systems formally define the semantics of answer subsumption. In a recent ICLP paper [20], we try to resolve this issue by giving a formal semantics for answer subsumption. We then examine under which conditions the systems are sound according to this semantics. Please see Section 3.1.1 for a short overview.

3 Objectives

The research mostly proceeds along two tracks: (1) we investigate the connection between functional programming and tabling with and without answer subsumption; (2) we investigate probabilistic logic programming–as embodied by the ProbLog system–from the functional perspective, in order to develop a general semantics for probabilistic programming languages. The semantics of probabilistic programs directly depends on the least-fixed point semantics mentioned above. Tabling approximates these semantics, and therefore frequently appears as an aspect of these probabilistic programming languages.

3.1 Current Status of the Research and Preliminary Results

3.1.1 Tabling with Sound Answer Subsumption

As mentioned in Section 2, Answer Subsumption, while very useful in practice, lacks a formally defined semantics, which hampers the user's ability to reason about the behaviour of his or her programs. In fact, without a formal semantics, we are reduced to reasoning based on intuitive knowledge of the implementation of a particular system, which is distinctly unportable, and even less satisfying.

In very recent work [20], we have attempted to mitigate this problem by defining what we believe is an appropriate denotational semantics, based on least fixed points of monotone functions on complete lattices. A *complete lattice* is a partially ordered set (poset) $\langle L, \leq_L \rangle$ such that every $X \subseteq L$ has a least upper bound $\bigvee X$, i.e.:

$$\forall z \in L : \bigvee X \leq_L z \iff \forall x \in X : x \leq_L z\,.$$

It is a well known result from lattice theory that least fixed points of monotone functions are guaranteed to exist. Our semantics is based on Van Emden's well known least fixed point semantics, which uses an immediate consequence operator $T_P : \mathcal{P}\left(H_P\right) \to \mathcal{P}\left(H_P\right)$, where H_P is the *Herbrand base*, the set of all ground atoms of a program P. Then the logical semantics (for definite programs, that is programs not containing negations) is given by its least fixed point, $\mathsf{lfp}(T_P)$.

In our work, we define a similar operator $\widehat{T}_P : \mathcal{P}\left(H_P\right) \to \mathcal{P}\left(H_P\right)$, that takes answer subsumption into account. We do so by showing that most tabling modes can be modelled by a semi-lattice L, with functions $\eta : H_P \to L$ and $\rho : L \to \mathcal{P}\left(H_P\right)$ to convert between ground atoms and L. The semantics of a tabled logic program using answer subsumption is then given by

$$\rho\left(\bigvee_{x \in \mathsf{lfp}(\widehat{T}_P)} \eta(x)\right)\,.$$

That is, we take the least fixed point of $\widehat{T}_P$, then convert this least fixed point to the lattice L where we aggregate it, and finally convert this aggregate to a set of ground atoms. It is important to note that we assume that the program is *stratified*, and $\widehat{T}_P$ is operating on a single stratum. The full details are beyond the scope of this text.

Finally, note that the semantics we have specified differs in an important way from actual subsumption implementations: this semantics only aggregates and subsumes answers *after* the least fixed point has been computed, while implementations generally execute subsumption in lock-step with the derivation of new answers.

In the paper we prove a theorem that specifies when an implementation is sound, i.e. when the difference alluded to above, does not produce different answers. Using the theorem requires that a programmer proves certain properties about their program, which may be difficult for realistically sized programs. Nevertheless, we believe this is an important first step towards formalisation of answer subsumption.

3.1.2 Fixing Non-determinism

In a recent paper [21] we reduce tabling (with and without answer subsumption) to its functional essence. Two key elements remain: recursion and non-determinism. This has the advantage, for instance, that this presentation is not muddled by *answer variance*: Prolog systems must avoid adding an answer if there is already a *variant* of the answer in the table.

Most languages don't have unification (and therefore no notion of variance), thus answer variance is an irrelevant detail that can be ignored.

The contributions of this work are:

- We define a monadic model that captures both non-determinism and recursion. This yields a finite representation of recursive non-deterministic expressions. We use this representation as a light-weight (for the programmer) embedded Domain Specific Language to build non-deterministic expressions in Haskell.
- We give a denotational semantics of the model in terms of the least fixed point of a semantic function $\mathcal{R}[\![\cdot]\!]$. In fact, the semantics closely resembles the T_P operator mentioned previously. The semantics is subsequently implemented as a Haskell function that interprets the model.
- We generalise the denotational semantics to arbitrary complete lattices. We illustrate the added power on a simple graph problem, which could not be solved with the more specific semantics. This new semantics corresponds to tabling with answer subsumption.
- We provide a set of benchmarks to demonstrate the expressivity of our approach and evaluate the performance of our implementation.

3.2 Open Issues and Expected Achievements

3.2.1 Automatic Verification of Sound Answer Subsumption

In the paper mentioned in Section 3.1.1, we define a high-level semantics for answer subsumption based on lattice theory. Then we generalise it to establish a correctness condition indicating when it is safe to use (greedy) answer subsumption implemented by most tabling systems. We show several examples where the existing implementations of answer subsumption fail that condition and derive an erroneous result.

This condition is sufficient, but not necessary: there exist programs that do not satisfy the condition, for which the greedy strategy nevertheless delivers correct results. Since we have not run across any non-contrived examples of such programs, we believe that this apparent lack of necessity is an artefact of our rather coarse semantics, which we intend to refine in the future.

The verification of correctness constitutes a non-trivial effort. Hence, manually proving the correctness condition for realistically sized programs could be unfeasible in practice. Ideally we would have an automated analysis that warns the programmer if it fails to establish the correctness condition.

One promising avenue of research is the fact that the program needs to be stratified, and the correctness condition need only hold for the stratum under consideration.

Currently the stratification is also rather coarse. A more fine-grained stratification should significantly reduce the work involved in proving the correctness condition. For automation purposes, abstract interpretation [1] could be used to statically inspect a program, or if we relax our requirements, a dynamic approach could be taken, that warns the programmer that the obtained answers are unreliable during or after the execution of the program

3.2.2 Algebraic Structures for Probabilistic Programming

Within the functional programming community the use of monads for probabilistic programming is both pragmatic and more theoretical. On the one hand several people, e.g. Scibior et al. [17] have developed efficient monadic interfaces for well-known probabilistic inference algorithms. A functional programmer can then use these interfaces to model a probabilistic problem monadically. On the other, there has been much research towards a measure theoretic formalisation of such monadic probabilistic programs [14].

However, little has been done into other more general algebraic structures related to monads. In particular, so called *applicative functors* or *idioms* [9] appear to model precisely those programs where the structure of the program is static, with respect to probabilistic choices that are made. This is especially relevant for probabilistic logic systems such as ProbLog, where the structure of the clauses is fixed. Moreover, as these structures are more restrictive, they may actually admit faster inference routines. Or conversely, ProbLog's specialised inference may apply to probabilistic programming languages that have applicative structure. There are already some promising early results, for instance, it is cleary that ProbLog programs exhibit applicative structure. However, the implications of these results are not yet fully understood, and are subject of ongoing research.

Acknowledgements. I would like to thank the reviewers for taking the time to provide insightful comments and in-depth remarks.

References

1 Samson Abramsky and Chris Hankin. *Abstract Interpretation of declarative languages*, volume 1, chapter An introduction to abstract interpretation, pages 63–102. 1987.

2 Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. ProbLog: A probabilistic prolog and its application in link discovery. In *IJCAI*, volume 7, pages 2462–2467, 2007.

3 Benoit Desouter, Marko van Dooren, and Tom Schrijvers. Tabling as a library with delimited control. *TPLP*, 15(4-5):419–433, 2015. doi:10.1017/S1471068415000137.

4 Jeremy Gibbons and Ralf Hinze. Just do it: simple monadic equational reasoning. In *ICFP*, pages 2–14. ACM, 2011.

5 Noah D. Goodman, Vikash K. Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. Church: a language for generative models. In *UAI*, pages 220–229. AUAI Press, 2008.

6 Hai-Feng Guo and Gopal Gupta. Simplifying dynamic programming via tabling. In *PADL*, volume 3057 of *LNCS*, pages 163–177. Springer, 2004.

7 Oleg Kiselyov, Amr Sabry, and Cameron Swords. Extensible effects: an alternative to monad transformers. In *Haskell*, pages 59–70. ACM, 2013.

8 Marta Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *CAV*, volume 6806, pages 585–591. Springer, 2011.

9 Conor McBride and Ross Paterson. Applicative programming with effects. *JFP*, 18(1):1–13, 2008.

10 Andrew McCallum, Karl Schultz, and Sameer Singh. FACTORIE: probabilistic programming via imperatively defined factor graphs. In *NIPS*, pages 1249–1257. Curran Associates, Inc., 2009.

11 Microsoft Research. Infer.NET. URL: http://research.microsoft.com/en-us/um/cambridge/projects/infernet/.

12 Eugenio Moggi. Notions of computation and monads. *Information and computation*, 93(1):55–92, 1991.

13 Avi Pfeffer. Figaro: An object-oriented probabilistic programming language. *Charles River Analytics Technical Report*, 137, 2009.

14 Norman Ramsey and Avi Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *POPL*, pages 154–165. ACM, 2002.

15 Amr Hany Saleh. Transforming delimited control: Achieving faster effect handlers. In *ICLP (Technical Communications)*, volume 1433 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2015.

16 Vítor Santos Costa, Ricardo Rocha, and Luís Damas. The YAP Prolog system. *TPLP*, 12(1-2):5–34, 2012.

17 Adam Ścibior, Zoubin Ghahramani, and Andrew D Gordon. Practical probabilistic programming with monads. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell*, pages 165–176. ACM, 2015.

18 Terrance Swift and David S. Warren. Tabling with answer subsumption: Implementation, applications and performance. In *LAI*, volume 6341 of *LNCS*, pages 300–312. Springer, 2010.

19 Terrance Swift and David S. Warren. XSB: Extending Prolog with tabled logic programming. *TPLP*, 12(1-2):157–187, January 2012.

20 Alexander Vandenbroucke, Maciej Piróg, Benoit Desouter, and Tom Schrijvers. Tabling with sound answer subsumption. *arXiv preprint arXiv:1608.00787*, 2016.

21 Alexander Vandenbroucke, Tom Schrijvers, and Frank Piessens. Fixing non-determinism. In *IFL 2015: Symposium on the implementation and application of functional programming languages Proceedings*. Association for Computing Machinery, 2016.

22 Philip Wadler. Monads for functional programming. In *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*, pages 24–52. Springer, 1995.

23 David S. Warren. Programming in Tabled Prolog, 1999. URL: `http://www3.cs.stonybrook.edu/~warren/xsbbook/`.

24 Jan Wielemaker, S Ss, and I Ii. Swi-prolog 2.7-reference manual, 1996.

25 Frank Wood, Jan Willem van de Meent, and Vikash Mansinghka. A new approach to probabilistic programming inference. In *Proceedings of the 17th International conference on Artificial Intelligence and Statistics*, pages 1024–1032, 2014.

26 Neng-Fa Zhou. The language features and architecture of B-Prolog. *TPLP*, 12(1-2):189–218, 2012.

Methods for Solving Extremal Problems in Practice

Michael Frank

Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, Israel
frankm@cs.bgu.ac.il

Abstract

During the 20th century there has been an incredible progress in solving theoretically hard problems in practice. One of the most prominent examples is the DPLL algorithm and its derivatives to solve the Boolean satisfiability problem, which can handle instances with millions of variables and clauses in reasonable time, notwithstanding the theoretical difficulty of solving the problem.

Despite this progress, there are classes of problems that contain especially hard instances, which have remained open for decades despite their relative small size. One such class is the class of extremal problems, which typically involve finding a combinatorial object under some constraints (e.g, the search for Ramsey numbers). In recent years, a number of specialized methods have emerged to tackle extremal problems. Most of these methods are applied to a specific problem, despite the fact there is a great deal in common between different problems.

Following a meticulous examination of these methods, we would like to extend them to handle general extremal problems. Further more, we would like to offer ways to exploit the general structure of extremal problems in order to develop constraints and symmetry breaking techniques which will, hopefully, improve existing tools. The latter point is of immense importance in the context of extremal problems, which often hamper existing tools when there is a great deal of symmetry in the search space, or when not enough is known of the problem structure. For example, if a graph is a solution to a problem instance, in many cases any isomorphic graph will also be a solution. In such cases, existing methods can usually be applied only if the model excludes symmetries.

1998 ACM Subject Classification D.3.2 Language Classifications

Keywords and phrases Extremal Problems, Constraints, SAT Solving, Logic Programming, Parallelism

Digital Object Identifier 10.4230/OASIcs.ICLP.2016.21

1 Introduction

A Fundamental research topic in Computer Science is that of combinatorics. Specifically that of finite combinatorial objects, such as finite graphs, finite groups, and circuits. Many of the problem instances which arise in these fields tend to be theoretically intractable, though they are often solvable in practice.

However, among such instances, there are many small, yet notoriously difficult structures to "crack": objects the understanding of which still eludes us in both theory and practice. Several prominent examples include: characterizing and finding Ramsey numbers [21], finding optimal size/depth sorting networks [16, 8], determining the complexity of XOR-AND circuits [1, 2, 23, 4], graph enumeration under constraints [17, 18] (e.g, limited girth, cuts, coloring, etc.) and forbidden-graph characteristics.

Broadly speaking: extremal combinatorics and extremal graph theory are the fields of research which examine finite combinatorial and graph problems the solutions of which usually have to satisfy some restrictions (such as the problems presented above). The problems associated with these fields are collectively referred to as extremal problems.

Historically, there are many techniques, which facilitate solving intractable (usually NP-Hard problem instances), in reasonable time. In recent years, two such techniques came to focus: constraint based techniques and iterative techniques. These techniques led to a plethora of constraints solvers [19, 14], SAT solvers [9, 22], graph iterators [17] and additional applications that can solve an abundance of theoretically hard problem instances in practical scenarios.

While these techniques can be extremely powerful, many times they are not enough to solve extremal problem instances on their own, evident by the lackluster progress with extremal problems. A prominent example comes from the search for Ramsey numbers – where only a handful of exact values are known for small instances [21]. Mathematician Paul Erdős was quoted as saying about the calculation of the Ramsey number $R(5,5)$:

"Suppose aliens invade the earth and threaten to obliterate it in a year's time unless human beings can find the Ramsey number for red five and blue five. We could marshal the world's best minds and fastest computers, and within a year we could probably calculate the value. If the aliens demanded the Ramsey number for red six and blue six, however, we would have no choice but to launch a preemptive attack."

"Ramsey Theory" by Ronald L. Graham and Joel H. Spencer, Scientific American (July 1990), pp. 112–117

Since the introduction of Ramsey numbers in 1930, and twenty five years after the quotation above, the precise value of $R(5,5)$ remains an open problem.

During the past few years, however, we are witnessing the rise of new methodologies, which enable us to better understand and solve extremal problems. Indeed, in the last two decades several prominent extremal problems, many of which have been open for over 50 years – were closed. Such problems include e.g, the computation of Ramsey numbers $R(4,3,3)$ [6], $R(4,5)$ [18], an improved lower bound for $R(4,8)$ [13], size optimal sorting networks for 9 and 10 inputs [5], depth optimal sorting networks for 17 inputs [3], improved lower bounds for circuit complexity of XOR-AND circuits for 5 and 7 inputs [4].

These new methodologies include on the one hand – improvements to existing techniques and theory of extremal problems, and on the other hand – the development of new, more sophisticated, albeit specific techniques aimed towards particular extremal problems. These techniques include e.g, SAT solving [9, 22], symmetry breaking [7, 15], abstraction [6], and parallelism [5]. Note that in many cases, extremal problems are NP-Hard, or Σ_2^P-Hard, which in part explains their difficulty.

During the past two years we have made contributions in the area of extremal combinatorial and graph problems. In particular in exploring extremal circuit problems (e.g, sorting networks and AND-XOR circuit complexity), and in the computation of multi-color Ramsey numbers. We propose to expand and generalize domain specific methods in order to solve general problems in the fields of extremal combinatorics and graph theory. Our initial goal is to expand on the techniques discussed in e.g [4, 15, 6, 5, 7, 20, 11, 10] in order to solve more difficult instances, and eventually develop generalized techniques which can be applied across extremal problems. Further more, we propose to exploit problem structure in order to employ optimized solving algorithms such as those discussed in [20].

2 Research Progress

Seminal to our work is the integration of two methods: (1) The *Generate & Test* method and (2) The *Constrain & Generate* method. With the *generate and test* method, one explicitly enumerates over all solutions, pruning undesired results, and checking each for a given property. Whereas with the *constrain and generate* approach, one typically encodes the problem for some general-purpose discrete satisfiability solver (i.e. SAT, integer programming, constraint programming), which does the enumeration implicitly, and outputs a result. One of the keys to our approach, is to combine these two methods. Using a generate and test algorithm to produce partial solutions, which are then encoded individually, and solved independently (and in parallel).

Moreover, in both the generate and test, and constrain and generate methods, structural knowledge of the problem as well as symmetry breaking techniques have been employed (e.g, [7, 15, 6]) to facilitate the search and limit the search space.

The following subsections present a short summary of work based on these methods. The first three present previously unknown results in the field of extremal problems, which rely on these methods, and the fourth subsection present a tool implemented to aid in the use of these methods.

2.1 Problem 1: Optimal Size Sorting Networks

In [5], we present a computer-assisted non-existence proof of 9-input sorting networks consisting of 24 comparators, thus showing that the 25-comparator sorting network found by Floyd in 1964 is optimal. As a corollary, the 29-comparator network found by Waksman in 1969 is also shown to be optimal for sorting 10 inputs. This proof employs three primary techniques that, although specific to sorting networks, also appear in some form in the problems discussed further in Sections 2.2 and 2.3.

2.2 Problem 2: Multi-Color Ramsey Number $R(4, 3, 3) = 30$

In [6], we present a computer-assisted non-existence proof of the multi-color Ramsey graph $(4, 3, 3)$ with 30 vertices, thus establishing that $R(4, 3, 3) = 30$. The problem of finding $R(4, 3, 3)$ has been characterized as the one with the best chances of being found “soon”. Nevertheless, the precise value of $R(4, 3, 3)$ has remained unknown for nearly 50 years. The proof employs two main techniques: *abstraction* and *symmetry breaking*, in order to first prune the search space and then split it into manageable pieces. Both techniques have a great deal in common with techniques explained in Section 2.1 and the ones discussed in Section 2.3. We believe that these techniques can either be generalized or integrated, as discussed in Section 3.

2.3 Problem 3: AND-XOR Circuit Complexity

In [4] we present a computer-assisted proof that a Boolean function on 7 inputs with multiplicative complexity of at least 7 exists. The multiplicative complexity of a function is a measure of its non-linearity, and is of particular interest in the fields of cryptographic cipher analysis, the study of hash functions, and the study of communication complexity of multiparty computations. The results presented in this chapter rely on examining the topologies of XOR-AND circuits, which are equivalent to Boolean functions, and eventually applying the pigeonhole principle to show that there must exist a function with multiplicative complexity of 7. Three primary techniques are described, that we believe may be explored further as discussed in Chapter 3.

2.4 `pl-nauty` & `pl-gtools`

In [12] we introduce the `pl-nauty` & the `pl-gtools` libraries, which integrate the `nauty` graph automorphism tool with Prolog, thereby allowing Logic Programming to interface with `nauty`. Adding the capabilities of `nauty` to Prolog combines the strength of the "generate and prune" approach that is commonly used in logic programming and constraint solving, with the ability to reduce symmetries while reasoning over graph objects. Moreover, it enables the integration of `nauty` in existing tool-chains, such as SAT-solvers or finite domain constraints compilers which exist for Prolog.

3 Future Work

We are currently looking into two generalizations of the problems presented in sections 2.1, 2.2, and 2.3.

3.1 The Subsumption Problem

The subsumption problem is to determine whether given two sets $A, B \subseteq \{0,1\}^n$, there exists a permutation $\pi\colon [n] \to [n]$ such that $\pi(A) \subseteq B$, where $\pi(A) = \{\pi(x) : x \in A\}$.

The subsumption problem arises when solving the sorting network problem mentioned in section 2.1, and it has close ties to the sub-graph isomorphism problem. A better understanding of this problem will hopefully lead to a better algorithm for solving it. A generalized algorithm for subsumption may be used to generate arbitrary monotone Boolean functions, as well as allow the methods in [5] to be generalized for larger sizes of sorting networks.

We are currently exploring the structural information that can be obtained from A and B in order to perform a quicker subsumption test, much in the vein of `nauty`.

3.2 Abstraction & Concretization for Coloring Problems

Many graph coloring problems are often given as a predicate φ such that the free variables of φ correspond to an adjacency matrix A with domain $0 \cup [k]$, and a satisfying assignment to $\varphi(A)$ implies the sought after coloring. Such problems are often notoriously difficult to solve, such as the case with e.g, the Ramsey coloring problem, variations of the Latin square and magic square problems, multi-color n-queens and more.

▶ **Definition 1** ((weak) isomorphism of graph colorings)**.** Let (G, κ_1) and (H, κ_2) be k-color graph colorings with $G = ([n], E_1)$ and $H = ([n], E_2)$. We say that (G, κ_1) and (H, κ_2) are weakly isomorphic, denoted $(G, \kappa_1) \approx (H, \kappa_2)$ if there exist permutations $\pi\colon [n] \to [n]$ and $\sigma\colon [k] \to [k]$ such that $(u, v) \in E_1 \iff (\pi(u), \pi(v)) \in E_2$ and $\kappa_1((u, v)) = \sigma(\kappa_2((\pi(u), \pi(v))))$. We extend the notation for the adjacency matrices of colorings and denote $A \approx B$ for the adjacency matrices A, B of $(G, \kappa_1) \approx (H, \kappa_2)$.

A graph coloring problem is said to be $\approx$-closed (i.e, closed under $\approx$ relation) if the following definition hold:

▶ **Definition 2** (≈-closed graph coloring problem). Let φ a graph coloring problem. φ is said to be ≈-closed if for all $(G_1, \kappa_1) \approx (G_2, \kappa_2)$ with adjacency matrices A_1 and A_2 respectively it holds that $\varphi(A_1) \iff \varphi(A_2)$. Alternatively:

$$(G_1, \kappa_1) \approx (G_2, \kappa_2) \iff (\varphi(A_1) \iff \varphi(A_2)) .$$

In [6] we present the method of *abstraction* and *concretization* for graph coloring problems. Although this method was developed specifically to solve Ramsey instances, it may be applied to any graph coloring problem closed under the weak isomorphism property. The degree matrix of coloring A is a matrix M such that $M_{i,j}$ is the number of j colored neighbour of node i. The *abstraction* of an adjacency matrix A is the lexicographically sorted degree matrix M of A, and denoted $\alpha(A)$, and that the *concretization* of M are all the adjacency matrices whose abstraction is M, denoted $\gamma(M)$. Notice also that:

▶ **Lemma 3.** *$A \approx A'$ if and only if $\alpha(A) = \alpha(A')$.*

Now, using observation 3, the search space of any graph coloring problem may be described in terms of the abstraction of degree matrices. The method then computes an over-approximation of possible solutions and uses that to guide the search for an actual solution (should one exists).

Notice that many classic coloring problems are closed under this relation e.g, Latin squares, Ramsey colorings, multi-colored n-queens. Therefore, it may be conceivable that the abstraction and concretization of graphs presented in [6], may be generalized for coloring problems which are ≈-closed.

References

1 Joan Boyar and René Peralta. Tight bounds for the multiplicative complexity of symmetric functions. *TCS*, 396(1–3):223–246, 2008.

2 Joan Boyar, René Peralta, and Denis Pochuev. On the multiplicative complexity of Boolean functions over the basis $(\wedge, +, 1)$. *TCS*, 235(1):43–57, 2000.

3 Daniel Bundala and Jakub Zavodny. Optimal sorting networks. In Adrian Horia Dediu, Carlos Martín-Vide, José Luis Sierra-Rodríguez, and Bianca Truthe, editors, *Language and Automata Theory and Applications – 8th International Conference, LATA 2014, Madrid, Spain, March 10-14, 2014. Proceedings*, volume 8370 of *Lecture Notes in Computer Science*, pages 236–247. Springer, 2014. `doi:10.1007/978-3-319-04921-2_19`.

4 Michael Codish, Luís Cruz-Filipe, Michael Frank, and Peter Schneider-Kamp. When six gates are not enough. *CoRR*, abs/1508.05737, 2015. URL: `http://arxiv.org/abs/1508.05737`.

5 Michael Codish, Luís Cruz-Filipe, Michael Frank, and Peter Schneider-Kamp. Sorting nine inputs requires twenty-five comparisons. *Journal of Computer and System Sciences*, 82(3):551–563, 2016. `doi:10.1016/j.jcss.2005.06.002`.

6 Michael Codish, Michael Frank, Avraham Itzhakov, and Alice Miller. Computing the ramsey number r(4, 3, 3) using abstraction and symmetry breaking. *CoRR*, abs/1510.08266, 2015. URL: `http://arxiv.org/abs/1510.08266`.

7 Michael Codish, Alice Miller, Patrick Prosser, and Peter James Stuckey. Breaking symmetries in graph representation. In Francesca Rossi, editor, *Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China*. IJCAI/AAAI, 2013. URL: `http://www.aaai.org/ocs/index.php/IJCAI/IJCAI13/paper/view/6480`.

8 Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

9 Niklas Eén and Niklas Sörensson. Minisat sat solver. URL: `http://minisat.se/Main.html`.

10 Thorsten Ehlers and Mike Müller. Faster sorting networks for 17, 19 and 20 inputs. *CoRR*, abs/1410.2736, 2014. URL: `http://arxiv.org/abs/1410.2736`.

11 Thorsten Ehlers and Mike Müller. New bounds on optimal sorting networks. *CoRR*, abs/1501.06946, 2015. URL: `http://arxiv.org/abs/1501.06946`.

12 M. Frank and M. Codish. Logic programming with graph automorphism: Integrating nauty with prolog (a tool paper). Technical report, Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, Israel, 2016. URL: `https://www.cs.bgu.ac.il/~frankm/plnauty/`.

13 Hiroshi Fujita. A new lower bound for the ramsey number r(4, 8). *CoRR*, abs/1212.1328, 2012. URL: `http://arxiv.org/abs/1212.1328`.

14 M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and M. Schneider. Potassco: The Potsdam answer set solving collection. *AI Communications*, 24(2):107–124, 2011.

15 Avraham Itzhakov and Michael Codish. Breaking symmetries in graph search with canonizing sets. *CoRR*, abs/1511.08205, 2015. URL: `http://arxiv.org/abs/1511.08205`.

16 Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

17 B. McKay. *nauty* user's guide (version 1.5). Technical Report TR-CS-90-02, Australian National University, Computer Science Department, 1990.

18 Brendan D. McKay and Stanislaw P. Radziszowski. $R(4, 5) = 25$. *Journal of Graph Theory*, 19(3):309–322, 1995. `doi:10.1002/jgt.3190190304`.

19 Amit Metodi and Michael Codish. Compiling finite domain constraints to sat with bee. *Theory and Practice of Logic Programming*, 12(4-5):465–483, 2012.

20 Amit Metodi, Michael Codish, and Peter J. Stuckey. Boolean equi-propagation for concise and efficient SAT encodings of combinatorial problems. *J. Artif. Intell. Res. (JAIR)*, 46:303–341, 2013. `doi:10.1613/jair.3809`.

21 Stanislaw P. Radziszowski. Small Ramsey numbers. *Electronic Journal of Combinatorics*, 1994. Revision #14: January, 2014. URL: `http://www.combinatorics.org/`.

22 Mate Soos. CryptoMiniSAT, v2.5.1, 2010. URL: `http://www.msoos.org/cryptominisat2`.

23 Meltem Sönmez Turan and René Peralta. The multiplicative complexity of Boolean functions on four and five variables. In Thomas Eisenbarth and Erdinç Öztürk, editors, *LightSec 2014*, volume 8898 of *LNCS*, pages 21–33. Springer, 2015.

Automating Disease Management Using Answer Set Programming

Zhuo Chen

University of Texas at Dallas, Dallas, Texas, USA
zxc130130@utdallas.edu

Abstract

Management of chronic diseases such as heart failure, diabetes, and chronic obstructive pulmonary disease (COPD) is a major problem in health care. A standard approach that the medical community has devised to manage widely prevalent chronic diseases such as chronic heart failure (CHF) is to have a committee of experts develop guidelines that all physicians should follow. These guidelines typically consist of a series of complex rules that make recommendations based on a patient's information. Due to their complexity, often the guidelines are either ignored or not complied with at all, which can result in poor medical practices. It is not even clear whether it is humanly possible to follow these guidelines due to their length and complexity. In the case of CHF management, the guidelines run nearly 80 pages. In this paper we describe a physician-advisory system for CHF management that codes the entire set of clinical practice guidelines for CHF using answer set programming. Our approach is based on developing reasoning templates (that we call knowledge patterns) and using these patterns to systemically code the clinical guidelines for CHF as ASP rules. Use of the knowledge patterns greatly facilitates the development of our system. Given a patient's medical information, our system generates a recommendation for treatment just as a human physician would, using the guidelines. Our system will work even in the presence of incomplete information. Our work makes two contributions: (i) it shows that highly complex guidelines can be successfully coded as ASP rules, and (ii) it develops a series of knowledge patterns that facilitate the coding of knowledge expressed in a natural language and that can be used for other application domains.

1998 ACM Subject Classification D.1.6 Logic Programming, I.2.3 Logic Programming

Keywords and phrases chronic disease management, knowledge pattern, answer set programming, knowledge representation, automated reasoning

Digital Object Identifier 10.4230/OASIcs.ICLP.2016.22

1 Introduction and problem description

Chronic diseases are health conditions that can neither be prevented nor be cured but can only be managed. They have been the major consumer of health-care funds throughout the world. In America alone there are more than 133 million people – which is more than 40% of the U.S. population – who suffer from one or more chronic diseases [17]. In the U.S. they account for 81% of hospital admissions, 91% of prescriptions filled and 76% of all physician visits [1]. Though the list of chronic conditions is long, the top five conditions are: heart disease, cancers, stroke, chronic obstructive pulmonary disease (COPD) and diabetes.

In 2010, 68% of the healthcare spending – more than trillion dollars – went towards the treatment of chronic diseases [5]. The successful management of chronic diseases has two components: (i) self-management by the patients, and (ii) management by physicians while adhering to strict guidelines. The failure of either component will lead to the failure of the whole enterprise for the management of chronic diseases.

In our research, we focus on the second component of CHF management, namely, a Physician Advisory System. This system assists physicians in adhering to the guidelines for managing CHF. The CHF management guidelines are published by the American College of Cardiology Foundation (ACCF) and the American Heart Association (AHA). The most recent version is the 2013 ACCF/AHA Guideline for the Management of Heart Failure [18]. These guidelines were created by a committee of physicians based on thorough review of clinical evidence on heart failure management. They represent a consensus among the physicians on the appropriate treatment and management of heart failure [11]. Though evidence-based guidelines should be the basis for all disease management [6], physicians' adherence to guidelines is very poor [4]. The major reasons for the failure of guideline implementation are lack of awareness, lack of familiarity, lack of motivation and external barriers. For 78% of clinical practice guidelines, more than 10% of the physicians are not aware of their existence. Even when the guidelines are readily accessible, the physicians are not familiar enough with the guidelines to apply them correctly. In all the physician surveys conducted, the lack of familiarity was more common than the lack of awareness [4].

One of the reasons for the lack of familiarity is that the guidelines can be quite complex, as in the case of CHF management. For example, more than 100 variables have been associated with mortality and re-hospitalization related to heart failure. In the 2013 ACCF/AHA Guideline for the Management of Heart Failure, the variables range from simple information like age and sex to sophisticated data like the patterns in electrocardiogram and history of CHF-related symptoms and diseases. To overcome the difficulties that physicians face in implementing the guidelines, we have developed a Physician Advisory System that automates the 2013 ACCF/AHA Guideline for the Management of Heart Failure. Our physician advisory system is able to give recommendations like a real human physician who is following the guidelines strictly, even under the condition of incomplete information about the patient. Our physician-advisory system for CHF management relies on answer set programming [9, 3] for coding the guidelines. The guideline rules are fairly complex and require the use of negation as failure, non-monotonic reasoning and reasoning with incomplete information. A fairly common situation in medicine is that a drug can only be recommended if its use is not contraindicated (i.e., the use of the drug will not have an adverse impact on that patient). Contraindication is naturally modeled via negation as failure. The ability of answer set programming to model defaults, exceptions, weak exceptions, preferences, etc., makes it ideally suited for coding these guidelines.

2 Background and overview of the existing literature

A large number of software systems have been designed to address CHF. However, none of them are designed to automatically advise physicians based on the ACCF/AHA guidelines. Chronic disease management systems designed thus far fall into seven categories [12]: accessibility, care management, point-of-care functions, decision support, patient self-management, population management, and reporting. The automation of these functionalities is certainly helpful in assisting health care providers with managing patients with chronic conditions, however, none of them cover what we have realized: a physician advisory system that automates the application of clinical practice guidelines.

The 2013 ACCF/AHA Guideline for the Management of Heart Failure is intended to assist healthcare providers in clinical decision making by describing a range of generally acceptable approaches for the management of chronic heart failure. The guideline is based on four progressive stages of heart failure. Stage A includes patients at risk of heart failure who are asymptomatic and do not have structural heart disease. Stage B describes asymptomatic patients with structural heart diseases; it includes New York Heart Association (NYHA) class I, in which ordinary physical activity does not cause symptoms of heart failure. Stage C describes patients with structural heart disease who have prior or current symptoms of heart failure; it includes NYHA class I, II (slight limitation of physical activity), III (marked limitation of physical activity) and IV (unable to carry on any physical activity without symptoms of heart failure, or symptoms of heart failure at rest). Stage D describes patients with refractory heart failure who require specialized interventions; it includes NYHA class IV. Interventions at each stage are aimed at reducing risk factors (stage A), treating structural heart disease (stage B) and reducing morbidity and mortality (stages C and D) [18].

Traditional techniques such as logic programming (Prolog) and production systems (OPS5), or traditional expert system styled approaches will result in a far more complex system due to the inability of these systems to model negation as failure effectively [2]. Thus, coding our system in these formalisms would be a much more difficult and complex task. In contrast, the CHF guidelines can be coded in ASP very naturally (it took about 2 months to develop the first version of the system).

3 Goal of research

We selected Chronic Heart Failure (CHF) as our first chronic disease to build tools to manage. Chronic Heart Failure is the inability of the heart to pump properly; consequently, not enough oxygen-rich blood can be supplied to all parts of the body. This causes congestion of blood in the lungs, abdomen, legs, etc., causing uneasiness while carrying out any kind of physical activity. Our physician advisory system for CHF management codes all the knowledge in the 2013 ACCF/AHA Guideline for the Management of Heart Failure [18] as an answer set program. Our system is able to recommend treatments just like a human physician who is strictly following these guidelines. Additionally, our system is able to recommend treatments even when a patient's information is incomplete. The input to our system is the patient's information which includes demographics, history, daily symptoms, known risk factors, measurements as well as ACCF/AHA stage and NYHA class [18]. A physician uses our system by posing a query to it. Our system then processes the query by essentially simulating the thinking process of a CHF specialist (represented by the ACCF/AHA guideline).

To implement the CHF guidelines in ASP, we first extensively studied the guidelines to extract reasoning templates. These templates can be thought of as general knowledge patterns. These patterns were next deployed to code the CHF guideline rules. Our research makes two major contributions:

1. We develop a system that completely automates the entire set of guidelines for CHF management developed by the American College of Cardiology Foundation and American Heart Association. The system takes its input from (i) a patient's electronic health record that includes demographic information, test results, etc., and (ii) a telemedicine system that provides data about vital signs (heart rate, blood pressure, weight, etc.). It then uses this information to recommend a treatment. The s(ASP) system also supports abduction, thus our system can also be used for abductive reasoning: a physician can, for example, figure out the symptoms that a particular patient must have in order for a given treatment to work.
2. We develop a set of general knowledge patterns that were used to realize our automated physician-advisory system and that can be helpful in translating rules expressed in a natural language into ASP for any application domain.

4 Current status of the research

4.1 Physician advisory system description

The physician-advisory system for CHF management has two major components, the rule database and the fact table. The rule database covers all the knowledge in 2013 ACCF/AHA Guideline for the Management of Heart Failure [18]. The fact table contains the relevant information of the patient with heart failure. The fact table is derived from a patient's electronic health record and from a telemedicine system used to measure vital signs. The patient information consists mainly of: 5 pieces of demographics information, 8 measurements and 25 types of HF-related diseases and symptoms. Treatment recommendations returned by the system may include: 11 pharmaceutical treatments, 9 management objectives, and 4 device/surgery therapies.

Our system is designed for running on top of the s(ASP) system, a goal-directed, predicate ASP system that can be thought of as Prolog extended with negation based on the stable model semantics [14]. Because of the goal-directed nature of the system, only the particular treatments applicable to the patient are reported by the system. With minor changes, our system will also work with traditional SAT-based implementations such as CLASP [7, 8]. However, these systems will compute the entire model, so if there are multiple treatments for a given condition, they will all be included in the answer set (these differences between goal-directed and SAT-based solvers are explained in [13]).

4.2 Knowledge patterns in the guidelines for the management of heart failure

The ACCF/AHA guidelines are written in English and are quite complex. Our task was to code these guidelines in ASP. To simplify our task, we developed reasoning templates that we call knowledge patterns. These knowledge patterns are quite general and serve as solid building blocks for systematically translating the specifications written in English to ASP. While developing these knowledge patterns and coding them in ASP, certain facts had to be noted: (i) Multiple rules can lead to the recommendation of a treatment; (ii) Multiple rules can lead to contraindication of a treatment; (iii) A treatment cannot be recommended if at least one contraindication for that treatment is present; and, (iv) A given treatment recommendation can impact the recommendation and/or contraindication of other treatments.

Next, we present the most salient knowledge patterns that we have developed. Many of these patterns are straightforward, however, some of them, such as the concomitant choice rule, are intricate. We present these patterns at a high level and ignore non-essential details.

1. Aggressive Reasoning: The aggressive reasoning pattern can be stated as "take an action (e.g., recommend treatment) if there is a reason; no evidence of danger means there is no danger in taking that action". The aggressive reasoning pattern is coded as follows:

```
recommendation(Choice) :- preconditions(Choice),
not contraindication(Choice).
contraindication(Choice) :- dangers(Choice).
```

The code above makes use of negation as failure. If the contraindication of a choice cannot be proved, and the conditions for making the choice hold, then that choice is recommended. An example of this knowledge pattern can be found in [18]: "Digoxin can be beneficial in patients with HFrEF, unless contraindicated, to decrease hospitalizations for HF."

2. Conservative Reasoning: This reasoning pattern is stated as "A reason for a recommendation is not enough; evidence that the recommendation is not harmful must be available".

The conservative reasoning pattern is coded as follows:

```
recommendation(Choice) :- preconditions(Choice),
not contraindication(Choice).
contraindication(Choice) :- not -dangers(Choice).
```

This coding pattern requires evidence of the absence of danger. Without such evidence, the choice would be considered contraindicated. Note that the "-" operator indicates classical negation. An example of this knowledge pattern can be found in [18]: "In patients with structural cardiac abnormalities, including LV hypertrophy, in the absence of a history of MI or ACS, blood pressure should be controlled in accordance with clinical practice guidelines for hypertension to prevent symptomatic HF."

3. Anti-recommendation: The anti-recommendation pattern is stated as "a choice can be prohibited if evidence of danger can be found".

The coding pattern for the anti-recommendation is coded as follows:

```
contraindication(choice) :- dangers(Choice).
```

The code above specifies the conditions under which a choice will be ruled out (i.e., contraindicated). Note that for a choice to be made, both aggressive reasoning and conservative reasoning require that the contraindication of the choice be false. An example of this knowledge pattern can be found in [18]: "Anticoagulation is not recommended in patients with chronic HFrEF without AF, a prior thromboembolic event, or a cardioembolic source."

4. Preference: The preference pattern is stated as "use the second-line choice when the first-line choice is not available". The preference pattern is coded as follows:

```
recommendation(First_choice) :- conditions_for_both_choices,
not contraindication(First_choice).
recommendation(Second_choice) :- conditions_for_both_choices,
contraindication(First_choice),
not contraindication(Second_choice).
```

This code chooses the treatment recommendation in the second choice only when the conditions are satisfied, the first choice is contraindicated, and there is no evidence preventing the use of second choice. An example of this knowledge pattern can be found in [18]: "ARBs are recommended in patients with HFrEF with current or prior symptoms who are ACE inhibitor intolerant, unless contraindicated, to reduce morbidity and mortality."

5. Concomitant Choice: The concomitant choice pattern is stated as "if a choice is made, some other choices are automatically in effect unless they are prohibited." The concomitant pattern is coded as shown below.

```
recommendation(Trigger_choice) :- preconditions(Trigger_choice),
not contraindication(Trigger_choice),
not skip_concomitant_choice(Trigger_choice).
skip_concomitant_choice(Trigger_choice) :-
not recommendation(Concomitant_choice),
not contraindication(Concomitant_choice).
recommendation(Concomitant_choice) :-
recommendation(Trigger_choice),
not contraindication(Concomitant_choice).
```

The above code makes sure that a concomitant choice appears in all stable models containing the trigger choice, provided the concomitant choice is not prohibited. The trigger choice is always recommended along with the concomitant choice unless the concomitant choice is contraindicated. An example of this knowledge pattern can be found in [18]: "Diuretics should generally be combined with an ACE inhibitor, beta blocker, and aldosterone antagonist. Few patients with HF will be able to maintain target weight without the use of diuretics."

6. Indispensable Choice: The indispensable choice pattern is stated as "if a choice is made, some other choices must also be made; if those choices can't be made, then the first choice is revoked". Note that choosing "Trigger_choice" forces "Indispensable_choice". The indispensable choice pattern is coded as shown below:

```
recommendation(Trigger_choice) :- preconditions(Trigger_choice),
not contraindication(Trigger_choice),
not absent_indispensable_choice(Trigger_choice).
absent_indispensable_choice(Trigger_choice) :-
not recommendation(Indispensable_choice).
recommendation(Indispensable_choice) :- recommendation(Trigger_choice),
not contraindication(Indispensable_choice).
```

The above code makes sure that the trigger choice will always appear with the indispensable choice. If for some reason the indispensable choice cannot be made, then the trigger choice cannot be made either. An example of this knowledge pattern can be found in [18]: "In patients with a current or recent history of fluid retention, beta blockers should not be prescribed without diuretics".

7. Incompatible Choice: The incompatibility pattern is stated as "some choices cannot be in effect at the same time". The incompatible choice pattern is coded as shown below:

```
taboo_choice(Choice_1) :-
recommendation(Choice_2),
...,
recommendation(Choice_n).
taboo_choice(Choice_2) :-
recommendation(Choice_1),
recommendation(Choice_3),
....
recommendation(Choice_n).
...
taboo_choice(Choice_n) :-
recommendation(Choice_1),
recommendation(Choice_2),
....
recommendation(Choice_n-1).
```

```
recommendation(Choice_1) :-
conditions_for_choice_1,
not contraindication(Choice_1),
not taboo_choice(Choice_1).
recommendation(Choice_2) :-
conditions_for_choice_2,
not contraindication(Choice_2),
not taboo_choice(Choice_2).

...
recommendation(Choice_n) :-
conditions_for_choice_n,
not contraindication(Choice_n),
not taboo_choice(Choice_n).
```

{accf_stage(c), hf_with_reduced_ef, history(standard_neurohormonal_antagonist_therapy), nyha_class(3), nyha_class_3_to_4, race(african_american), recommendation(hydralazine_and_isosorbide_dinitrate,class_1), not contraindication(hydralazine_and_isosorbide_dinitrate)}

Figure 1 Result of abductive reasoning in physician-advisory system for CHF management.

The above code makes sure that incompatible choices will not be made together. Note that we did not use a simple constraint to implement this pattern. A constraint would kill all stable models if each of the choices in question can be made. Our implementation, on the other hand, will produce partial answer sets supporting the query, thanks to the goal-driven mechanism of s(ASP) [14]. An example of this knowledge pattern can be found in [18]: "Routine combined use of an ACE inhibitor, ARB, and aldosterone antagonist is potentially harmful for patients with HFrEF."

4.3 Abductive reasoning in the management of heart failure

Our system can also perform abductive reasoning thanks to the s(ASP) system's support for abduction [14]. Abductive reasoning is a form of logical inference where one attempts to augment a theory with sufficient information to explain an observation (the augmentations come from a set of predicates that are declared as *abducibles*). To illustrate, consider the following two rules in the ACCF/AHA guideline [18]:

- Combination of hydralazine & isosorbide dinitrate is recommended to reduce morbidity & mortality for patients self-described as African Americans with NYHA class III-IV HFrEF receiving optimal therapy with ACE inhibitors & and beta blockers, unless contraindicated.
- Combination of hydralazine & isosorbide dinitrate should not be used for the treatment of HFrEF in patients who have no prior use of standard neurohormonal antagonist therapy.

Suppose we have an African American patient who is suffering from NYHA class III HFrEF, but that is all we know about the patient. Since a hydralazine and isosorbide dinitrate combination is highly effective in reducing the mortality of African Americans with HFrEF, the physician might pose the following query:

```
?-recommendation((hydralazine_and_isosorbide_dinitrate), class_1)
```

to the s(ASP) system. The system would return the results shown in Figure 1.

Note that the system abduced two things: (i) a "history of standard neurohormonal antagonist therapy", and (ii) the absence of "contraindication of hydralazine and isosorbide dinitrate". This means in order for us to recommend hydralazine and isosorbide dinitrate to the patient, they must have received standard neurohormonal antagonist therapy before. Otherwise, hydralazine and isosorbide dinitrate would be contraindicated.

5 Preliminary results accomplished

Our system has been tested in-house and has shown accurate results that are compatible with what a physician following the guidelines would conclude. A clinical trial is planned.

The input to the system is a patient's information, including demographics, history, daily symptoms, risks and measurements, as well as ACCF/AHA stage and NYHA class. When queried for a treatment recommendation, our system is able to give recommendations according to the guideline just as a physician would.

```
%doctor's assessments
accf_stage(c).
nyha_class(3).
expectation_of_survival(3).
diagnosis(hypertension).
%demographics of the patient
gender(female).
age(78).
history(mi, recent).
%measurements from the lab
hf_with_reduced_ef.
measurement(creatinine, 1.8).
measurement(potassium, 4.9).
measurement(lvef, 0.35).
measurement(lbbb, 180).
measurement(sinus_rhythm).
```

```
%history of the patient
diagnosis(myocardial_ischemia).
diagnosis(atrial_fibrillation).
diagnosis(coronary_artery_disease).

evidence(ischemic_etiology_of_hf).
evidence(sleep_apnea).
evidence(fluid_retention).

history(stroke).
history(cardiovascular_hospitalization).
post_mi(40).
```

Figure 2 Representation of a patient's information in physician-advisory system for CHF management.

To illustrate how our system works, consider a female heart failure patient who is in ACCF/AHA stage C, belongs to NYHA class 3 and has been diagnosed as myocardial ischemia, atrial fibrillation, coronary artery disease. She also suffers from sleep apnea, fluid retention and hypertension. Her left ventricular ejection fraction (LVEF) is 35%. There is evidence that she has ischemic etiology of heart failure. Her electrocardiogram (ECG) has sinus rhythm and a left bundle branch block (LBBB) pattern with a QRS duration of 180ms. The blood test says her creatinine is 1.8 mg/dL and potassium is 4.9 mEg/L. She has a history of stroke. It has been 40 days since the acute myocardial infarction happened to her. Her doctor assessed that her expectation of survival is about 3 years.

The patient's information derived from her electronic health record is coded as the facts shown in Figure 2. There are multiple treatments for this patient. Figure 3 shows some of the treatment recommendations our system infers once we give the query "recommendation(Treatment, Class)". Each treatment recommendation (represented as a partial answer set) contains all of the predicates that must hold in order for the query to be successful. For instance, consider the recommendation of ace inhibitors as a treatment option (answer #2). Ace inhibitors are recommended because the patient is in ACCF/AHA Stage C, per the doctor's assessment, and has heart failure with reduced ejection fraction condition. Proof of contraindication for ace inhibitors is absent as the patient does not have a history of angioedema (`not history(angiodema)`) and is not pregnant (`not pregnancy`). The system also gives us the concomitant treatments for ace inhibitors, namely, beta blockers and diuretics. It is worth mentioning that we used the aggressive reasoning pattern (see Section 4.2) when coding the rules of ace inhibitors.

Had we adopted the conservative reasoning pattern, ace inhibitors would not have been recommended unless we explicitly asserted `-history(angioedema)` and `-pregnancy` in the patient's information (a definitive proof of the latter can be derived from patient's age (78)).

Given that there may be multiple treatment options for a particular patient, the choice of a particular treatment will depend on the physician's preference. Rules that capture a physician's or a nurse's preference can also be coded as answer set programs in our system.

While our testing indicates that the system works well and the results produced are consistent with what a physician may recommend, if they were to exactly follow the guidelines,

```
{ accf_stage(c), recommendation(sodium_restriction,class_2a), not contraindication(sodium_restriction) } Treatment = sodium_restriction, Class = class_2a

{ accf_stage(c), hf_with_reduced_ef, recommendation(ace_inhibitors,class_1), recommendation(beta_blockers,class_1), recommendation(diuretics,class_1), not contraindication(ace_inhibitors), not contraindication(beta_blockers), not contraindication(diuretics), not history(angioedema), not history(angioedema,recent), not history(angioedema,remote), not pregnancy } Treatment = ace_inhibitors, Class = class_1
```

Figure 3 Output of the physician-advisory system for CHF management.

a clinical trial is needed to truly validate our system, and is indeed planned. As mentioned earlier, our system can be used for abductive reasoning as well. Running the system in the abductive mode can allow a physician to try out what-if scenarios and to make sure that all the pre-conditions required for treatment are met.

6 Open issues and expected achievements

In this paper we report on our work on developing a ASP-based physician advisory system for managing CHF using a telemedicine platform. The system automates the rules laid out in the 2013 ACCF/AHA Guide for the Management of Heart Failure. It is able to take a patient's data as input and produce treatment recommendations that strictly adhere to the guidelines. It can also be used by a physician to abduce symptoms and other conditions that must be met by a given treatment recommendation.

Our approach to developing the system was based on identifying knowledge patterns and using them as building blocks for constructing the ASP code. There are many ways to further extend our work that we plan to pursue in the future:

- Extending the system for comorbidities: We would like our system to handle comorbidities of heart failure [12]. A typical CHF patients suffers from other chronic ailments as well, i.e., CHF generally never occurs by itself.
- Performing clinical trials: our system has been tested in-house, however, we plan to compare the recommendations given by our system to the prescriptions by human cardiologists in a formal clinical trial to validate the effectiveness of our system.
- Integrating with EMRs and a Telemedicine Platform: Future work would include integrating our system with our telemedicine platform so that the input comes directly from the electronic medical record while vital signs are directly obtained from the patient through our telemedicine hardware and software [16, 15]. A user-friendly GUI will also be designed to make the system more usable.
- Adding justification to recommendations given by our system: Although the rationale behind a recommendation is shown in the partial answer set, it is hard to decipher it. We plan to augment s(ASP) [14] so that reasonably detailed justifications for a query are printed in a human-readable form.
- Formal Analysis: Conducting research to formally establish the correctness of our system.

Acknowledgements. I want to thank Dr. Gopal Gupta, my supervisor, to introduce me to both logic programming and answer set programming. I would also like to thank Dr. Lakshman Tamil, my co supervisor, for his wonderful insight into chronic disease management. Finally, I want to thank Dr. Kyle Marple and Mr.Elmer Salazar for the enlightening discussions.

References

1 Gerard Anderson. *Chronic conditions: making the case for ongoing care.* Johns Hopkins University, 2004.

2 Chitta Baral. Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press, 2003.

3 Gerhard Brewka, Thomas Eiter, and Mirosław Truszczyński. Answer set programming at a glance. *Commun. ACM 54,* 12, 92–103, 2011.

4 Michael D. Cabana, Cynthia S. Rand, Neil R. Powe, and et al. Why don't physicians follow clinical practice guidelines?: A framework for improvement. *JAMA 282,* 15, 1458–1465, 1999.

5 Centers for Disease Control and Prevention. Chronic disease overview page. `http://www.cdc.gov/chronicdisease/overview/index.htm`.

6 David P. Faxon, et al. Improving quality of care through disease management principles and recommendations from the American heart association's expert panel on disease management. *Stroke 35,* 6, 1527–1530, 2004.

7 Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. Conflict-driven answer set solving: From theory to practice. In *Artif. Intell. 187*, 52-89, 2012.

8 Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Clingo = ASP + Control: Preliminary Report. In *Proc. ICLP 2014 Proc.*

9 Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. Proc. ICLP 1988. MIT Press, 1070–1080.

10 McDonnell Norms Group. Enhancing the use of clinical guidelines: a social norms perspective. *Journal of the American College of Surgeons 202,* 5, 826–836, 2006.

11 Alice K. Jacobs, Frederick G. Kushner, and et al. ACCF/AHA clinical practice guideline methodology summit report: A report of the American college of cardiology foundation/american heart association task force on practice guidelines. *Journal of the American College of Cardiology 61,* 2, 213–265, 2013.

12 Laura Jantos and Michelle Holmes. *IT Tools for Chronic Disease Management: How Do They Measure Up?* California HealthCare Foundation (chfc.org; retrieved Jan. 2016).

13 Kyle Marple and Gopal Gupta. Goal-directed execution of answer set programs. *Proc. PPDP* 2012: 35-44

14 Kyle Marple, Elmer Salazar, and Gopal Gupta. s(ASP). `https://sourceforge.net/projects/sasp-system/`.

15 Savio Monteiro. An intelligent telemedicine platform with cognitive support for chronic care management. *Ph.D. thesis, Quality of Life Technology Lab, UT Dallas*, 2015.

16 Savio Monteiro, Gopal Gupta, Mehrdad Nourani, and Lakshman S Tamil. Hygeiatel: An intelligent telemedicine system with cognitive support. In *Proceedings of the First ACM Workshop on Mobile Systems, Applications, and Services for Healthcare.* mHealthSys'11. ACM, New York, NY, USA, 9:1–2.

17 Shin-Yi Wu and Anthony Green. Projection of chronic illness prevalence and cost inflation. *Santa Monica, CA: RAND Health*, 2000.

18 Clyde W. Yancy, Mariell Jessup, Biykem Bozkurt, and et al. 2013. 2013 ACCF/AHA guideline for the management of heart failure: A report of the american college of cardiology foundation/American heart association task force on practice guidelines. *Journal of the American College of Cardiology 62,* 16, e147.

Scalable Design Space Exploration via Answer Set Programming[*]

Philipp Wanko

University of Potsdam, Institute for Computer Science, Potsdam, Germany
wanko@cs.uni-potsdam.de

Abstract

The design of embedded systems is becoming continuously more complex such that the application of efficient high level design methods are crucial for competitive results regarding design time and performance. Recently, advances in Boolean constraint solvers for Answer Set Programming (ASP) allow for easy integration of background theories and more control over the solving process. The goal of this research is to leverage those advances for system level design space exploration while using specialized techniques from electronic design automation that drive new application-originated ideas for multi-objective combinatorial optimization.

1998 ACM Subject Classification D.1.6 Logic Programming

Keywords and phrases Answer Set Programming, System Synthesis, Multi-Objective Optimization

Digital Object Identifier 10.4230/OASIcs.ICLP.2016.23

1 Introduction and problem description

Embedded computing systems are application-specific computers. They typically have to satisfy among others real-time, power, and area constraints while being at the same time reliable and cost efficient. These often conflicting design goals can only be met because each embedded computing system is designed for a specific and thus restricted set of applications. Examples of embedded computing systems can be found in smart phones, automation systems, automotive electronics, medical devices, industrial automation systems, train control systems, etc. However, increasing application complexity paired with increasingly complex computing platforms hamper good design decisions and, thus, the optimization of the final product. As a consequence, new tools and methodologies are required, which permit to automatically and effectively explore design options at system level. The goal of this research is to leverage advances in Boolean constraint technology for system level design space exploration. In turn, specialized techniques from electronic design automation drive new application-originated ideas into multi-objective combinatorial optimization.

2 Background and overview of the existing literature

2.1 Design Space Exploration

(DSE) can be performed at various abstraction levels. The goal is always to identify an optimal implementation for the given set of applications. Depending on the level of abstraction, the considered applications can be as complex as a video decoder or as simple as a single logic

[*] This work was partially supported by DFG-SCHA-550/11.

operation. All possible implementations, also called design points, of the given applications define the *design space*, denoted by X. The problem of DSE is twofold [17]: (1) How to *evaluate* the quality of a single design point and (2) how to *cover* the design space during exploration? Again, depending on the abstraction level, the implementation can be as complex as a heterogeneous many-core system or as simple as a single logic gate. Our research targets the electronic system level and assumes a top-down design methodology [14]. At this level, applications are typically of the complexity of video decoders and computing platforms are many-core systems.

Starting from a given set of applications, a computing platform has to be *allocated* and the applications have to be *bound* and *scheduled* onto the allocated hardware resources [42]. Each application is typically assumed to consist of communicating tasks. During allocation, processing and interconnection resources including memories are selected and configured. In general, the result is a heterogeneous many-core computing platform, consisting of software-programmable processors and hardware accelerators interconnected by a network equipped with a memory hierarchy. During binding, tasks are assigned to processors and hardware accelerators as well as variables are assigned to memory locations. Moreover, transactions are routed on the interconnection resources. This step is crucial as *infeasible* implementations can be generated by binding two communicating tasks to virtually unconnected resources. The set of *feasible implementations* is denoted by $X_f \subseteq X$. Finally, scheduling resolves resource conflicts by precomputing either dedicated computing and communication times or priorities. According to the design decisions during allocation, binding, and scheduling, the set of applications can be refined. With respect to the resulting system decomposition, the design process can be continued at a lower level of abstraction [42]. The motivation for making as many decisions as possible during design time stems from the fact that embedded computing systems often have to guarantee many properties like safety, reliability, performance, etc. and, hence, demand for a high degree of predictability.

Depending on the made design decisions, the resulting system level implementations show different qualities. Typically, more than a single property is assessed to measure the quality of an implementation. Important properties are power and area consumption, throughput and response time, or mean time to failure. As an example, a video decoder implemented in a handheld system has to meet timing constraints in order to provide some quality of service, while simultaneously being power efficient. During DSE, appropriate evaluation methods have to be applied in order to *estimate* the quality (see below). Given a feasible implementation $x \in X_f$, its quality can be represented by a vector, which is commonly referred to as *quality vector* $f(x)$, where $f(x) = (f_1(x), \ldots, f_k(x))$ is a k-dimensional function consisting of k objective functions. Note that the notion of quality vectors, even if possible to compute, does not have any meaning for infeasible implementations.

Often several conflicting design goals are considered simultaneously. As a consequence, a set of *Pareto-optimal* solutions has to be found [36]. A solution is said to be Pareto-optimal, if it is not *dominated* by any other solution. For minimization problems and any two feasible implementations $x_1, x_2 \in X_f$, we say (cf. [45]):

$$\begin{array}{rrll} x_1 \succ x_2 & (x_1 \text{ dominates } x_2) & \text{iff} & \forall i : f_i(x_1) \leq f_i(x_2) \land \exists i : f_i(x_1) < f_i(x_2) \\ x_1 \sim x_2 & (x_1 \text{ is indifferent to } x_2) & \text{iff} & \forall i : f_i(x_1) = f_i(x_2) \\ x_1 \parallel x_2 & (x_1 \text{ is incomparable to } x_2) & \text{iff} & \exists i, j : f_i(x_1) > f_i(x_2) \land f_j(x_1) < f_j(x_2). \end{array}$$

This is illustrated in Figure 1(a): For design point x, the regions containing other design points by which x is dominated, which are dominated by x, and which are incomparable to x are shown. Typically, additional quality constraints $g_i(x) \leq b_i$ are imposed on each

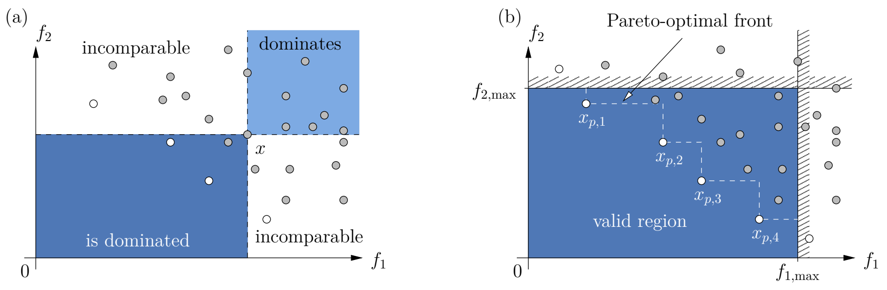


Figure 1 (a) Dominance in MOPs. (b) 2-dimensional objective space of a minimization problem.

implementation. In Figure 1(b) maximum values for both objectives f_1 and f_2 are given. Feasible implementations in X_f obeying all m constraints $g_1(x) \leq b_1, \ldots, g_m(x) \leq b_m$ are said to be *valid*, i.e., $X_v \subseteq X_f$. Among all valid implementations in X_v, the non-dominated ones are called *Pareto-optimal*. The set of Pareto-optimal implementations is denoted by $X_p \subseteq X_v$. The Pareto-optimal front Y_p is given by their corresponding quality vectors , i.e., $Y_p = \{f(x) \mid x \in X_p\}$. Without loss of generality, only minimization problems are considered in the proposal at hand.

2.2 Covering the Design Space

Nearly all approaches to DSE at the electronic system level follow the commonly known *Y-chart approach* [24] to represent the design space. Prominent examples are DOL [43], Daedalus [35], openDF [7], Octopus [5] and our approach SystemCoDesigner [23]. The Y-chart methodology starts by defining the *set of applications* and a *target architecture template*. Next, all possible *mappings* of tasks to resources in the target architecture template are defined. Such a *specification* is often represented by a graph structure [9]. Graph elements are annotated with implementation characteristics like task (worst case) execution times, required area, power, etc. These values are used by the objective functions and constraints to determine implementation properties. The values have to be provided in a preceding characterization phase [19, 25]. With this information, the exploration can be performed automatically by selecting resources from the target architecture template and by selecting the actual binding of tasks [9]. As state-of-the-art computing platforms are often heterogeneous many-core architectures including a Network-on-Chip (NoC) communication infrastructure, transaction routing is an additional complex synthesis task. A simple example is shown in Figure 2. The specification is shown in Figure 2(a). It consists of a task graph with three tasks (yellow circles) and a 3×3 meshed NoC architecture template. The mapping options are shown as green dashed edges m_1 to m_9. Figure 2(b) shows one feasible implementation. All resources and their interconnects are allocated. Task t_1 is bound onto resource $r_{1,1}$, t_2 onto $r_{2,3}$, and t_3 onto $r_{3,1}$. The transaction routing is shown by blue arrows.

The main benefit of the Y-chart approach is the opportunity to formulate the synthesis step as a selection problem [20]. As a consequence, DSE can be formally specified as a multi-objective combinatorial optimization problem [32], or for short Multi-objective Optimization Problem (MOP). With this abstraction, different optimization strategies such as *enumerative* optimization (e.g., exhaustive search), *deterministic* optimization, (e.g., hill climbing or branch and bound), or *stochastic* optimization, (e.g., simulated annealing, tabu search, or evolutionary algorithms) can be used to perform the search and, thus, to cover the design space.

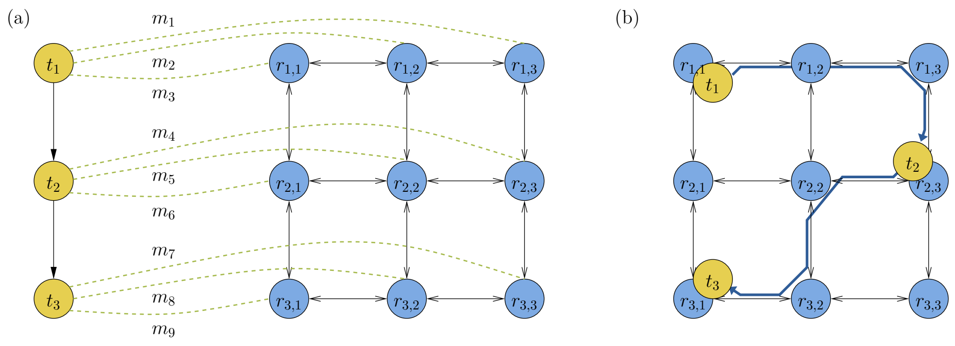


Figure 2 (a) Example of a specification and (b) a resulting implementation.

Due to the sheer size of typical design spaces, enumerative approaches are prohibitive. On the other hand, deterministic approaches often fail in the presence of non-linear objective functions and constraints. Among the stochastic approaches, population-based optimization strategies often perform best in the presence of MOPs [11]. The reason lies in the simultaneous approximation of the entire Pareto front by the individuals in the population, which preserves diversity among solutions while simultaneously converging to the true Pareto front by constantly improving good solutions. Examples of population-based optimization strategies are particle swarm optimization, ant colony optimization, and evolutionary algorithms. In particular, Multi-Objective Evolutionary Algorithms (MOEAs) [11] have been successful in the DSE domain [40]. Especially *elitist MOEAs*, which store the best found solutions in an external archive, show good properties in converging to the Pareto front [27].

Beside these advantages, MOEAs suffer from similar problems as many other stochastic optimization strategies: in the presence of design spaces only containing few feasible solutions, MOEAs spend most of their computing time in finding feasible solutions instead of optimizing feasible ones [40]. As a consequence, exploration time is not used efficiently. Due to the combinatorial nature of the optimization problem, this drawback can be alleviated by incorporating symbolic methods into MOEAs. In that case, symbolic methods are applied to perform the actual synthesis step, i.e., allocation, binding, and scheduling, which guarantees that feasible implementations are found if they exist. As a consequence, the MOEA now can focus on the optimization of feasible implementations. The idea of using symbolic methods in system synthesis is not new. In [34], a symbolic hardware/software partitioning based on Integer Linear Programming (ILP) is proposed. A symbolic system synthesis approach based on Binary Decision Diagrams (BDD) is presented in [33]. [20] presents an encoding as Boolean satisfiability problem (SAT), which enables system synthesis by programs known as SAT solvers. For this purpose, all design decisions (allocation, binding, routing, scheduling) are represented by Boolean variables z_i. Linear (feasibility) constraints $h_1(z_1, \ldots, z_l) \leq c_1, \ldots, h_n(z_1, \ldots, z_l) \leq c_n$ ensure that satisfying assignments represent design decisions leading to feasible implementations $x = \psi(z_1, \ldots, z_l) \in X_f$, where ψ is the decoding function that transforms a satisfying variable assignment into the corresponding feasible implementation. Nevertheless, all these approaches do not perform a Multi-Objective Optimization (MOO) as required by an unbiased DSE. An effective way to split the work among a SAT solver and the MOEA was presented in [31]: The MOEA triggers the SAT solver to generate a new satisfying assignment to the variables z_i (which represents a feasible implementation x) if possible. Afterwards, the MOEA computes the quality vectors and checks the quality constraints $g_i(x) \leq b_i$. In all, the DSE can be modeled as an MOP:

$$\begin{array}{rll} \min & (f_1(\psi(z_1,\ldots,z_l)),\ldots,f_k(\psi(z_1,\ldots,z_l))) & \text{Checked by the MOEA} \\ \text{subject to} & g_1(\psi(z_1,\ldots,z_l)) \leq b_1,\ldots,g_m(\psi(z_1,\ldots,z_l)) \leq b_m & \text{Checked by the MOEA} \\ & h_1(z_1,\ldots,z_l) \leq c_1,\ldots,h_n(z_1,\ldots,z_l) \leq c_n & \text{Checked by the SAT solver} \end{array}$$

In the following, we will use the terms *quality constraints* and *feasibility constraints* in order to distinguish both kinds of constraints $g_i(\psi(z_1,\ldots,z_l)) \leq b_i$ and $h_i(z_1,\ldots,z_l) \leq c_i$, respectively.

2.3 Decision Procedures

As system design problems are becoming more and more stringent, the identification of feasible implementations in X_f is gaining importance in DSE. The formulation of the underlying system synthesis task as a Boolean satisfiability problem increases the interest in techniques for finding satisfying variable assignments. SAT solvers have been successfully applied to system verification in the past. Their success is largely boosted by the significant progress in Boolean constraint technology, often performing successfully even on huge instances with millions of variables and clauses. Though rooted in the classical DPLL algorithm, modern SAT solvers are mostly based on *Conflict-Driven Constraint Learning* (CDCL); see [8]. While both rely on unit propagation, CDCL basically extends DPLL by backjumping and constraint learning. Further essential supporting roles are played by dynamic conflict driven heuristics, lazy data structures, and restart policies (cf. [8]). Meanwhile, this advanced Boolean constraint technology is also used in many neighboring areas, like Maximum SAT (MAXSAT; [28]), Pseudo Boolean solving (PB; [38]), as well as Answer Set Programming (ASP; [4]).

A major weakness of the SAT-based approaches is, however, that reachability cannot be natively expressed. As a consequence, multi-hop communication has to be encoded as a sequence of communication steps, leading to unnecessarily huge (Pseudo-)Boolean formulas $h_i(z_1,\ldots,z_l) \leq c_i$ and, thus, long solving times. This is especially true for computing platforms with many different routing options, as we have shown for meshed-based NoCs [2, 3]. An approach similar to that of SAT yet directly supporting reachability is *Answer Set Programming*. ASP is an alternative approach to Boolean constraint solving tailored to knowledge representation and reasoning. As such, it combines a rich yet simple modeling language with advanced Boolean constraint technology. ASP's first-order language does not only offer scalability in terms of modeling and maintenance but moreover provides advanced language constructs like cardinality and weight constraints as well as optimization constructs. As a consequence, ASP's solving capacities do not only match the high performance of modern SAT solvers, but go well beyond clause-oriented satisfiability testing in integrating pseudo-Boolean constraints as well as optimization. The aforementioned representational edge of ASP over SAT is due a more stringent semantics that allows for more succinct Boolean problem representations [29]. Moreover, full-fledged ASP allows for solving all search problems in NP^{NP} in a uniform way. Given this expressiveness, it cannot only be used for computing feasible implementations in X_f but principally to even identify Pareto-optimal ones.

Finally, ASP solvers feature a whole spectrum of combinable reasoning modes surpassing satisfiability testing, among them, different forms of enumeration of solutions, intersection or union, as well as multi-criteria and -objective optimization. Notably, ASP supports polynomial space enumeration algorithms [12], which allows us to enumerate Pareto frontiers without risking an exponential blow-up in memory.

2.4 Evaluating Design Points

Independent of the selected optimization strategy, different design points can be constructed from the specification. Each of these solutions can be evaluated regarding feasibility and different objective functions. Important objective functions are power and area consumption, throughput and response time, or mean time to failure. In particular, assessing performance in terms of throughput and response time is often critical, as it is a foundation for determining other system properties like power efficiency and reliability. In our research, we are not going to develop new performance estimation methods. Instead, we rely on existing ones[1] and focus on a different problem: After evaluating a single design point x, it is known whether it obeys the quality constraints $g_i(x) \leq b_i$. If so, the implementation is called *valid*, otherwise *invalid*. All the above presented DSE approaches suffer from poor solving times if only a small fraction of feasible solutions is valid, i.e., $|X_v| << |X_f|$. The reason lies in an insufficiently strong feedback to the stochastic optimization method. Often a weak feedback exists by *punishing* invalid solutions by assigning uncompetitive objective values to them. However, as a consequence, invalid solutions might still be revisited again and again. A better strategy is to incorporate knowledge about the validity into the search process. Ideally, the decision procedure is used for this purpose. For constraints, which could be represented as continuous programming models, the classical Benders' decomposition [6] can be used. Benders' decomposition is a common method for solving mixed logical linear problems, where Boolean *indicator variables* are used to link different constrained problems. Thus, huge propositional logic formulas can be avoided. However, embedded systems design usually relies on combinatorial optimization. In this case Logic-Based Benders Decomposition (LBBD) could be used instead [21]. Its application to system synthesis is shown first in [39].

As LBBD only allows to test complete and consistent assignments of indicator variables, inconsistencies in linked programming models are thus lazily detected. This is avoided by using Satisfiability Modulo Theories (SMT; [8]) solvers, which permit working on partial assignments of indicator variables. Hence, larger regions of inconsistent assignments can be pruned and the search process is accelerated. This, however, requires monotonic constraints [1]. Unfortunately, this is not the case in embedded systems design, e.g., adding tasks to a partial implementation might decrease the response time. SMT is widely accepted in the domain of hardware and software verification. In SMT solving, a formula is tested for satisfiability with respect to a given *background theory*, e.g., Linear Real Arithmetics [41], Equality and Uninterpreted Functions [10]. SMT solvers are today typically *indirect* solvers, i.e., they are traditionally combinations of SAT solvers with background theory solvers. The SAT solver controls the solving process and assigns values to regular Boolean as well as indicator variables in the background theory. The background solver afterwards tries to find a corresponding variable assignment in the background theory to match the assignment of indicator variables. If a conflict is detected in the background theory, the reason could be learned by the SAT solver via the indicator variables.

In [30], the usage of SMT solving in systems synthesis is shown. The authors use a latency computation as background theory and perform optimization by a branch and bound strategy incorporated into the SMT solver. The system model, however, is based on a simple application model and communication architecture. Moreover, the proposed optimization is only applicable to single-objective optimization problems. Another SMT-based approach

[1] To be more precise, we consider Scenario-Aware Data Flow Graphs (SA-DFGs) [13] as application model of computation. For SA-SDFGs a performance analysis based on (max,+)-algebra exists [15, 16].

to synthesis is proposed in [22]. The underlying platform is a time-triggered architecture. As background theory, the authors use linear arithmetics for adding worst-case execution times. Thus, they stick with linear (monotonic) quality constraints. It was shown in [37] how to use Modular Performance Analysis (MPA) [44] as background theory to test real-time constraints, and, hence, how to integrate non-monotonic quality constraint checking into a SAT-based symbolic synthesis approach. Another group shows in [26] how SMT-solving with MPA as background theory can be used to compute processor frequency settings to meet delay, buffer, and energy constraints.

2.5 Assessing Exploration Quality

When developing different optimization approaches, it becomes mandatory to define appropriate performance measures to compare these approaches. In MOO, there are two different goals which must be considered when assessing optimization strategies: (1) The *convergence* towards the true Pareto-optimal front and (2) the *diversity* of the found non-dominated solutions [11]. In [45], a framework for comparing different performance assessment methods for multi-objective optimizers is presented. As a key result, it has been shown that binary quality indicators have to be used in order to decide whether an approximation set computed by an optimization strategy is better than one computed by another optimization strategy.

One of the best known binary quality indicators is ϵ-dominance [27]: A quality vector a is said to *weakly ϵ-dominate* (in a minimization problem) a quality vector b, denoted by $a \succeq_\epsilon b$, if and only if $a \succeq \epsilon \cdot b$. By scaling quality vector b by a factor ϵ, quality vector a is superior to quality vector b. Complementary to ϵ-dominance, which is primarily used to measure convergence, we use *entropy* [18] to measure diversity and keep diversity high when selecting representative design points.

3 Goal of the research

The state-of-the-art section has presented in detail that today's Design Space Exploration (DSE) approaches at the Electronic System Level (ESL) have the following drawbacks:

1. Often complex multi-hop communication is not supported. This neglects state-of-the-art computing platforms like many-core systems. Approaches that support multi-hop communication fail in the presence of computing platforms with many routing options, as can be typically found in Networks on Chip (NoCs).
2. Typically, no strong feedback from constraint checking to the optimization strategy exists. As a consequence, invalid solutions might be revisited again and again. This significantly lowers the exploration performance, which is particularly problematic when designs are becoming more stringent.
3. The specification of the target architecture template and all mapping options is a time consuming task. On the other hand, this specification allows to formulate the system synthesis problem as a selection problem and, thus, the automatic DSE.

From these shortcomings, we derive the following objectives from the perspective of electronic design automation:

O1: *Accelerate* DSE by integrating routing computation and dominance checking into the decision procedure.

O2: *Extend the applicability* of DSE at ESL by tightly incorporating non-monotonic quality constraint checking.

O3: *Improve the usability* of DSE at ESL by moving from selective methods to novel generative approaches.

From the viewpoint of Answer Set Programming (ASP), the general objective is to *invent new solving strategies* inspired from novel application-specific problems. More specifically, (i) *the integration* of application-specific knowledge and strategies into ASP solving should be *improved* and (ii) *the applicability* of ASP towards robust Multi-Objective Optimization (MOO) should be *extended*.

4 Current status and preliminary results of the research

Right now, the main focus is on exploring technologies and techniques to efficiently implement *O1-O3*. While no new publications have been made for Design Space Exploration specifically, the following contributions laid the groundwork for future applications:

Theory Solving made easy with Clingo 5 by M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and P. Wanko to appear as Technical Communication in ICLP'16. The new theory framework in *Clingo 5* allows for a tight coupling of decision procedures and efficient Boolean constraint solving. As an example, Difference Logic is implemented in the paper which is an efficient theory to implement temporal constraints which can be used to encode the scheduling needed in DSE.

Computing Diverse Optimal Stable Models by J. Romero, T. Schaub, and P. Wanko to appear as Technical Communication in ICLP'16. The paper introduces a system to pose queries over and enumerate diverse optimal solutions. This can be used for covering the Design Space and finding representative Pareto optimal solutions.

5 Open issues and expected achievements

We expect to achieve the following during our research:

1. ASP-Based Synthesis that includes Encodings for Many-Core Synthesis of Streaming Applications
2. Application-Specific Search and Enumeration Methods Based on ASP
3. Application-Specific Multi-Objective Optimization based on ASP
4. Application-Specific Theory Solving

References

1 Santosh G. Abraham, B. Ramakrishna Rau, and Robert Schreiber. Fast Design Space Exploration Through Validity and Quality Filtering of Subsystem Designs. Technical report, Hewlett Packard, Compiler and Architecture Research, HP Laboratories Palo Alto, July 2000.

2 B. Andres, M. Gebser, M. Glaß, C. Haubelt, F. Reimann, and T. Schaub. A combined mapping and routing algorithm for 3D NoCs based on ASP. In C. Haubelt and D. Timmermann, editors, *Sechzehnter Workshop für Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV'13)*, pages 35–46. Institut für Angewandte Mikroelektronik und Datentechnik, Universität Rostock, 2013.

3 B. Andres, M. Gebser, M. Glaß, C. Haubelt, F. Reimann, and T. Schaub. Symbolic system synthesis using answer set programming. In P. Cabalar and T. Son, editors, *Proceedings of the Twelfth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'13)*, volume 8148 of *Lecture Notes in Artificial Intelligence*, pages 79–91. Springer, 2013.

4 C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.

5 T. Basten, M. Hendriks, L. Somers, and N. Trcka. Model-Driven Design-Space Exploration for Software-Intensive Embedded Systems. In *Proceedings of the International Conference on Formal Modeling and Analysis of Timed Systmes (FORMATS)*, pages 1–6, 2012.

6 J. F. Benders. Partitioning Procedures for Solving Mixed-Variables Programming Problems. *Numerische Mathemathik*, 4(3):238–252, 1962.

7 S. Bhattacharyya, G. Brebner, J. Janneck, J. Eker, C. von Platen, M. Mattavelli, and M. Raulet. OpenDF: A Dataflow Toolset for Reconfigurable Hardware and Multicore Systems. *ACM SIGARCH Computer Architecture News*, 36(5):29–35, 2009.

8 A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

9 T. Blickle, J. Teich, and L. Thiele. System-Level Synthesis Using Evolutionary Algorithms. In *Design Automation for Embedded Systems*, 3, pages 23–62. 1998.

10 J. R. Burch and D. L. Dill. Automatic Verification of Pipelined Microprocessor Control. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*, pages 68–80, 1994.

11 K. Deb. *Multi-Objective Optimization using Evolutionary Algorithms*. John Wiley & Sons, Inc., Chichester, New York, Weinheim, Brisbane, Singapore, Toronto, 2001.

12 M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Conflict-driven answer set enumeration. In C. Baral, G. Brewka, and J. Schlipf, editors, *Proceedings of the Ninth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07)*, volume 4483 of *Lecture Notes in Artificial Intelligence*, pages 136–148. Springer, 2007.

13 M. Geilen and S. Stuijk. Worst-Case Performance Analysis of Synchronous Dataflow Scenarios. In *Proceedings of the Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 125–134, 2010.

14 A. Gerstlauer, C. Haubelt, A. D. Pimentel, T. P. Stefanov, D. D. Gajski, and J. Teich. Electronic System-Level Synthesis Methodologies. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(10):1517–1530, 2009.

15 A. H. Ghamarian, M. C. W. Geilen, S. Stuijk, T. Basten, A. J. M. Moonen, M. J. G. Bekooij, B. D. Theelen, and M. R. Mousavi. Throughput Analysis of Synchronous Data Flow Graphs. In *Proceedings of the International Conference on Application of Concurrency to System Design (ACSD)*, pages 25–36, 2006.

16 A. H. Ghamarian, S. Stuijk, T. Basten, M. C. W. Geilen, and B. D. Theelen. Latency Minimization for Synchronous Data Flow Graphs. In *Proceedings of the Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD)*, pages 189–196, 2007.

17 M. Gries. Methods for Evaluating and Covering the Design Space during Early Design Development. *Integration, The VLSI Journal*, 38(2):131–183, 2004.

18 S. Gunawan, Ali Farhang-Mehr, and Shapour Azarm. Multi-Level Multi-Objective Genetic Algorithm Using Entropy to Preserve Diversity. In *Proceedings of the International Conference on Evolutionary Multi-Criterion Optimization (EMO)*, pages 148–161, 2003.

19 W. Haid, M. Keller, K. Huang, I. Bacivarov, and L. Thiele. Generation and Calibration of Compositional Performance Analysis Models for Multi-Processor Systems. In *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, pages 92–99, 2009.

20 C. Haubelt, J. Teich, R. Feldmann, and B. Monien. SAT-Based Techniques in System Design. In *Proceedings of the Design, Automation and Test in Europe (DATE)*, pages 1168–1169, 2003.

21 J. N. Hooker and G. Ottosson. Logic-Based Benders Decomposition. *Mathematical Programming*, 96(1):33–60, 2003.

22 E. Jackson, E. Kang, M. Dahlweid, D. Seifert, and T. Santen. Components, Platforms and Possibilities: Towards Generic Automation for MDA. In *Proceedings of the International Conference on Embedded Software (EMSOFT)*, pages 39–48, 2010.

23 J. Keinert, M. Streubühr, T. Schlichter, J. Falk, J. Gladigau, C. Haubelt, J. Teich, and M. Meredith. SystemCoDesigner – An Automatic ESL Synthesis Approach by Design Space Exploration and Behavioral Synthesis for Streaming Applications. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 14(1):1–23, 2009.

24 B. Kienhuis, E. Deprettere, K. Vissers, and P. van der Wolf. An Approach for Quantitative Analysis of Application-Specific Dataflow Architectures. In *Proceedings of the Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, pages 338–349, 1997.

25 R. Kiesel, M. Streubühr, C. Haubelt, O. Löhlein, and J. Teich. Calibration and Validation of Software Performance Models for Pedestrian Detection Systems. In *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, pages 182–189, 2011.

26 P. Kumar, D. B. Chokshi, and L. Thiele. A Satisfiability Approach to Speed Assignment for Distributed Real-Time Systems. In *Proceedings of the Design, Automation and Test in Europe (DATE)*, pages 749–754, 2013.

27 M. Laumanns, L. Thiele, K. Deb, and E. Zitzler. Combining Convergence and Diversity in Evolutionary Multi-Objective Optimization. *Evolutionary Computation*, 10(3):263–282, 2002.

28 C. Li and F. Manyà. MaxSAT. In Biere et al. [8], chapter 19, pages 613–631.

29 V. Lifschitz and A. Razborov. Why are there so many loop formulas? *ACM Transactions on Computational Logic*, 7(2):261–268, 2006.

30 W. Liu, Z. Gu, J. Xu, X. Wu, and Y. Ye. Satisfiability Modulo Graph Theory for Task Mapping and Scheduling on Multiprocessor Systems. *IEEE Transactions on Parallel and Distributed Systems*, 22(8):1382–1389, 2011.

31 M. Lukasiewycz, M. Glaß, C. Haubelt, and J. Teich. SAT-Decoding in Evolutionary Algorithms for Discrete Constrained Optimization Problems. In *Proceedings of the Congress on Evolutionary Computation*, pages 935–942, 2007.

32 M. Lukasiewycz, M. Glaß, C. Haubelt, and J. Teich. Efficient Symbolic Multi-Objective Design Space Exploration. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, pages 691–696, 2008.

33 S. Neema. *System Level Synthesis of Adaptive Computing Systems.* PhD thesis, Vanderbilt University, Nashville, Tennessee, 2001.

34 R. Niemann and P. Marwedel. An Algorithm for Hardware/Software Partitioning Using Mixed Integer Linear Programming. *Design Automation for Embedded Systems*, 2(2):165–193, 1997.

35 H. Nikolov, M. Thompson, T. Stefanov, A. D. Pimentel, S. Polstra, R. Bose, C. Zissulescu, and E. F. Deprettere. Daedalus: Toward Composable Multimedia MP-SoC Design. In *Proceedings of the Design Automation Conference (DAC)*, pages 574–579, 2008.

36 V. Pareto. *Cours d'Économie Politique*, volume 1. F. Rouge & Cie., 1896.

37 F. Reimann, M. Glaß, C. Haubelt, M. Eberl, and J. Teich. Improving Platform-Based System Synthesis by Satisfiability Modulo Theories Solving. In *Proceedings of the Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 135–144, 2010.

38 O. Roussel and V. Manquinho. Pseudo-Boolean and cardinality constraints. In Biere et al. [8], chapter 22, pages 695–733.

39 N. Satish, K. Ravindran, and K. Keutzer. A Decomposition-Based Constraint Optimization Approach for Statically Scheduling Task Graphs with Communication Delays to Multiprocessors. In *Proceedings of the Design, Automation and Test in Europe (DATE)*, pages 57–62, 2007.

40 T. Schlichter, C. Haubelt, and J. Teich. Improving EA-based Design Space Exploration by Utilizing Symbolic Feasibility Tests. In *Proceedings of Genetic and Evolutionary Computation Conference> (GECCO)*, pages 1945–1952, 2005.

41 H. M. Sheini and K. A. Sakallah. A Scalable Method for Solving Satisfiability of Integer Linear Arithmetic Logic. In *Theory and Applications of Satisfiability Testing*, pages 241–256, 2005.

42 J. Teich and C. Haubelt. *Digitale Hardware/Software-Systeme – Synthese und Optimierung*. Springer, Berlin, Heidelberg, 2007. 2. erweiterte Auflage.

43 L. Thiele, I. Bacivarov, W. Haid, and K. Huang. Mapping Applications to Tiled Multiprocessor Embedded Systems. In *Proceedings of the International Conference on Application of Concurrency to System Design (ACSD)*, pages 29–40, 2007.

44 L. Thiele and E. Wandeler. Performance Analysis of Distributed Embedded Systems. In *Embedded Systems Handbook*, pages 15.1–15.18. CRC Press, Boca Raton, FL, 2006.

45 E. Zitzler, L. Thiele, M. Laumanns, C. Fonseca, and V. Grunert da Fonseca. Performance Assessment of Multiobjective Optimizers: An Analysis and Review. *IEEE Transactions on Evolutionary Computation*, 7(2):117–132, 2003.